Evolutionnary Neural Network Tools for Life Emergence and More

F. Furfaro

2021

Abstract

L'idée de vie artificielle remonte au année 1940 par principe d'autoréplication, meme époque ou les concepts sur l'intelligence artificielle sont crées. Nous proposons un modele à définition incomplete, mais suffisante pour etudier la fonctionnalisation des structure cerebrale lors de l'evolution. On combine l'apprentissage par renforcement et la modification aléatoire des connections intercouche. Pour tester notre modele, le jeu du chat et la souris est un exemple de choix pour etudier l'evolution des comportement basique de proie/predateurs, tout en etant assez complet pour une approche évolutive. Ces outils n'ont pas la prétentions de reproduire la vie sur ordinateurs, mais peuvent donner des idée, minime soit elle, pour l'optimisation des structure de reseau de neurones et d'avancer vers la creation de vie articiel. En effet, lors du processus d'apprentissage (evolution + descente de gradient), on on observe par l'apparition de structure et stratégie convergente. Ces structures sont conservé pendant un grand de cycle alors meme que des reseau "challenger" sont toujours ajouter à chaque cycle de reproduction. En dehors, on observe que ce type d'algorithme peut s'averer efficasse avec moins de neurone pour la classification MNIST.

Introduction

La vie est apparu il y a 3,8 milliards d'année et n'a pas cessé d'évoluer et de coloniser l'environnement [5]. Elle a commencer de maniere unicellulaire sans noyau [3, 7], à un large panel de forme avec l'appartition du noyau [4, 6] et de la multicellularité [2]. Pour autant, il est assez difficiele de definir ce qui est vraiment vivant et cette question est toujours débatu. L'une des definition les plus simple est celle donnée par la NASA lors des programme de recherche de vie extra-terrestre : Un systeme chimique auto-entretenu capable d'évolution darwinienne. L'evolution darwinienne est un principe fondamentale dans l'etude des systeme complexe disposant d'heredité, de variation et de selection. On ne le retrouve pas seulement en biologie, mais on peut l'obervers dans l'evolution des courants musicaux, artistique, philosophique, technologique, etc. Une autres définition de la Vie existe, et elle est plus lié à l'une des transformation thermodynamique fondamentale : L'entropie. La vie serait : Une structure dissipative capable d'auto-catalyse, d'homéostatie et d'apprentissage [1]. Ce qui diminu localement l'entropie en favorisant l'auto-organisation. Mais ces deux definitions se restreigne à un systeme "chimique", est ce qu'un robot dotée d'une intelligence, voirent d'emotion humaine serait vivant dans cette définition ? (tout probleme n'est pas définissable). Aujourd'hui, il existe un grand nombre de simulation de vie sur ordinateurs (liste de 2-3) et certaine sont encore en cours d'execution depuis 1990. Et son d'une grande importance pour comprendre l'emergence de la vie (avec les experiences de cellules minimaliste)

AUTOMATE : Dans notre cas, nous allons essayer de reproduire un systeme vivant à l'intersection de ses deux definitions. Un systeme auto-entretenu capable d'apprentissage evolutif. Qu'on restreindra dans un premiers lieux à "un systeme capable d'apprentissage évolutif" pour en developper les outils futurs. Pour cela on a creer un enviromment de jeu simple : le jeu du chat et la souris, alternant ainsi deux comportements : proie/prédateur. On se base sur deux outils : les automates cellulaires et les reseaux de neurones évolutifs. Les automates sont a l'origne des objets mathématique permettant de resoudre des probleme de decidabilité mathématique [Von Newman]. On en retrouva ensuite des utilisation experiementale comme le plus connu "jeu de la vie" [Conway], qui à partir de 2 regles simple, faire emerger une forte complexité, voir meme reproduire une machine de Turing [ref à revoir]. Le L'état d'une cellule au temps t+1 est fonction de l'état au temps t d'un nombre fini de cellules appelé son « voisinage ». Le probleme est donc une grille ou l'on a des case "automates" qui se deplace dans la grille, elle a un adversaire qui va chercher soit à "attraper" l'automate, soit la "fuir". La decision de deplacement de l'automate est controler/réguler par un reseau de neurone evolutif par principe d'apprentissage par renfocement. Les propriété de

l'automate changeront ainsi à chaque génération, aussi bien parametre de "vue" de l'automate que les actions ou la structure du réseau.

STATISTIQUE : Les reseaux neuronaux ont ete construit à partir du modele de neurone biologique, le neurone formel [McCulloch & Pitts], ou l'on constate qu'une neurone est une fonction "affine" avec une fonction d'activation. Aujourd'hui il s'est rapproché des methodes statistique non inductive (descriptive : on ne sait pas la loi de probabilité) pour resoudre des probleme de régression, classification et partitionnement. Les principaux outils d'analyse de grosse donnée en statistiques sont, pour ne citer que les plus connu : La PCA, analyse de composante principale, qui permet de projeter l'ensemble des points sur les regression multilinéaire des variable corrélé entre elle, ce qui reduit la dimension [Karl Pearson], Les SVM, séparateurs à vaste marge, generalisant les classificateur linéaire par des vecteurs supports (théorie de la « régularisation statistique » + surapprentissage) [Vapnik] et enfin Le K-mean qui divise un ensemble de points par minimisation d'un certain nombre de graine barycentrique [Steinhauss, Llyod]. Tout ces outils encore utilisé aujourd'hui, sont peu à peu remplacé par des reseau de neurone vers 2000s apres le succes de l'apprentissage profond en compétition.

APPRENTISAGE : Le question de l'apprentissage a été étudié au cours des année 1950, ou il a pu emerger la regles de Hebbs [Hebbs 1949] qui modifier simplemnt la valeurs des coefficient synaptique d'un reseau simple couche, il eue un declin en fin 1970 par l'impossibilité de reseont probleme non linéaire XOR ou connexité [Minsky 1969]. Mais ce probleme fut resolu par les perceptron multicouche, mais cette fois la modification des poids était plus compliqué. C'est le systeme de retropropagtion du gradient [1984] qu'on retrouve aussi dans le cerveau [Stuart 1997] et l'algorithme du gradient, par minimisation itérative de l'erreurs, qui permit de resoudre ce probleme. En parallele, inspirer par les neurone moteurs de la vue, on devellopa les reseau de neurone convolutif qui reduit fortement le calcul des poids en entrée du reseau, car n'est pas entiereement connecté [Lecun, mais pas que]. Le souci de ce genre de reseau est qu'il faut un grand nombre de donnée pour adapté les poids, pour reseoudre un probleme de decision markovien, il y eu l'apparition du Q-learning convergent par l'apprentissage des récompense [Watkins 1989 Dayan 1992] qui ont encore était amélioré par des reseau adversariaux "actor-critic" à l'image des reseau génératif, il permette d'accelerer la convergence des poids [Konda 2003].

FONCTIONALISATION : Néamoins, ces reseaux optimise l'apprentissage du reseau mais ne cherche pas à maximiser la structure des couches du reseau. Seule les reseau de neurone recurant ont été le plus traité car il y avait des probleme de "vanishing gradient", surtout pour le traitement du texte. Une autre catégorie de reseau est apparu dans la fin des années 1990, les reseau NEAT, ou les poids et les connexion sont "appris" par un processus de selection. Hors le cerveau n'a pas des poids qui ont été selectionné par l'evolution, mais ce qui est le cas la structure [Ref]. L'idée est donc de s'impirer des donnée en neuroscience recente, ou il est montré le principe de fonctionnalisation cerebrale que le cerveau est pré-cablé pour voir certain danger ou encore marcher à la naissance [araignées Rakison, D. H. & Derringer, J 2008 et sommeil] et il n'y a pas eu besoin d'avoir un temps d'apprentissage long. A cette image, on a donc develloper un algorithme ou les couches interconnecté de maniere aléatoire de façon à fonctionnaliser le probleme "chat-souris" aussi bien dans la structure que dans l'apprentissage par renforcement.

Méthodes

L'ensemble du projet est disponible sur le lien `github.com/fabienfrfr`. Le projet a été codé intégralement en Python de facon à utiliser efficacement les bibliothèques logiciel : PyTorch pour les reseaux de neurone, Keras-Tensorflow pour la base de donnée MNIST, Numpy et Scipy pour les calculs scientifique, Pandas pour l'analyse et le stockage des données, Networkx pour la representation et l'analyse des Graphes et enfin Matplotlib pour la visualisation et l'animation des données. On distingue 6 fichiers d'experience, 2 fichiers d'analyse de donnée (DATA_ANALYSIS et EXTRA_FUNCTION) et un fichiers de debuggage modulaire. Dans les fichiers "experience" on retrouve au plus bas niveau GRAPH_EAT encapsulant GRAPH_GEN, generant la liste des connections pour chaque couche de neurone, ainsi que pRNN, générant la structure du reseau en fonction de la liste d'adjacence. À plus haut niveau, on retrouve AGENT qui contient les propriété d'entrée/sortie de l'agent et son algorithme d'apprentissage et de mémorisation, ainsi que TAG-ENV contenant cette fois les regles du jeu "chat-souris". Enfin on a le MAIN qui va lister l'ensemble des agents et environnement associé pour lancer les experiences d'evolution. On retrouve aussi un fichier LOG qui permet de stocker au format *csv* les experience.

1 Reseau de neurone évolutif

Le perceptron est un algorithme d'apprentissage fonctionnant comme une fonction de seuillage et decrivant le neurone formel. Une fonction de seuil est un classifieurs linéaire qui a pour entrée un vecteurs à valeurs réelle (virgule flottante) et une valeurs f(z) en sortie, qui peut etre suivant la fonction d'activation : binaire (Heaviside), reel (Sigmoid) ou entre les deux (ReLu). Le perceptrons se represente par le produit scalaire entre le vecteurs d'entrée et le vecteurs poids des neurones, soit l'application multilinéaire suivante :

$$o = f(z) = \begin{cases} 1 & si \quad \sum_{i=1}^{n} w_i x_i > \theta \\ 0 & sinon \end{cases}$$

Ce qui s'ecrit dans la plupart des bibliothèques logiciels comme la sequence d'un neurone à **"n"** entrée, suivie d'une fonction d'activation.

Code PyTorch :

```
Layers = nn.Sequential(nn.Linear(N, 1), nn.ReLU())
x = Layers(x)
```

Le perceptrons est un classifieurs linéaire, il converge uniquement si l'ensemble des donnée d'entrée sont linéairement séparable (droite). Mettre à jours les poids **"w"** correspond à l'apprentissage du perceptrons, la *regles delta* [Russell Ingrid 2002] est la plus simple et consiste à comparer linéairement la valeurs obtenu par la sortie du reseau **"z"** et la valeurs attendu par le reseau de neurones **"d"**. Il existe d'autre methodes plus sophistiqué ou limitant les probleme d'annulation tout aussi simple (Least Mean Square), mais ici présenté, la regle delta se presente sous la forme :

$$w_i(t+1) = w_i(t) + (d - z)x_i$$

Dans la plupart des bibliothèques logiciels, on precise le type de calcul de perte (loss ou criterion) avant la boucle d'apprentissage, ainsi que l'algorithme de calcul de la descente de gradiant (optimizer). Les algorithme les plus utilisé sont SGD "Stochastic Gradiant Descent" plutot adapté à classification et "Adam" plutot adapté au problème de regression [ref car pas sûr].

Code PyTorch :

```
LOSS = torch.nn.MSELoss(reduction='sum')
optimizer = optim.SGD(Layers.parameters())
### Loop
LOSS = LOSS(x_pred, x)
optimizer.zero_grad()
LOSS.backward()
optimizer.step()
```

Un reseau de neurone correspond à une interconnexion entre couche de neurone contenant **"n"** perceptrons. Une unique couche de **"n"** perceptron corresponds au reseau le plus simple et permet de resoudre des probleme linéairement séparable de type hyperplan. Comme toute les entrée **"x"** sont interconnecté au **"n"** neurone, les vecteurs poids **"w"** forme une matrice **"m*n"** avec **"m"** le nombre d'entrée au réseau. La sortie s'exprime dans ce cas :

$$\vec{y} = \begin{bmatrix} w_{0,0} & & \\ & & \\ & & w_{m,n} \end{bmatrix} . \begin{bmatrix} x_0 & \dots & x_m \end{bmatrix} + \begin{bmatrix} b_0 \\ \\ b_n \end{bmatrix}$$
$$\vec{s} = f(\vec{y})$$

Pour que l'algorithme mette à jours efficassement les poids du réseau, on utilise un certain lot de donnée d'apprentissage, qu'on appele "batch", soit il contient l'ensemble des données de l'experience, soit il est réparti en "mini-batch" pour eviter la sur-utilisation de la mémoire et augmente la vitesse de convergence de la vallé de stabilité, le taille du lot ne doit pas etre trop petit non plus car augmente le bruit [ref mini-batch]. Dans ce cas, le vecteurs d'entrée devient une matrice **"m*b"**, et la matrice des poids devient un tenseurs d'ordre 3 **"n*m*b"**. La plupart des bibliothèques logiciels s'adapte automatiquement à la taille du batch, et n'a pas besoin d'etre spécifié.

Code PyTorch :

```
Layers = nn.Sequential(nn.Linear(N, M), nn.ReLU())
x = Layers(x)
```

Pour autant, les reseaux linéaire ne sont pas les seules à etres employé et ne sont pas forcement les plus adapté pour les problemes d'images [ref]. En effet, une image est une donnée 2D dont les données spatiale sont localement lié, mais les informations ne sont pas interconnecté, c'est pour cela qu'il existe les reseaux de convolution. Ces reseau ont l'avantage d'etre interconnecté mais par fenetre "spatiale" ce qui reduit fortement le nombre de calculs pour l'actualisation des poids. Elle est composé de **"N"** perceptron, mais ne represente plus comme un produit tensoriel, mais comme un produit de convolution où les entrée sont moyenné pondérément par le glissement **"C"** d'une fenetre de taille **"k"** sur les poids **"w"**. La sortie s'exprime de la sorte :

$$out(N_i, C_{out_j}) = bias(C_{out_j}) + \sum_{k=0}^{C_{in}-1} weight(C_{out_j}, k) \star input(N_i, k)$$

Dans le cas le plus simple, on peut considerer qu'il y a qu'une seule entrée par neurone, pour cela, le nombre d'entrée est egal au nombre de sortie du canal de la couche. Ensuite, la taille de la fenetre est de **"1"** et le groupement de connexion doit etre egal au nombre de neurones, qui est un nombre divisible de l'entrée et de la sortie par lui meme. Ce type parametrage est equivalent à **N** neurones indépendants à 1 seules connexion chacune. Contrairement au réseau linéaire, la taille du batch doit etre specifié lorsqu'on donne les données d'entrée à la couche de convolution.

Code PyTorch :

```
Layers = nn.Sequential(nn.Conv1d(I, I, 1, groups=I), nn.ReLU())
x = Layers(x.view(BATCH_SIZE, I, 1)).view((BATCH_SIZE, I))
```

1.1 Fonctionnalisation

S'inspirer du cerveaux "modèle" pour construire des algorithmes neuronaux à toujours été present dans le developpement de nouvelles methodes en *Intelligence Artificielle*. Dans ce projet, nous nous sommes inpiré de la specialisation fonctionnele du cerveau. En effet, le moyen le plus efficace de réaliser des fonctions différentes c'est de confier la réalisation de chaque fonction à un outil particulier et spécialisé [Tooby, J. & Cosmides, 1992 ; Tooby, J. & Cosmides, 2015]. C'est pour cette raison que la sélection naturelle (survie, reproduction) a organisé notre corps en différents organes qui sont chacun spécialisés, mais aussi dans le domaine de la cognition [Confer, J. C. et al. 2010]. La specialisation fonctionnele du cerveau est observé empiriquement par plusieurs troubles neuropsychologique, comme par exemple avec le syndrome de Capgras, ou une defaillance d'une region entraine la perte de reconnaissance des visage [Thomas Antérion, 2008].

Dans notre cas, nous avons simplifié le plus possible un cerveau, où l'on a en entrée des neurones de "vision", suivi de couches sequentielle "fonctionnelle" et en sortie, des neurones "moteurs". On distingues des neurones stables, les neurones d'entrée et de sortie qui ne changeront pas pendant le processus évolutifs. Les neurones de "visions" sont à l'image des cones de la rétines càd des neurones connecté à des batonnée de vue unique [Dale Purves, G-J Augustine, 2005] qu'on represente dans notre cas comme une couches de convolution "simple sparse" : une seule entrée par neurones. Les neurones "moteurs" quand à eux sont analogue au motoneurones qui active la contraction musculaire et donc le mouvement des organismes [Fitzpatrick, D. (2001) The Primary Motor Cortex], on les represente dans notre cas comme une couche linéaire de neurones où les sortie seront lié au action du reseau de neurone complet.

La vrai particularité du projet est que les couches intermediaire "fonctionnalisé" ne sont pas contruite manuellement, mais adapté par un processus evolutif de selection. Ainsi, l'objectif est de faire emerger un structure intermediaire fonctionnalisé à notre probleme, ici le "jeu du chat et la souris". Les connections intermediaires sont initialement aléatoire avec au minimum une connection entre la couche "n+1" et "n", pour avoir toujours un lien entre l'entrée et la sortie et ne pas avoir de probleme lors du calculs de poids des neurones. Au cours de l'apprentissage, on s'attend à ce que le reseau soit de plus en plus structuré.

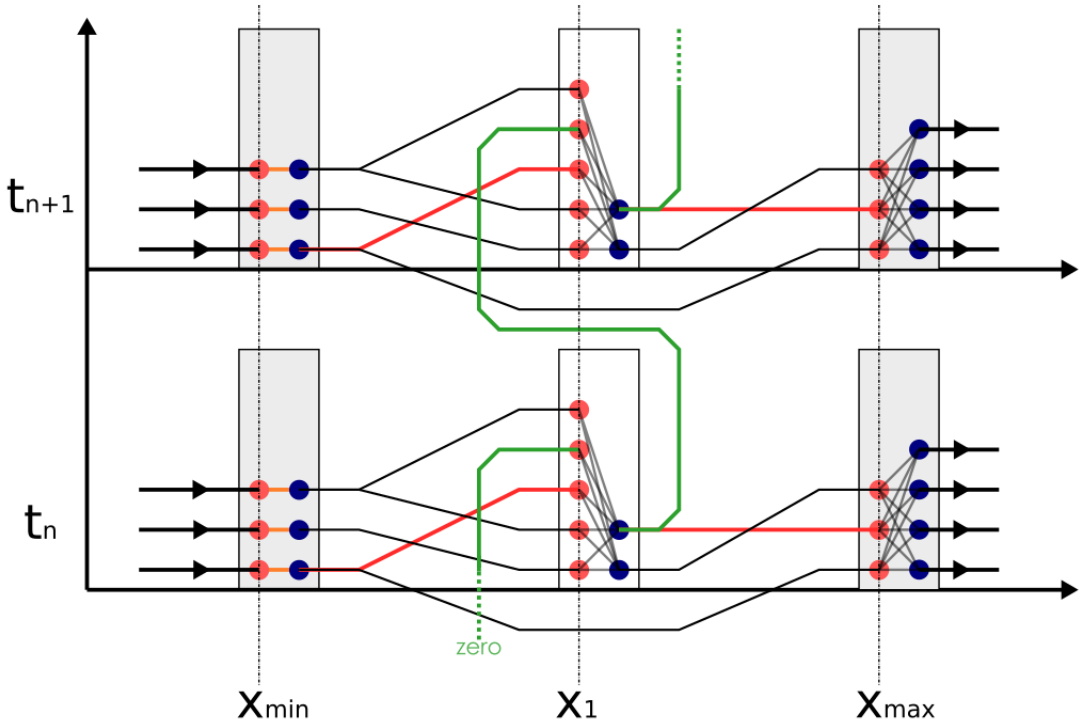


Figure 1: Reseau monocouche à connection aléatoire; Le meme reseau est representé 2 fois à deux intervalle de temps à la suite, les nombres entrée/sorties sont pour l'exemple. De bas en haut : La commucation entre deux temps. De gauche à droite : L'entrée vers la sortie.

Comme les connections entre l'entrée d'une couche X_n et la sortie d'une couche en X_{n+m}, avec $m \geqslant 0$, n'est pas possible pour le calcul de la retropropagation à un temps donnée, on se trouve avec la connection $X_{n+m,t-1} \rightarrow X_{n,t}$. Ce type de connection laisse apparaitre une forme de reseau recurrent pour le calcul du gradient, mais comme nous voulons eviter les structure artificielle comme LSTM qui empeche le *vanishing gradient* des reseau recurrent [Ref LSTM], nous avons consideré cette entrée à "t-1" comme une ***entrée virtuelle***. Cette entrée virtuelle, n'est pas la solution la plus elegante, mais est la plus flexible dans le cas ou les connections sont completement desordonné et reduit le temps de calculs car taille de l'entrainement est linéaire. L'entrée virtuelle consiste au détachement de la sortie du graphes computationnelle à l'algorithme du gradient. On obtient un réseau de "*neurones à propagation avant*" que l'on qualifiera dans le programme de ***pseudo-récurrents*** à cause des entrée virtuelle.

1.2 Construction du graphe

La topologie du reseau presenté precedement peut etre vu comme un graphe orienté acyclique, en effet, les connection entre l'entrée X_n et la sortie X_{n+m} sont vue comme des entrée virtuelle indépendante de la rétropropagation. Les connection intercouche peuvent etre négligé pour la construction initiale du graphe, car elles sont completement connecté (voir code PyTorch *nn.linear*) . Pour les connections entre les couches, on se retrouve face à plusieurs contraintes :

- Quel est le nombre minimale de connection pour que le reseau soit complement connecté ?
- Combien de connection sont attribué par couche ?
- Comment attribuer les connection par couche de neurone ?
- Comment stocker ces informations pour reconstruire le graphe ?

Empiriquement, le nombre total de perceptron dans la couche intermédiaire "N_h" est initialement défini comme la racine entiere du nombre d'agent par génération d'entrainement et ne peut dépasser 32.

Propriétés "Unicité des entrée": Les liens/connections entre les différentes couches de neurones depend des entrée de celle ci. On peut se connecter plusieurs fois à une sortie, mais "une" entrée n'a qu'une "seule et unique" entrée.

À partir de cette propriétés, si on veut que toute les entrée soit non vide, on peut definir le nombre minimal de connection pour notre reseau comme la somme du nombre de perceptron de la couche intermédiaire "N_h" et du nombre de perceptron en entrée "N_i". Ainsi, pour que le graphe soit complet, la relations du nombre minimal de connection "C_{min}" est :

$$C_{min} = N_h + N_i$$

Le nombre de connection total "C_{tot}" est ensuite attribué aléatoirement dans l'interval $[C_{min}, 2C_{min}]$. Puis le nombre de couche intermediaire du reseau "N_L" est attribué aléatoirement dans l'interval $[1, C_{tot}]$. Lorsque que "C_{tot}" et "N_L" sont défini, il reste à définir le nombre de connection et de perceptron que va avoir chaque couche de neurone.

Propriétés "Connection limité" : Il ne peut y avoir moins de "une et unique" connection par couche, mais elle ne peut dépasser l'ecart entre "C_{tot}" et "N_L".

Propriétés "Perceptron limité" : Il ne peut y avoir moins d'"un seule et unique" perceptron par couche, mais elle ne peut dépasser l'ecart entre "N_h" et "N_L".

À partir de ces deux propriétés, on peut definir par récurrance l'attribution aléatoire du nombre de connection et perceptron par couche intermediaire de neurone. Le schéma de l'evolution de la densité de probabilité uniforme à discrétiser est le suivant :

$$f_{C_n}(x) = \begin{cases} \frac{1}{(C_{tot} - C_{n-1} - N_L - n) - 1} & pour\, 1 \leq x \leq (C_{tot} - C_{n-1} - N_L - n) \\ 0 & sinon \end{cases}$$

$$f_{N_n}(x) = \begin{cases} \frac{1}{(N_h - N_{n-1} - N_L - n) - 1} & pour\, 1 \leq x \leq (N_h - N_{n-1} - N_L - n) \\ 0 & sinon \end{cases}$$

Une fois l'attribution du nombre de connection réalisé par couche, il est necessaire de positionner ces couches spatialement pour le calcul de la rétropropagation. Dans notre cas, nous avons limité le nombre de couche à 32, avec une seule et unique couche possible par position spatiale discrete. Ainsi la position des couches intermediaire est atribué aléatoirement et sans remplacement dans l'interval $[1; 31]$, où la position ***zero*** est reservé à l'entrée et ***32*** à la sortie. Les positions spatiales des couches de neurone n'influence pas le calcul de la rétropropagation, ainsi, les connection entre couche sont relative à l'ordre positionnel. On stocke ensuite l'ensemble des connection possible dans une liste. Mais une fois le positionnement des couches réalisé, comment connecter les noeud de facon à ce qu'il y ait un lien entre l'entrée et la sortie et qu'il n'y ait pas que des entrée virtuelle ?

Pour cette question, nous avons considéré 3 cas où l'attribution des connections suit des loi uniforme discrétisé avec des densité de probabilité différente. On distingue les 3 cas suivants :

1. La premiere connection : on se connecte à l'une des sortie de la couche la plus proche derriere, si c'est la premiere, on se connecte à l'une des sortie des neurones d'entree. En effet, il est necessaire qu'il y est au moins un chemin qui mene d'entrée vers la sortie pour la retropropagation, ce cas garantis cette condition. La probabilité est directement défini par $P(X_k) = \begin{cases} 1 & X_k = X_{k+1} - 1 \\ 0 & sinon \end{cases}$ pour ce cas.

2. Tant que toute les connections n'ont pas été attribué : On privilégie les connections vers l'arriere à $2/3$ des probabilité. De cette facon, on limite l'excess d'entrée virtuelle sans que cela ne soit impossible. Cette regle à été défini empiriquement et ne semble pas avoir d'effet sur le processus de selection sur le long termes (non calculé). La probabilité est calculé à partir de la discretisation de la densité de probabilité discrétisé $f_X = \begin{cases} \frac{2}{3} & pour\, a \leq x \leq b \\ \frac{1}{3} & pour\, b \leq x \leq c \\ 0 & sinon \end{cases}$ dans ce cas.

3. Lorsque toute les connection avec les neurones de sortie ont été au moins attribué une fois, alors il n'y a plus de contrainte spatiale au choix des neurones de sortie : La probabilité suit une loi uniforme sur l'ensemble de l'intervalle de definition spatiale.

Au cours de ce processus, les connections des couches au sortie sont listées dans l'ordre d'attribution par couche, ce qui nous donne une liste d'adjacence par couche. Chacune des listes ont par définition la taille du nombre de connection par couche. En representant l'ensemble des listes dans une matrice d'adjacence, on remarque que la diagonale sépare les connections en amonts des connection en aval. La matrice à par définition la structure d'un graph orienté.

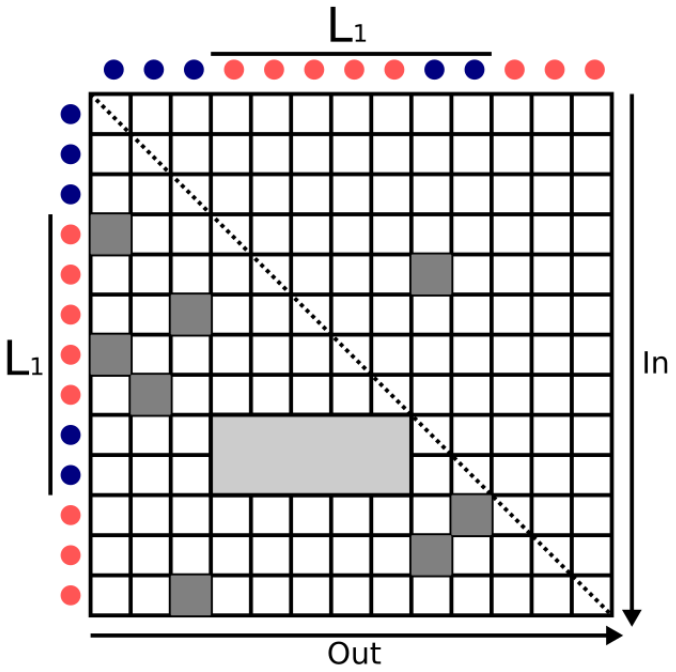


Figure 2: Matrice adjacente des liens du reseau de neurone précedent. En rouge, les noeud de connection (entrée d'une couche) et en bleu, les noeuds de perceptron (sortie de couche). Les case grisé correspondent à une connection de poids constants (1 par défauts).

Cette matrice, n'est pas utiliser dans le processus de restructuration du reseau. La matrice etant creuse, on peut se limiter à un ensemble de liste adjacente par couche de neurone. La representation en liste simplifie la restructuration du reseau et optimise la taille d'utilisation de la mémoire. Cette matrice sera toutefois necessaire pour l'analyse des donnée de graphe, on la reconstruiera à posteriori. Les connection intercouches represenété en gris clair sur la [Fig 2.] ne sont pas stocké en mémoire car ils sont définit par des opérations tensoriel déjà implémenté.

2 Methodes d'apprentissage

Les methodes d'apprentissages standards ont besoin d'un grand nombre de donnée pour que l'ajustement des poids permettent à un modele d'etre prédictif [ref]. Avec l'ere du Big Data, cette approche fut concluante, meme si elle risque de favoriser le surapprentissage d'un reseau en classifiant trop avec les données d'entrainement [ref]. Pour autant, dans le vivant, il n'y a pas besoin d'un tres grand nombre d'experience pour qu'un cerveau apprenne efficassement un probleme [ref]. Dans certain cas, le cerveau est meme pré-cablé chez certaine espece pour repondre à certaine fonction, on peut citer comme exemple la marche des enfants à la naissance [ref] ou des mammifere juvénile proie [ref]. C'est le principe de fonctionnalisation cérébrale qu'on essaye ici de reconstituer en selectionnant des reseau "optimum" à la descente de gradient d'un probleme de decision. A partir du graphe vu precedemant, on peut reconstruire les connections entre les différentes couches du *réseau de neurones à propagation avant*. L'algorithme "forward" de la plupart des bibliothèques logiciel se retrouve restructuré.

Algorithm 1 Forward construction by Adjency List; **X** correspond au Tenseur d'entrée de taille I*Batch. **NET** correspond au information des neurones (index, position, liste adjacente). **Trace** correspond à la mise en mémoire des sorties des couches modulés des liste de neurones **Layers** (PyTorch : *nn.ModuleList*). **h** correspond à la copie détaché de la liste des Tenseurs Trace à t-1.

```
Batch size, Input size ← dim(X)
BATCH_ ← np.arange(len(x))
Trace[-1] ← Layers[-1](X.view(Batch size, Input size,1)).view(Batch size, Input size)
for i, network index ordered by position do :
    tensor ← []
    for j,k in NET[i, -1] :
        if j == 0 :
            tensor += [Trace[-1][BATCH_,None,k]]
        else :
            if (NET[i, 3] >= NET[i, 3]) :
              tensor += [h[j][BATCH_,None,k]]
            else :
              tensor += [Trace[j][BATCH_,None,k]]
    tensor_in ← torch.cat(tensor, dim=1)
    Trace[i] ← Layers[i](tensor_in)
    Trace[-2] ← Layers[-2](tensor_in)
for t in range(len(Trace)):
    h[t][BATCH_] ← Trace[t][BATCH_].detach()
return Trace[i], Trace[-2]
```

Une fois le reseau completement cablé, celui ci est equivalent à un *réseau à propagation avant* avec plusieurs entrée intermédiaire. Ces entrée intermédiaire corresponde comme on l'a vu dans la partie precedante au *entrée virtuelle*. L'objectif qui suit est d'optimiser le calcul de la fonction de cout $J(a,b)$ avec $J(a,b) = \frac{1}{2m}\sum_{i=1}^{m}(f-y)$. Dans notre cas, on distingue deux etapes dans l'optimisation du calcul la fonction de perte, d'abords à "***temps courts***" qui correspond à un apprentissage classique de descente de gradient à chaque cycle de reproduction, et à "***temps long***" où l'on va selectionner à chaque étapes les reseau qui auront le mieux reussi pendant de l'entrainement à temps court. La premiere étapes correspond à une methodes classique en apprentissage par renforcement, le Q-learning, la seconde, plus exploratoire, correspond à l'ajout de mutation dans le graphe du reseau qui va changer les connections entre les différents noeuds.

2.1 Temps courts : Q-Learning

A temps courts on cherche un algorithme de descente de gradient adapté à notre probleme. La plupart du temps, on donne un jeu de donnée d'entrainement adapté à un probleme, comme par exemple *MNIST* pour la reconnaissance des chiffres au format image, ou encore, *ImageNet* pour la detection et classification d'image d'objet. Dans notre cas, on a essaie de reproduire un systeme "vivant" caractérisé par un automate cellulaire [Voir partie suivante]. Ainsi, le système suit une succession d'états et d'action distincts dans le temps et ceci en fonction de probabilités de transitions. L'evolution du systeme correspond à un processus de decision markovien, representé par une chaine de markov. Une chaine de markov est une suite de variable aléatoire (X_n) dans l'espace probabilisé (E, B, P), où pour chaque **n**, sachant X_n, X_{n+1} indépendant de X_k, on a la probabilité de transition (Hypothèse de Markov) :

$$P(X_{n+1} = i_{n+1} | X_1 = i_1, \ldots, X_n = i_n) = P(X_{n+1} = i_{n+1} | X_n = i_n)$$

Dans notre cas, on associe à chaque transition, des action et des récompense associé à l'agent de facon à le guider dans le temps. On peut représenter cela suivant le couple de matrices (T, R), où $\boldsymbol{T}$ correspond à la matrice de transition et $\boldsymbol{R}$, la matrice des récompense. La complexité est que l'on ne connait pas la probabilité de transition.

Le but dans un processus décisionnel markovien est de trouver une bonne « politique » pour le décideur : une fonction $\boldsymbol{\pi}$ qui spécifie l'action $\pi(s)$ que le décideur choisira lorsqu'il sera dans l'état **s**. Une politique décrit les choix des actions à jouer par l'agent dans chaque état. L'agent choisit une politique à l'aide de la fonction de récompense R. Lorsqu'une politique et un critère sont déterminés, deux fonctions centrales peuvent être définies : **V**, la fonction valeurs des etats qui représente le gain engrangé par l'agent s'il démarre à l'état s et **Q**, la fonction de valeur des états-actions qui représente le gain engrangé par l'agent s'il démarre à l'état s et commence par effectuer l'action a. Les expressions de V et Q, sont ainsi déterminer par les relations de récurrence :

$$V_{k+1}(s) = (1-\alpha)V_k(s) + \alpha[r + \gamma V_k(s')]$$
$$Q_{k+1}(s,a) = (1-\alpha)Q_k(s,a) + \alpha[r + \gamma \max_{a'} Q_k(s',a')]$$

Cette derniere correspond à l'*équation de Bellman* et c'est elle qui est utilisé pour calculer la fonction de coût du reseau de neurone lorsqu'on realise une action. La premiere equation ne peut etre utilisé seul car elle nous donne l'efficassité de l'agent pour une action donnée, mais peut utiliser en cas d'utilisation de reseau adversariaux [Ref actor-critic]. Dans notre cas, on realise initalement une suite d'evenement avec une matrice de transition aléatoire (reseau non entrainé), puis on calcul de la fonction de perte entre la prédiction des valeurs Q obtenu au cours de l'experience avant l'action et le resultat de l'equation de Bellman Q apres l'action d'un batch d'entrainement donnée, puis on repete N_{cycle} fois cette etapes avant le prochain cycle de reproduction.

Code PyTorch :

```
old_state, action, new_state, reward, DONE = MEMORY
actor = MODEL(old_state)
pred_q_values_batch = torch.sum(actor.gather(1, action),dim=1).detach()
pred_q_values_next = MODEL(new_state)
target_q_values_batch = reward+(1DONE)*GAMMA*torch.max(pred_q_values_next, 1)[0]
MODEL.zero_grad()
loss = criterion(pred_q_values_batch,target_q_values_batch)
```

L'action "acteur" correspond à la sortie du reseau, et correspond à la probabilité d'action optimale. Normalement, à chaque pas de temps, on choisi la valeurs maximale en sortie du reseau et l'on attribu une probabilité d'action aléatoire, c'est le **dilemne exploration-exploitation**. Ce dilemne permet au reseau de neurone d'explorer des parametre et de ne pas se stabiliser dans une valée non optimale. Mais dans notre cas, nous avons choisi d'attribuer une probabilité d'action à chaque "***pas***" de l'entrainement à partir de la sortie du reseau, qui est equivalent à un dilemne d'exploration-exploitation.

Code Python :

```
DILEMNA = np.squeeze(action_probs.detach().numpy())
p_norm = DILEMNA/DILEMNA.sum()
next_action = np.random.choice(self.IO[1], p=p_norm)
```

2.2 Temps longs : Structure neuronale

A partir des différents modele de réseau entrainée à temps courts, l'objectif qui suit est de selectionner ceux qui on eu le meilleurs score d'entrainement, puis de modifier legerement la structure du graph neuronal. Le calcul des scores est relatif à l'environement d'entrainement et est tres utiliser dans les modele evolutif. L'attribution des points sera plus detaillé dans la partie 3. Le principe est le suivant : Lors du premier entrainement à temps court, on a N_r reseau en parallele et distinct, chacun vont suivre le processus d'entrainement vue dans la sous-partie précedante. Ensuite, les N_b reseau ayant le meilleurs score sont selectionné, N_b reseau garde la meme topologie pour le le cycle suivant, N_m reseau par N_b meilleurs reseau hérite d'une mutation alétoire. Enfin N_c nouveau reseau aléatoire sont introduit dans le processus d'entrainement suivant, ceux ci peuvent hériter de parametre optimisé des reseau precedant, mais pas de la structure du reseau, c'est les reseau "compétiteur". Les nombre d'agent sont definit comme $N_r = (N_b + 1)^2$, de facon à ce que N_r est une racine entiere, $N_m = N_b$, tel que le nombre total de mutation soit $N_{mtot} = N_b^2$ et enfin $N_c = N_b + 1$, de cette facon, on a bien $N_r = N_b + N_{mtot} + N_c$. Dans notre modèle, on distingue 5 types de mutations sur le graphe neuronal :

1. Ajouter une connection : Ne change pas le nombre de couche neural, mais ajoute une connection à l'une d'entre elle. Comme on n'ajoute pas de neurone, la liste des connections possible ne change pas et la nouveau noeud va etre connecté aléatoirement à une neurone pre-existant.

2. Ajouter un neurone à l'une des couches : Comme on ajoute une connection de sortie possible, on renouvelle la liste des connection. Mais vue que le reseau doit etre complet, on rajoute également une connection dans l'une des couches qui va se connecter exclusiement à cette derniere connection de sortie.

3. Ajouter une couche 1*1 : L'ajout d'une couche n'est composé que d'une connection d'entrée et d'une seule et unique sortie, ce qu'y est equivalent à un seul neurone. La position de la couche doit etre différent de celle precedante et compris entre $[1, 31]$. L'entrée de cette nouvelle couche se connecte uniquement vers l'arriere, par contre, il n'est pas necessaire que la sortie soit connecté vers l'avant, le reseau etant complet, seul lui meme est interdit. Si ce neurone se connecte vers l'arriere, il devient equivalent à un générateur d'entrée virtuelle. Comme on ajoute une connection de sortie, la liste de connection et une nouvelle connection est ajouté tout comme l'ajout d'un neurone.

4. Enlever une connection doublon : Enlever une connection quelquonque n'a pas été envisagé dans notre cas, car ils changerait radicalement la structure du reseau. En effet, elle supprimerai possiblement plusieurs connection, ce qui rendrait fortement le reseau imcomplet. Par contre, comme le reseau peut etre connecté plusieurs fois à la meme sortie, il est possible d'enlever une connection doublon. Cela ne changerait pas la liste des connections de sortie possible.

5. Enlever un neurone : Cette opération contient deux contrainte, on ne peut pas supprimer un neurone qui correspond au premier lien d'une couche donnée et que le décallage des connections soit possible. Corrolaire des définitions : on ne peut pas supprimer une couche qui ne contient qu'un seul et unique neurone et de meme pour la connection d'entrée. Tout comme l'ajout de neurone, la liste des connections de sortie possible est mise à jours.

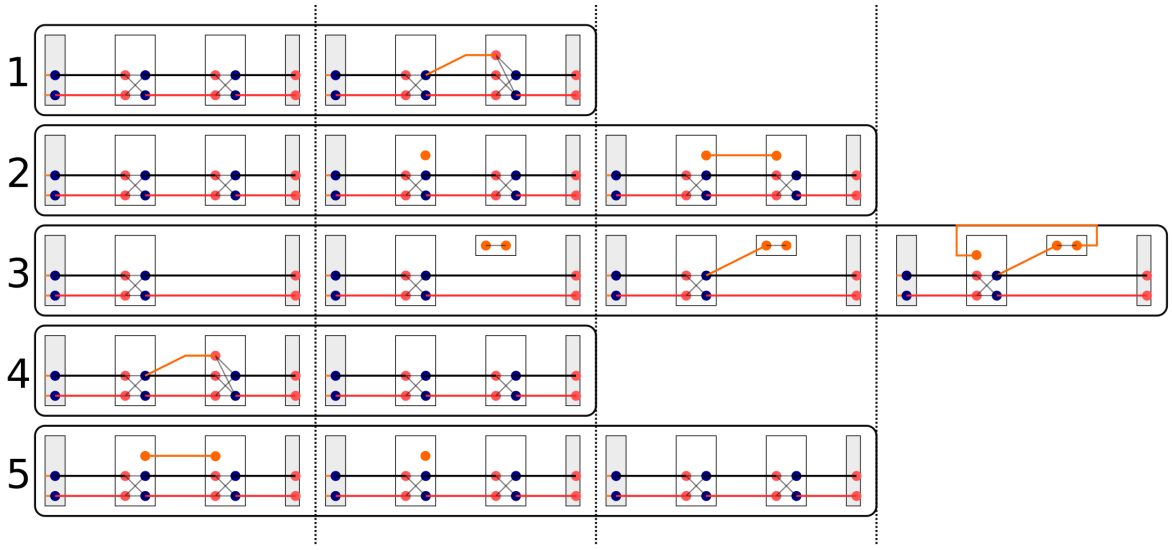


Figure 3: Algorithme visuel des mutations. Respectivement les 5 types de mutations citer en paragraphes. De gauche à droite, l'etats initial vers l'etat final. Les propriétés du graphe sont à titre indicatives, cette representation tres simplifié ne reflete pas l'optimum d'une fonction. Les opérations $1 \leftrightarrow 4$ et $2 \leftrightarrow 5$ sont symetriques, l'operation 3, n'a pas de symetrique stable simple.

Dans notre modele, on conserve au minimum un neurone par couche et toujours le premier, et on ne supprime pas les couches de neurone. Ce choix plus simple permet de conserver la symetrie des opérations $1 \leftrightarrow 4$ et $2 \leftrightarrow 5$ [voir Fig 3], mais aussi de maintenir une structure vestigiale du reseau. En effet, au cours de l'evolution certaine structure sont maintenu, mais ne sont pas les choix optimums por remplir certaine fonction. Par exemple, le chemin qu'emprunte les vaiseau sanguin entre la tete et le coeur chez les mammifere emprunte un chemin non optimal, pourtant il est maintenu car serait trop couteux evolutivement pour changer de trajectoire [ref necessaire]. La vallée de stabilité evolutive lié à ce choix criticable est plus abordé en discussion.

3 Regles du jeu

Dans le cadre des processus de decision markovien, les donnée sont généré au cours de l'entrainement, c'est l'**environement** d'entrainement. Cet environement contient des regles et s'apparente à un jeu, la biblioteques la plus utilisé en python est *OpenGym* et sa structure à servi d'exemple pour realiser l'environement de notre probleme. Dans notre cas, on souhaite reproduire certaine stratégie de survie que l'on observe dans le vivant, en particulier, les stratégies de prédations et de fuites d'une proie. On a choisi pour cela le **jeu du chat et de la souris**, ou *Tag game* en anglais, en effet, on alterne entre proie lorsqu'on est souris, et prédateurs lorsqu'on est chat, ce qui pousse le modele à s'adapter à deux configurations différente. Les étapes d'une partie sont les suivantes :

1. Initialement, l'agent est une proie "la souris". Il doit eviter de se faire attraper par le prédateur, "le chat". Le prédateur n'est pas un agent "intelligent", celui-ci calcul uniquement le déplacement optimal qui minimise la distance euclidienne entre le chat et la souris pour l'etat suivant.

2. Lorsque l'agent se fait "attraper" par le prédateur, il en devient lui meme un et les role s'inverse. L'agent doit maintenant attraper la proie, la nouvelle souris. La souris cette fois n'est pas un agent "intelligent", celui ci calcul uniquement le déplacement optimal qui maximise la distance euclidienne entre le chat et la souris pour l'etat suivant.

3. On revient à l'etapes 1 si l'agent attrape la souris, la partie s'arrete apres N_g pas de temps.

On a dans ce cas, un agent qui interagit avec un environement de jeu. L'agent contient les informations de **vue** et d'**action** qui va transmettre à l'environment. L'environement contient les information de la position de l'agent et son adversaire, ainsi que les regles de jeu et comptage de points. L'environement peut etre vue comme une grille à limite périodique (CLP) de dimension $N \times M$ où les agents et adversaire sont des automates cellulaires avec des fonctionnement différents. L'agent dispose de 9 entrée "états", 3 sortie "action" et un reseau de neurone entre les deux, où les propriété positionnel entrée/sortie sur la grille sont généré aléatoirement en debut d'experience, puis maintenu à la descendance. Ainsi, l'agent lit 9 case sur une grille 5×5 centré sur la position de l'agent, soit 36% de son environement local, et l'agent se déplace d'une seul case parmit 3 mouvement d'une grille 3×3 centré sur la position de l'agent, soit 33,3% de son environement local. Par contre, l'adversaire voit uniquement la position de l'agent et de lui-meme, mais n'est qu'une fonction mathématique se deplacant suivant 4 trajectoires, haut $(0, 1)$, bas $(0, -1)$, gauche $(-1, 0)$ et droite $(1, 0)$, minimisant ou maximisant la distance entre elle et l'agent suivant son état "chat" ou "souris". La informations positionnel sont discretisé (grille), le calcul du déplacement de l'adversaire revient à :

$$\begin{cases} d_2 = \sqrt{(v_{adv} - v_{agent})^2} \\ m_{vt} = \begin{cases} m_{vt} - min(d_2) & if\ IT \\ m_{vt} + max(d_2) & else \end{cases} \end{cases}$$

Enfin, l'environement attribue des points à chaque étapes du jeu, l'ensemble des points donne le score de l'agent. L'attribution des points à l'agent est un parametre important en apprentissage par renforcement, un desequilibre des points peut rendre soit un agent tres "agressif" ou encore à l'opposé "passif" [ref necessaire et importante]. Pour cela, les points ont été reflechi en fonction de la taille de grille de jeu, pour une grille de jeu 16×16, la moyenne des déplacement pour atteindre le centre quelque soit la position est d'environs 8 déplacement. Dans cette configuration, pour un calcul equilibré, les comptages des points sont les suivants :

- -1 par "pas" de temps où l'agent est un prédateur "chat"
- +10 si l'agent attrape la proie "souris" et devient la proie à son tour.
- +1 par "pas" de temps où l'agent est une proie "souris".
- -10 si l'agent est attrapé par le prédateur "chat" et devient le prédateur à son tour.^

Corrolaire : L'agent restant le plus longtemps sous l'état proie (souris) et le moins longtemps sous l'état prédateur (chats) à le plus grand score de jeu.

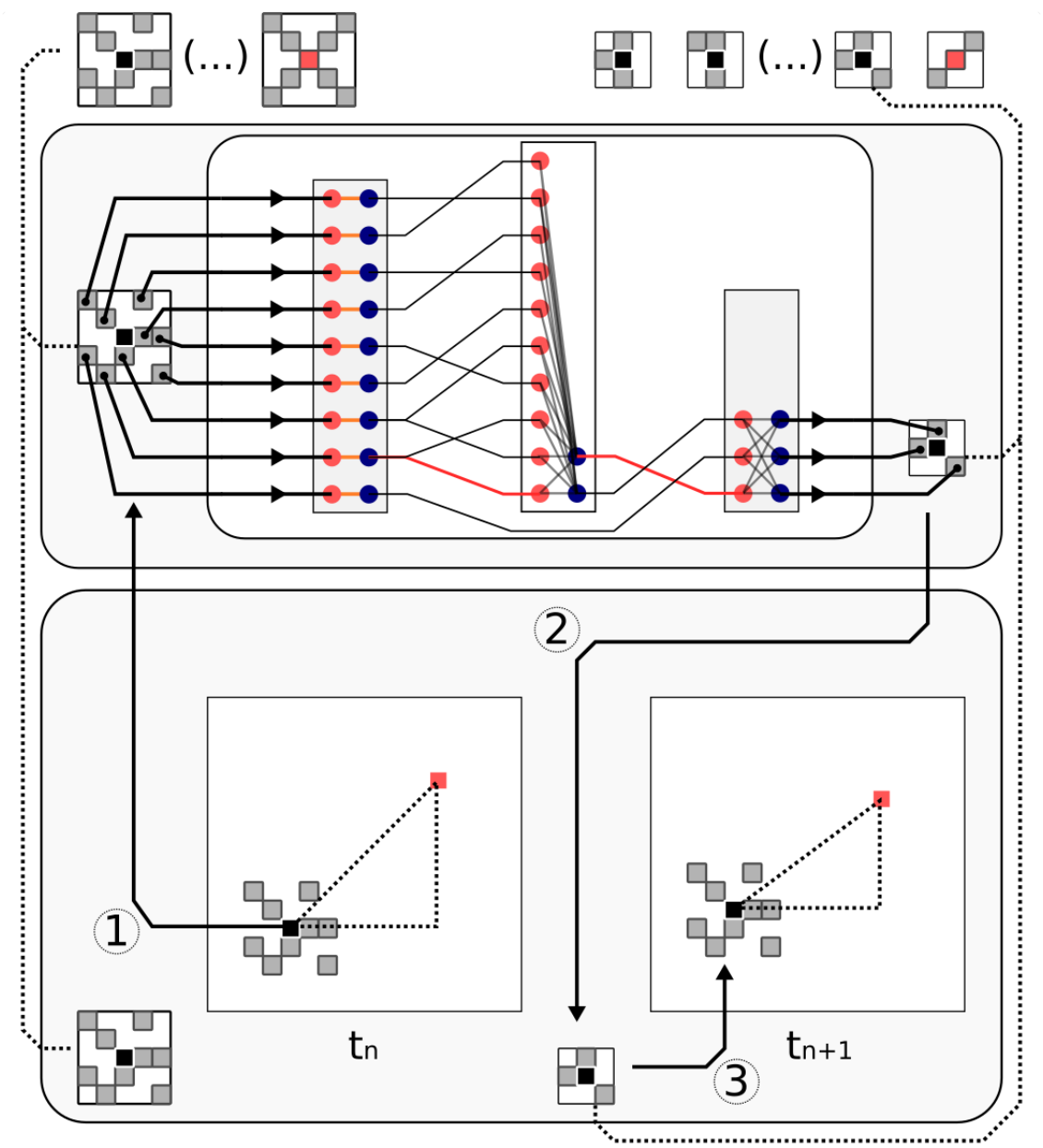


Figure 4: Les interactions entre un agent et un environement de jeu. En haut, la génération alétoire des positions relative possible des entree et sortie. Suivi des interactions entre l'agent en haut et l'environement de jeu en bas. 1 : envoi de l'environement des informations d'entrée (pixel) à l'agent; 2 : envoi de l'information de sortie de l'agent (mouvement) à l'environement; 3 : mise à jour de la carte de jeu, l'adversaire effectue son déplacement à ce moment.

Les case des positions relative d'entrée et de sortie sont défini aléatoirement, mais comme le pourcentage des case utilisé sur la grille est au alentour de 35%, les combinaisons possibles sont de l'ordre factoriel. Pour la sortie, on a 3 positions parmi un total de 9 cases, ce qui donne $\binom{9}{3} = 84$ et comme le probleme est symetrique au 4 mouvement de l'adversaire (exemple : $(1,2,3) \iff (7,8,9)$), le nombre de combinaison est de l'ordre du nombre du nombre d'agent par génération. Par contre, pour l'entrée, on a 9 positions parmi un total de 25 cases, ce qui donne $\binom{25}{9} = 2042975$ ce qui pose un probleme de convergence de la vision optimal. Mais, comme les quadrant d'observation sont equivalent, on peut considerer uniquement les 9 cases par quadrant, ce qui nous donne

par ratio de 36% $\begin{pmatrix} 9 \\ 3 \end{pmatrix} = 84$, cela est presque un ordre au dessus du nombre de génération et ne pose moins probleme de stabilité. Les cases ayant obtenu les meilleurs scores sont classé dans l'ordre et l'on attribu la somme des points par case sur les grilles d'observation et d'action. La normalisation de la grille donne la probabilité de choisir l'une des cases pour les etapes suivantes des "challenger" et sera plus abordé dans la partie résultats. Pour les action, on observe quelque cas typique : 3-cyclique (Exemple : $(1, 6, 8) \Rightarrow (\nwarrow, \downarrow, \rightarrow)$), 4-cyclique (Exemple : $(1, 3, 8) \Rightarrow (\nwarrow, \downarrow, \downarrow, \nearrow)$), semi-2-cyclique (Exemple : $(2, 6, 8)$), asymétrique (Exemple : $(1, 2, 3)$) et statique (Exemple : $(1, 5, 8)$). Les positions des entrée et des sorties sont donc aussi selectionné dans le processus evolutif des agents.

Résultats

Pour générer l'ensemble des données experiementale, il est necessaire d'avoir plusieurs agents par génération pour la convergence de la densité de probabilité des positions d'entre/sortie et plusieurs génération pour verifier s'il y a une convergence des structures neuronales. Pour que le nombre d'agent par génération soit de l'ordre du nombre de combinaison d'entree/sortie, nous avons choisi 25 agents par cycle de reproduction. 25 est le carré de 5, ce qui compatible avec l'algorithme de mutation de la structure neuronale. Ce qui donne, 4 meilleurs agents par cycle, ceux ci sont maintenu au prochain cycle et chacun recoit 4 mutation, ce qui fait 20 agents ayant des propriétés analgogue à la génération precedante, les 5 restants sont des nouveaux agents avec des structures neuronale aléatoire, mais avec informations d'entrée/sortie hérité de la densité de probabilité des evement precedant. Les grilles de jeu sont de 16×16, ce qui permet à l'agent de pas voir tout le plan de jeu, mais ne pas que l'adversaire soit trop loin lorsque l'agent est un prédateur. L'avantage de cette configuration est que l'on peut representer l'ensemble des environement à un temps donnée par une grille $(5 * 16) \times (5 * 16)$, ce qui facilité la visualisation et l'interpretation des resultats. Il est possible aussi de representer cette grille par décomposition de deux nombre premier si le nombre d'agents par cycle n'est pas le carré d'un nombre. Cette methodes correpond à verification parité et recurrance des nombre impaire jusqu'a la racine entiere supérieur du module du nombre d'agent par génération : $n\% \begin{cases} 2 \\ impair(\sqrt{n}) \end{cases}$. Cette derniere n'a pas été utilisé ici, mais est inclu dans le programme.

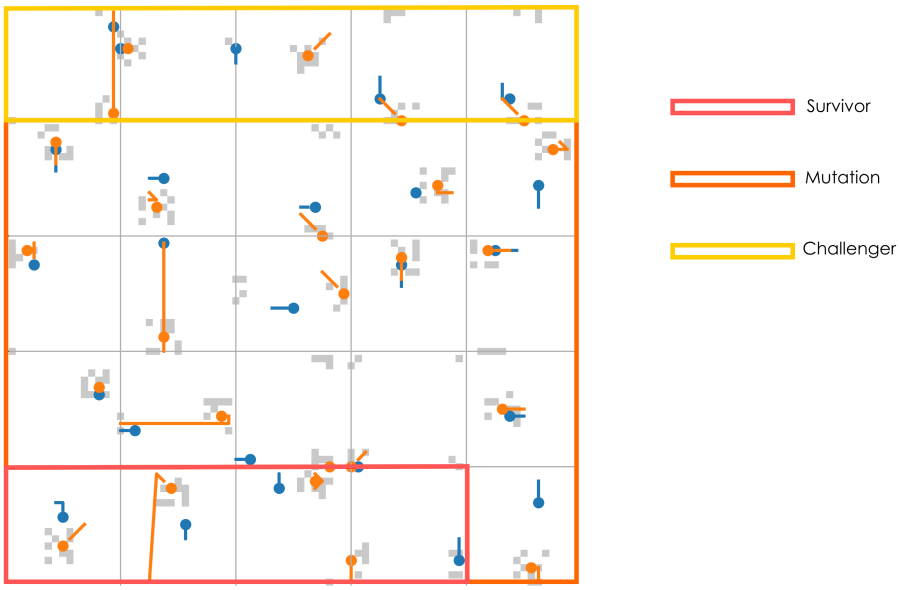


Figure 5: Representation d'une donnée experimentale à un instant t. En encadré, les différentes catégories d'agent (survivant, mutation et compétiteur). En points bleu, les adversaires et en orange, l'agent intelligent. Les cases grisé corresponde à la vue de l'agent à un instant t.

Le nombre de génération n'est pas clairement défini, mais peut, tout comme le nombre d'agent par génération,

etre justifié par la loi des grands nombres. On va donc verifier, pour 25 agents par génération, combien, il faudrait pour que la densité moyenne des entrée et des sortie converge. Soit $Ng = 25$, le nombre de (...)

... Le choix a été motivé par le calcul de la densité moyenne I/O car le choix des case est de 9 sur 25 en entrée. Preuve statistique de la taille minimal echantillon :

$$n = t^2 \times p \times (1-p)/m^2$$

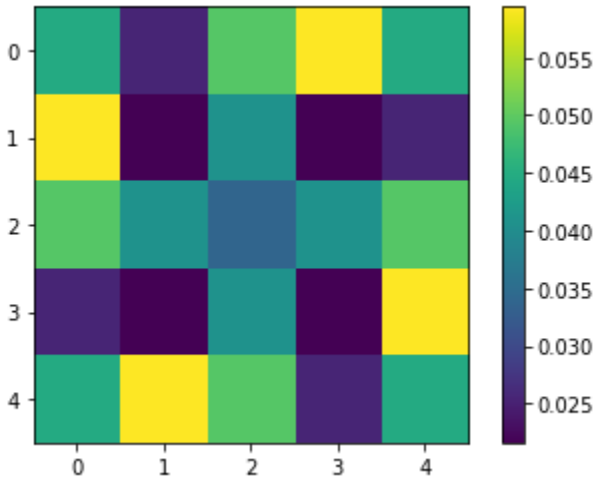
Je comprend pas d'ou ca sort : loi des grand nombre ? (inegalité de BT ?) Je preferai une demonstration de l'intervalle de confiance plutot qu'une formule toute faite...

4 Densité I/O

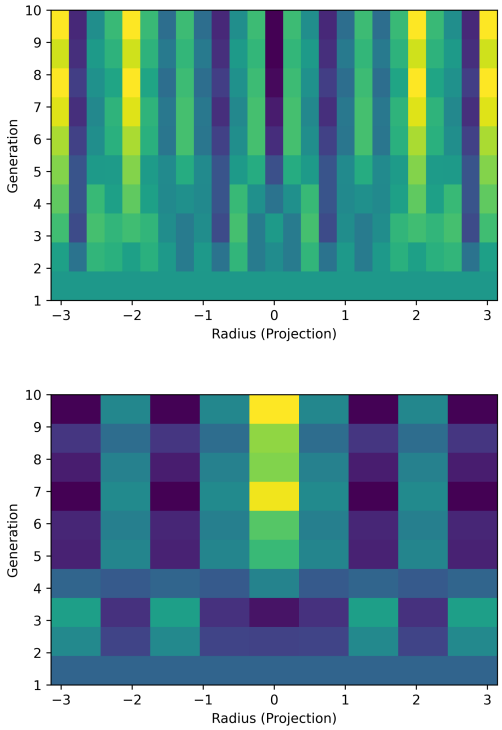
Au cours des cycles de reproduction, on met à jour la densité de probabilité de succés des I/O ayant le plus reussi pour le evenement aléatoire suivant (génération spontanée à chaque cycle). La convergence de la densité de probabilité par une variable aléatoire est defini par le theoreme de Transfert, qui revient dans notre cas à un algorithme de monte-carlo, tel que :

$$G = E[g(X)] = \int g(x) f_X(x) dx$$

On obtient pour l'entrée et la sortie deux densité différente, pour l'entrée ... et la sortie ... . Discutter du caractere cyclique ou non de la sortie, et des emplacement strategique de l'entrée ! Utilisation d'une matrice de rotation $\pi/2$ car symetrie de l'adversaire suivant 4 direction



Pourtant on constate JE NE SAIS PAS ENCORE... ce qui nous donne la courbe d'ecart entre T+1 et T pour le rayon de convergence de la serie I/O (je ne sais pas si on peut parler de serie?). La serie converge en "N" étapes.

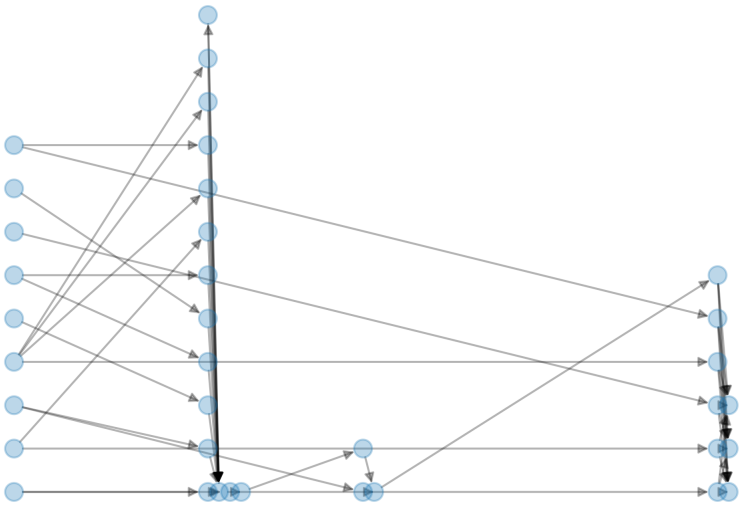


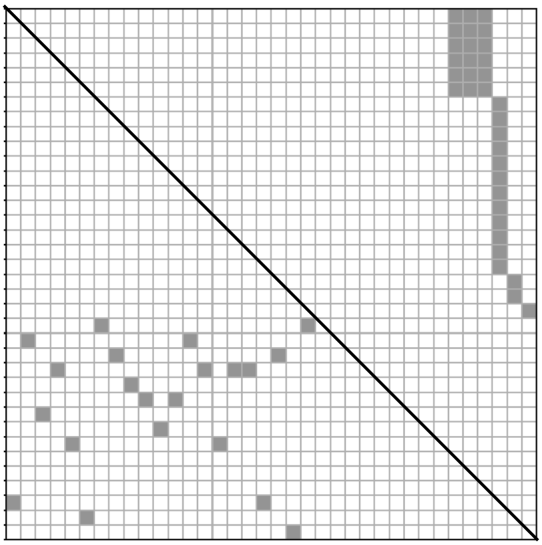
Comme les I/O sont stable (ou pas, je ne sais pas), on utilisera uniquement la densité finale pour les nouveau evenement aléatoire (ou pas). De toute facon, ce parametre(/variable) devient un invariant assez vite (comme on peut le voir dans la courbe evolution ecart : enfin c'est ce qu'a l'air de donner les images préliminaires). VOIR AUSSI FFT ?

5 Connectivité des reseaux

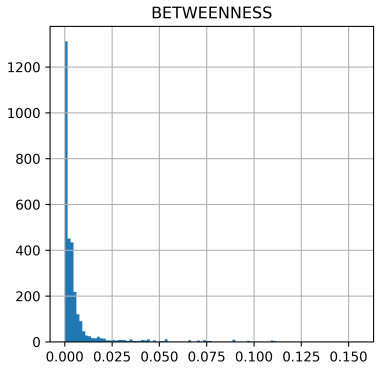
Etudier quel "type" de structure est plus avantageuse pour ce probleme

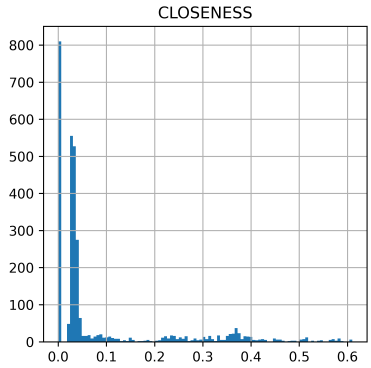
D'abords type de graphes à rapeller (dirigé évidement), ensuite densité du graph, puis voisinage et degrée.

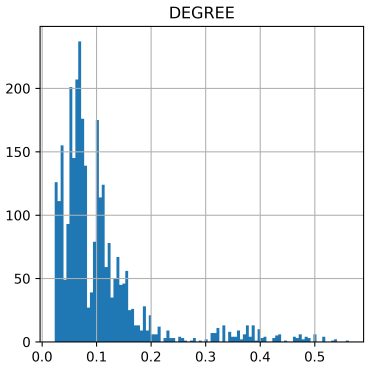




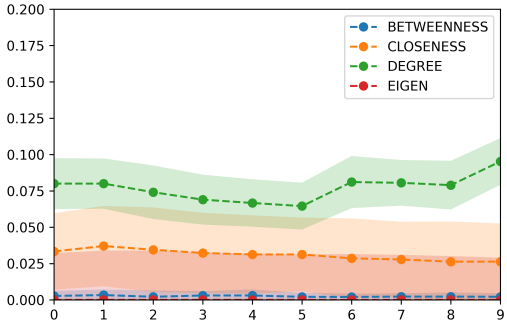
Trouver un moyen de representer le graph “moyen” (et les motifs ?). Les longueurs (si c’est graphe géométrique), closeness, betweeness, degré, orthogonalité (si graphe géométrique)







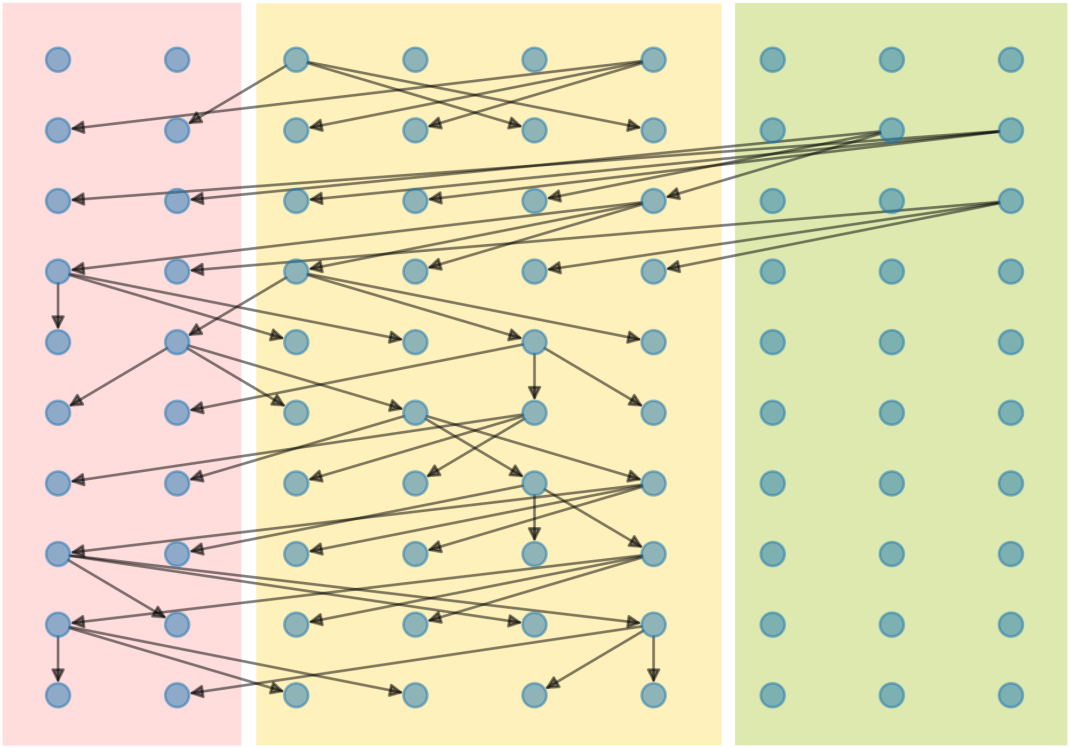
Convergence de la serie de graphe par l’evolution ?



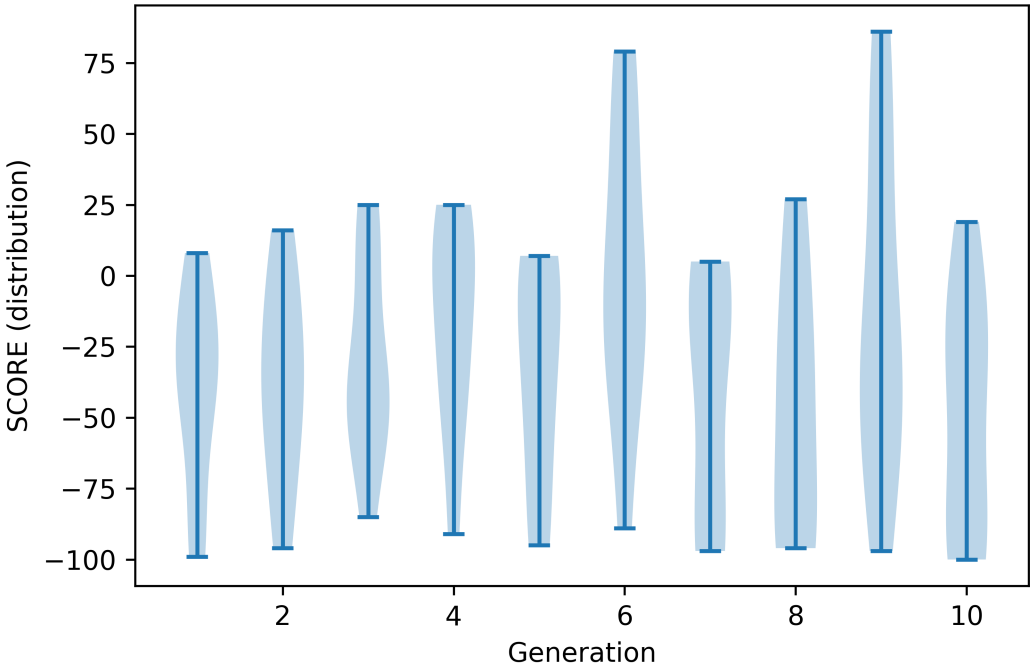
On n’observe pas de stabilité, mais les graphes les plus asymetrique marche le mieux à court terme, mais pas à long terme (l’entrainement q-learning marche moins bien ?) –> oui enfin, c’est par construction, et puis je ne sais pas comment calculer l’ecart type entre deux graphes ??

6 Efficacité selective

Conclure un peu les deux donnée precedante sur la convergence des I/O et de la structure du reseau.



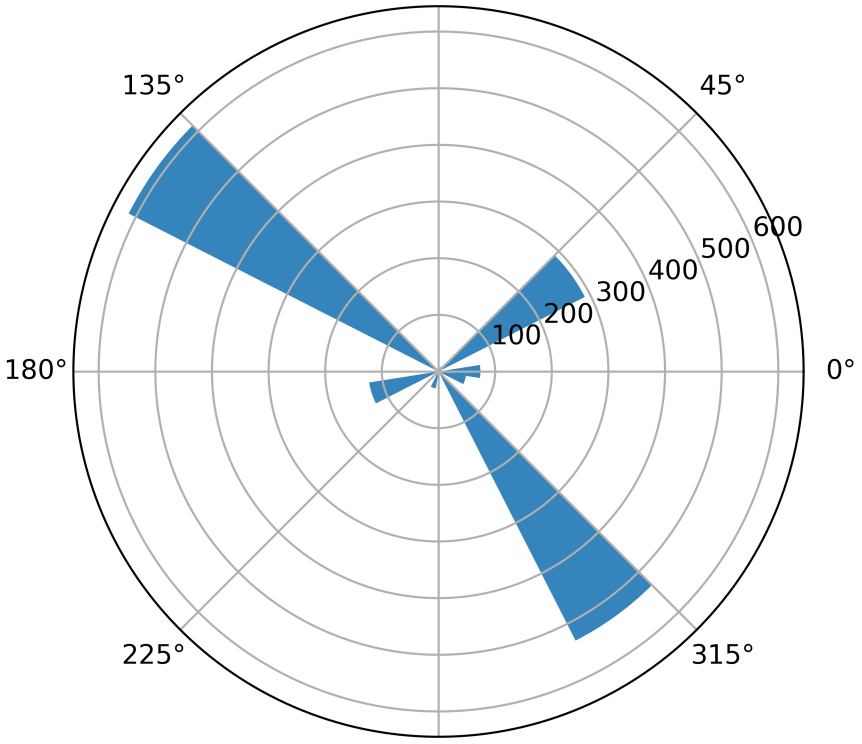
Idée ici est d'appronfondir la vitesse d'apprentissage, un ce qu'un reseau tres adapté à un probleme apprend vite au debut, mais atteint un plafond moins optimale qu'un reseau moins adapté au debut, mais à la suite devient plus performant ?



Temps pour que le reseau soit adapté, heritabilité, comparaison entre aléatoire totale et avec choix de densité

I/O.

Mersure de la dispersion des agent :



On oberse quel type de tracjectoire domine suivant condition limite périodique ou non.

Discussion

Paragraphe critique 1 : case carré ! pourquoi pas un cercle ? car carré probleme de symetrie
Paragraphe critique 2 : reseau vestigiale n'est peut etre pas optimal et empeche d'aller vers une vallé plus optimum (voir pb en bio-evo –> illustrtion vallé d'optimalité)
Paragraphe critique 3 : Artefact de trajectoire, le reseau est avantagé si trajectoire uniquement en diagonal (+ CLP)
Paragraphe ouverture 1 : Optimisation de la structure du reseau en meme temps que l'apprentissage. (citer les autre methode de "self-learning", NEAT, GNN, etc, mais ici dans le plus elementaire)
Paragraphe ouverture 2 : 1 seul enviroement partagé (coop vs compet)
Paragraphe ouverture 3 : Utilisation hors contexte sur base MNIST : comparaison efficassité reseau simple et reseau structuré.

IMAGE BATCH_SIZE comparaison et reseau classique (temps d'adaptation par reseau) (en fonction des cycle d'entrainment) –> (comparé entre cycle d'herédité et cycle d'entrainement)

References

[1] Stuart Bartlett and Michael L. Wong. Defining lyfe in the universe: From three privileged functions to four pillars. 10(4):42. Number: 4 Publisher: Multidisciplinary Digital Publishing Institute.

[2] John Tyler Bonner. The origins of multicellularity. 1(1):27–36.

[3] Roger Buick. When did oxygenic photosynthesis evolve? 363(1504):2731–2743. Publisher: Royal Society.

[4] S. Blair Hedges, Jaime E. Blair, Maria L. Venturi, and Jason L. Shoe. A molecular timescale of eukaryote evolution and the rise of complex multicellular life. 4(1):2.

[5] J. Huxley. Evolution. the modern synthesis. Publisher: London: George Alien & Unwin Ltd.

[6] Andrew Knoll, Malcolm Walter, Guy Narbonne, and Nicholas Christie Blick. The ediacaran period: a new addition to the geologic time scale. 39(1):13–30. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1080/00241160500409223.

[7] S. J. Mojzsis, G. Arrhenius, K. D. McKeegan, T. M. Harrison, A. P. Nutman, and C. R. L. Friend. Evidence for life on earth before 3,800 million years ago. 384(6604):55–59. Number: 6604 Publisher: Nature Publishing Group.