

Evolutionary 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 créés. 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 efficace 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 difficile de definir ce qui est vraiment vivant et cette question est toujours débattu. 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'évolution darwinienne est un principe fondamentale dans l'étude des systeme complexe disposant d'heredité, de variation et de selection. On ne le retrouve pas seulement en biologie, mais on peut l'observer 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'homéostasie et d'apprentissage [1]. Ce qui diminue localement l'entropie en favorisant l'auto-organisation. Mais ces deux definitions se restreignent à un systeme "chimique", est ce qu'un robot dotée d'une intelligence, voient 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 environnement 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 à l'origine des objets mathématique permettant de resoudre des probleme de decidabilité mathématique [Von Newman]. On en retrouva ensuite des utilisation experimentale 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 renforcement. Les propriété de

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

STATISTIQUE : Les réseaux neuronaux ont été construits à partir du modèle de neurone biologique, le neurone formel [McCulloch & Pitts], où l'on constate qu'un neurone est une fonction "affine" avec une fonction d'activation. Aujourd'hui il s'est rapproché des méthodes statistiques non inductive (descriptive : on ne sait pas la loi de probabilité) pour résoudre des problèmes de régression, classification et partitionnement. Les principaux outils d'analyse de grosse donnée en statistiques sont, pour ne citer que les plus connus : La PCA, analyse de composante principale, qui permet de projeter l'ensemble des points sur les axes de régression multilinéaire des variables corrélées entre elles, ce qui réduit la dimension [Karl Pearson], Les SVM, séparateurs à vaste marge, généralisant les classificateurs linéaires 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 [Steinhaus, Lloyd]. Tous ces outils encore utilisés aujourd'hui, sont peu à peu remplacés par des réseaux de neurones vers 2000s après le succès de l'apprentissage profond en compétition.

APPRENTISSAGE : La question de l'apprentissage a été étudiée au cours des années 1950, où il a pu émerger les règles de Hebb [Hebb 1949] qui modifient simplement les valeurs des coefficients synaptiques d'un réseau simple couche, il eut un déclin en fin 1970 par l'impossibilité de résoudre le problème non linéaire XOR ou connexité [Minsky 1969]. Mais ce problème fut résolu par le perceptron multicouche, mais cette fois la modification des poids était plus compliquée. C'est le système de rétropropagation du gradient [1984] qu'on retrouve aussi dans le cerveau [Stuart 1997] et l'algorithme du gradient, par minimisation itérative des erreurs, qui permet de résoudre ce problème. En parallèle, inspiré par les neurones moteurs de la vue, on développa le réseau de neurone convolutif qui réduit fortement le calcul des poids en entrée du réseau, car n'est pas entièrement connecté [Lecun, mais pas que]. Le souci de ce genre de réseau est qu'il faut un grand nombre de données pour adapter les poids, pour résoudre un problème de décision markovien, il y eut l'apparition du Q-learning convergent par l'apprentissage des récompenses [Watkins 1989 Dayan 1992] qui ont encore été améliorés par des réseaux adversariaux "actor-critic" à l'image des réseaux génératifs, il permet d'accélérer la convergence des poids [Konda 2003].

FONCTIONNALISATION : Néanmoins, ces réseaux optimisent l'apprentissage du réseau mais ne cherchent pas à maximiser la structure des couches du réseau. Seuls les réseaux de neurones récurrents ont été le plus traités car il y avait des problèmes de "vanishing gradient", surtout pour le traitement du texte. Une autre catégorie de réseau est apparue dans la fin des années 1990, les réseaux NEAT, où les poids et les connexions sont "appris" par un processus de sélection. Hors le cerveau n'a pas des poids qui ont été sélectionnés par l'évolution, mais ce qui est le cas la structure [Ref]. L'idée est donc de s'inspirer des données en neurosciences récentes, où il est montré le principe de fonctionnalisation cérébrale que le cerveau est pré-cablé pour voir certains dangers 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. À cette image, on a donc développé un algorithme où les couches interconnectées de manière aléatoire de façon à fonctionnaliser le problème "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 façon à utiliser efficacement les bibliothèques logicielles : PyTorch pour les réseaux de neurones, Keras-Tensorflow pour la base de données MNIST, Numpy et Scipy pour les calculs scientifiques, Pandas pour l'analyse et le stockage des données, Networkx pour la représentation et l'analyse des graphes et enfin Matplotlib pour la visualisation et l'animation des données. On distingue 6 fichiers d'expérience, 2 fichiers d'analyse de données (DATA_ANALYSIS et EXTRA_FUNCTION) et un fichier de débogage modulaire. Dans les fichiers "expérience" on retrouve au plus bas niveau GRAPH_EAT encapsulant GRAPH_GEN, générant la liste des connexions pour chaque couche de neurone, ainsi que pRNN, générant la structure du réseau en fonction de la liste d'adjacence. À plus haut niveau, on retrouve AGENT qui contient les propriétés d'entrée/sortie de l'agent et son algorithme d'apprentissage et de mémorisation, ainsi que TAG-ENV contenant cette fois les règles du jeu "chat-souris". Enfin on a le MAIN qui va lister l'ensemble des agents et environnements associés pour lancer les expériences d'évolution. On retrouve aussi un fichier LOG qui permet de stocker au format *csv* les expériences.

1 Réseau de neurone évolutif

Le perceptron est un algorithme d'apprentissage fonctionnant comme une fonction de seuillage et décrivant le neurone formel. Une fonction de seuil est un classifieur linéaire qui a pour entrée un vecteur à valeurs réelles (virgule flottante) et une valeur $f(z)$ en sortie, qui peut être suivant la fonction d'activation : binaire (Heaviside), réel (Sigmoid) ou entre les deux (ReLU). Le perceptron se représente par le produit scalaire entre le vecteur d'entrée et le vecteur poids des neurones, soit l'application multilinéaire suivante :

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

Ce qui s'écrit dans la plupart des bibliothèques logicielles comme la séquence 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 perceptron est un classifieur linéaire, il converge uniquement si l'ensemble des données d'entrée sont linéairement séparables (droite). Mettre à jour les poids “**w**” correspond à l'apprentissage du perceptron, la *regle delta* [Russell Ingrid 2002] est la plus simple et consiste à comparer linéairement la valeur obtenue par la sortie du réseau “**z**” et la valeur attendue par le réseau de neurones “**d**”. Il existe d'autres méthodes plus sophistiquées ou limitant les problèmes d'annulation tout aussi simple (Least Mean Square), mais ici présenté, la règle delta se présente sous la forme :

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

Dans la plupart des bibliothèques logicielles, on précise le type de calcul de perte (loss ou criterion) avant la boucle d'apprentissage, ainsi que l'algorithme de calcul de la descente de gradient (optimizer). Les algorithmes les plus utilisés sont SGD “Stochastic Gradient Descent” plutôt adapté à la classification et “Adam” plutôt adapté au problème de régression [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 réseau de neurone correspond à une interconnexion entre couche de neurone contenant “**n**” perceptrons. Une unique couche de “**n**” perceptron correspond au réseau le plus simple et permet de résoudre des problèmes linéairement séparables de type hyperplan. Comme toutes les entrées “**x**” sont interconnectées au “**n**” neurone, les vecteurs poids “**w**” forment une matrice “**m*n**” avec “**m**” le nombre d'entrée au réseau. La sortie s'exprime dans ce cas :

$$\vec{y} = \begin{bmatrix} w_{0,0} & & \\ & \ddots & \\ & & w_{m,n} \end{bmatrix} \cdot \begin{bmatrix} x_0 & \dots & x_m \end{bmatrix} + \begin{bmatrix} b_0 \\ \vdots \\ b_n \end{bmatrix}$$

$\vec{s} = f(\vec{y})$

Pour que l'algorithme mette à jour efficacement les poids du réseau, on utilise un certain lot de données d'apprentissage, qu'on appelle “batch”, soit il contient l'ensemble des données de l'expérience, soit il est réparti en “mini-batch” pour éviter la sur-utilisation de la mémoire et augmenter la vitesse de convergence de la vallée de stabilité, la taille du lot ne doit pas être trop petite non plus car augmenter le bruit [ref mini-batch]. Dans ce cas, le vecteur d'entrée devient une matrice “**m*b**”, et la matrice des poids devient un tenseur d'ordre 3 “**n*m*b**”. La plupart des bibliothèques logicielles s'adapte automatiquement à la taille du batch, et n'a pas besoin d'être 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 être employé et ne sont pas forcément les plus adaptés pour les problèmes d’images [ref]. En effet, une image est une donnée 2D dont les données spatiales sont localement liées, mais les informations ne sont pas interconnectées, c’est pour cela qu’il existe les réseaux de convolution. Ces réseaux ont l’avantage d’être interconnectés mais par fenêtre “spatiale” ce qui réduit fortement le nombre de calculs pour l’actualisation des poids. Elle est composée de “N” perceptrons, mais ne représente plus comme un produit tensoriel, mais comme un produit de convolution où les entrées sont moyennées pondérément par le glissement “C” d’une fenêtre 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 considérer qu’il y a qu’une seule entrée par neurone, pour cela, le nombre d’entrée est égal au nombre de sortie du canal de la couche. Ensuite, la taille de la fenêtre est de “1” et le groupement de connexion doit être égal au nombre de neurones, qui est un nombre divisible de l’entrée et de la sortie par lui-même. Ce type de paramétrage est équivalent à N neurones indépendants à 1 seule connexion chacune. Contrairement au réseau linéaire, la taille du batch doit être spécifiée 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 cerveau “modèle” pour construire des algorithmes neuronaux a toujours été présent dans le développement de nouvelles méthodes en *Intelligence Artificielle*. Dans ce projet, nous nous sommes inspirés de la spécialisation fonctionnelle 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 spécialisation fonctionnelle du cerveau est observée empiriquement par plusieurs troubles neuropsychologiques, comme par exemple avec le syndrome de Capgras, où une défaillance d’une région entraîne la perte de reconnaissance des visages [Thomas Antérion, 2008].

Dans notre cas, nous avons simplifié le plus possible un cerveau, où l’on a en entrée des neurones de “vision”, suivis de couches séquentielles “fonctionnelles” et en sortie, des neurones “moteurs”. On distingue des neurones stables, les neurones d’entrée et de sortie qui ne changeront pas pendant le processus évolutif. Les neurones de “visions” sont à l’image des cônes de la rétine c’est-à-dire des neurones connectés à des batonnets de vue unique [Dale Purves, G-J Augustine, 2005] qu’on représente dans notre cas comme une couche de convolution “simple sparse” : une seule entrée par neurone. Les neurones “moteurs” quant à eux sont analogues aux motoneurones qui activent la contraction musculaire et donc le mouvement des organismes [Fitzpatrick, D. (2001) The Primary Motor Cortex], on les représente dans notre cas comme une couche linéaire de neurones où les sorties seront liées à l’action du réseau de neurone complet.

La vraie particularité du projet est que les couches intermédiaires “fonctionnalisées” ne sont pas construites manuellement, mais adaptées par un processus évolutif de sélection. Ainsi, l’objectif est de faire émerger une structure intermédiaire fonctionnalisée à notre problème, ici le “jeu du chat et la souris”. Les connexions intermédiaires sont initialement aléatoires avec au minimum une connexion entre la couche “n+1” et “n”, pour avoir toujours un lien entre l’entrée et la sortie et ne pas avoir de problème lors des calculs de poids des neurones. Au cours de l’apprentissage, on s’attend à ce que le réseau soit de plus en plus structuré.

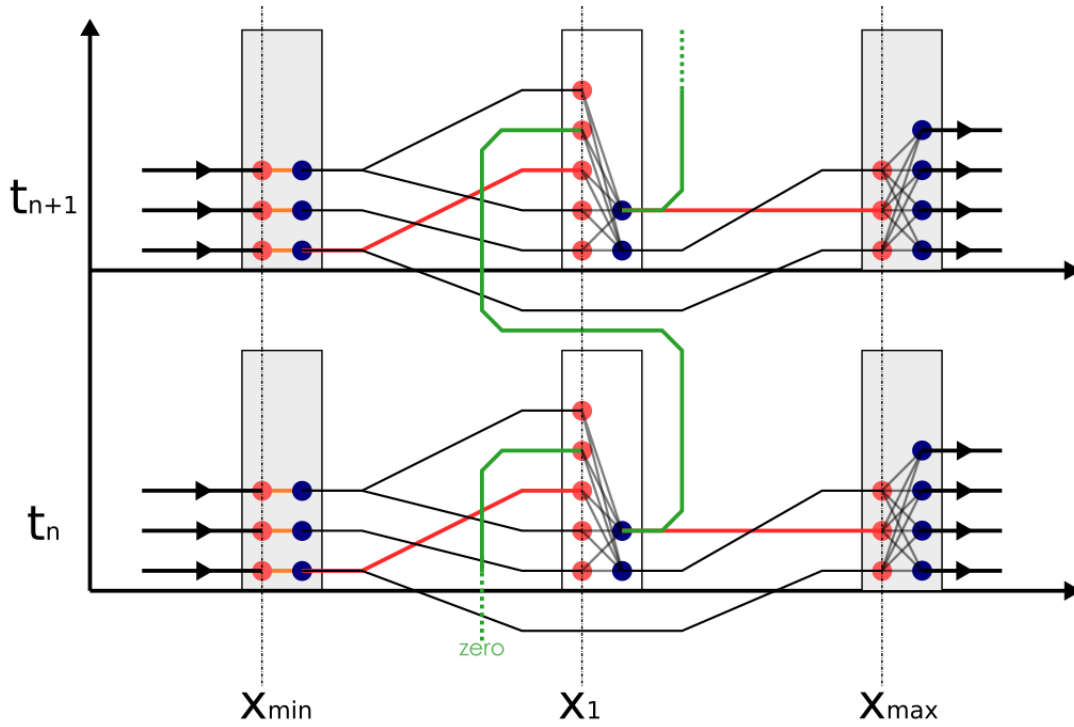


Figure 1: Réseau monocouche à connexion aléatoire; Le même réseau est représenté 2 fois à deux intervalle de temps à la suite, les nombres entrée/sorties sont pour l'exemple. De bas en haut : La commutation entre deux temps. De gauche à droite : L'entrée vers la sortie.

Comme les connexions entre l'entrée d'une couche X_n et la sortie d'une couche en X_{n+m} , avec $m \geq 0$, n'est pas possible pour le calcul de la rétropropagation à un temps donnée, on se trouve avec la connexion $X_{n+m,t-1} \rightarrow X_{n,t}$. Ce type de connexion laisse apparaitre une forme de réseau récurrent pour le calcul du gradient, mais comme nous voulons éviter les structure artificielle comme LSTM qui empêche le *vanishing gradient* des réseaux récurrents [Ref LSTM], nous avons considéré cette entrée à "t-1" comme une **entrée virtuelle**. Cette entrée virtuelle, n'est pas la solution la plus élégante, mais est la plus flexible dans le cas où les connexions sont complètement désordonnées et réduit le temps de calculs car la taille de l'entraînement 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 entrées virtuelles.

1.2 Construction du graphe

La topologie du réseau présenté précédemment peut être vu comme un graphe orienté acyclique, en effet, les connexions entre l'entrée X_n et la sortie X_{n+m} sont vues comme des entrées virtuelles indépendantes de la rétropropagation. Les connexions intercouche peuvent être négligées pour la construction initiale du graphe, car elles sont complètement connectées (voir code PyTorch *nn.linear*). Pour les connexions entre les couches, on se retrouve face à plusieurs contraintes :

- Quel est le nombre minimal de connexions pour que le réseau soit complètement connecté ?
- Combien de connexions sont attribuées par couche ?
- Comment attribuer les connexions par couche de neurone ?
- Comment stocker ces informations pour reconstruire le graphe ?

Empiriquement, le nombre total de perceptrons dans la couche intermédiaire " N_h " est initialement défini comme la racine entière du nombre d'agent par génération d'entraînement 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 intermediaire “ 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’écart 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’écart entre “ N_h ” et “ N_L ”.

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

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

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

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 attribué 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 $P(X_k) = \begin{cases} 1 & X_k = X_{k+1} - 1 \\ 0 & \text{sinon} \end{cases}$ 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

$$f_X = \begin{cases} \frac{2}{3} & \text{pour } a \leq x \leq b \\ \frac{1}{3} & \text{pour } b \leq x \leq c \text{ dans ce cas.} \\ 0 & \text{sinon} \end{cases}$$

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 amont des connection en aval. La matrice à par définition la structure d'un graph orienté.

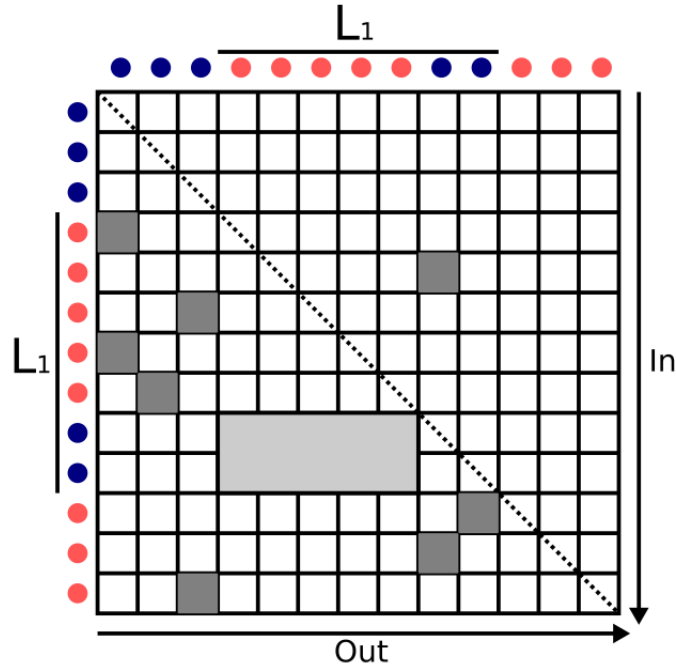


Figure 2: Matrice adjacente des liens du reseau de neurone précédent. En rouge, les noeud de connection (entrée d'une couche) et en bleu, les noeuds de perceptron (sortie de couche). Les case grisé correspondent à une connection de poids constants (1 par défauts).

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 reconstruira à posteriori. Les connection intercouches representé 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'être 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 efficacement 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 *reseau de neurones à propagation avant*. L'algorithme "forward" de la plupart des bibliothèques logiciel se retrouve restructuré.

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

```

Batch size, Input size  $\leftarrow \dim(\mathbf{X})$ 
BATCH_  $\leftarrow \text{np.arange}(\text{len}(\mathbf{x}))$ 
Trace[-1]  $\leftarrow \mathbf{Layers}[-1](\mathbf{X}.\text{view}(\text{Batch size}, \text{Input size}, 1)).\text{view}(\text{Batch size}, \text{Input size})$ 
for i, network index ordered by position do :
    tensor  $\leftarrow []$ 
    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  $\leftarrow \text{torch.cat}(\text{tensor}, \text{dim}=1)$ 
    Trace[i]  $\leftarrow \mathbf{Layers}[i](\text{tensor\_in})$ 
    Trace[-2]  $\leftarrow \mathbf{Layers}[-2](\text{tensor\_in})$ 
for t in range(len(Trace)):
    h[t][BATCH_]  $\leftarrow \text{Trace}[t][\text{BATCH\_}].\text{detach}()$ 
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'évolution du systeme correspond à un processus de decision markovien, représenté 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) :

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

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 \mathbf{T} correspond à la matrice de transition et \mathbf{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 π 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 états 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 déterminées 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 dernière correspond à l'équation de Bellman et c'est elle qui est utilisée pour calculer la fonction de coût du réseau de neurone lorsqu'on réalise une action. La première équation ne peut être utilisée seule car elle nous donne l'efficacité de l'agent pour une action donnée, mais peut être utilisée en cas d'utilisation de réseau adversariaux [Ref actor-critic]. Dans notre cas, on réalise initialement une suite d'événements avec une matrice de transition aléatoire (réseau non entraîné), puis on calcule la fonction de perte entre la prédiction des valeurs Q obtenue au cours de l'expérience avant l'action et le résultat de l'équation de Bellman Q après l'action d'un batch d'entraînement donnée, puis on répète N_{cycle} fois cette étape 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 + (1-DONE)*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 réseau, et correspond à la probabilité d'action optimale. Normalement, à chaque pas de temps, on choisit la valeur maximale en sortie du réseau et l'on attribue une probabilité d'action aléatoire, c'est le **dilemme exploration-exploitation**. Ce dilemme permet au réseau de neurone d'explorer des paramètres et de ne pas se stabiliser dans une vallée non optimale. Mais dans notre cas, nous avons choisi d'attribuer une probabilité d'action à chaque "*pas*" de l'entraînement à partir de la sortie du réseau, qui est équivalent à un dilemme 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

À partir des différents modèles de réseau entraînés à temps courts, l'objectif qui suit est de sélectionner ceux qui ont eu le meilleur score d'entraînement, puis de modifier légèrement la structure du graphe neuronal. Le calcul des scores est relatif à l'environnement d'entraînement et est très utilisé dans les modèles évolutifs. L'attribution des points sera plus détaillée dans la partie 3. Le principe est le suivant : Lors du premier entraînement à temps court, on a N_r réseaux en parallèle et distincts, chacun va suivre le processus d'entraînement vu dans la sous-partie précédente. Ensuite, les N_b réseaux ayant le meilleur score sont sélectionnés, N_b réseaux gardent la même topologie pour le cycle suivant, N_m réseaux par N_b meilleurs réseaux héritent d'une mutation aléatoire. Enfin N_c nouveaux réseaux aléatoires sont introduits dans le processus d'entraînement suivant, ceux-ci peuvent hériter de paramètres optimisés des réseaux précédents, mais pas de la structure du réseau, c'est les réseaux "compétiteurs". Les nombres d'agents sont définis comme $N_r = (N_b + 1)^2$, de façon à ce que N_r est une racine entière, $N_m = N_b$, tel que le nombre total de mutations soit $N_{mtot} = N_b^2$ et enfin $N_c = N_b + 1$, de cette façon, 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 connexion : Ne change pas le nombre de couche neuronale, mais ajoute une connexion à l'une d'entre elles. Comme on n'ajoute pas de neurone, la liste des connexions possibles ne change pas et le nouveau nœud va être connecté aléatoirement à un neurone pré-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 vu que le reseau doit etre complet, on rajoute également une connection dans l'une des couches qui va se connecter exclusivement 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 quelconque 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 incomplet. 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 décalage 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.

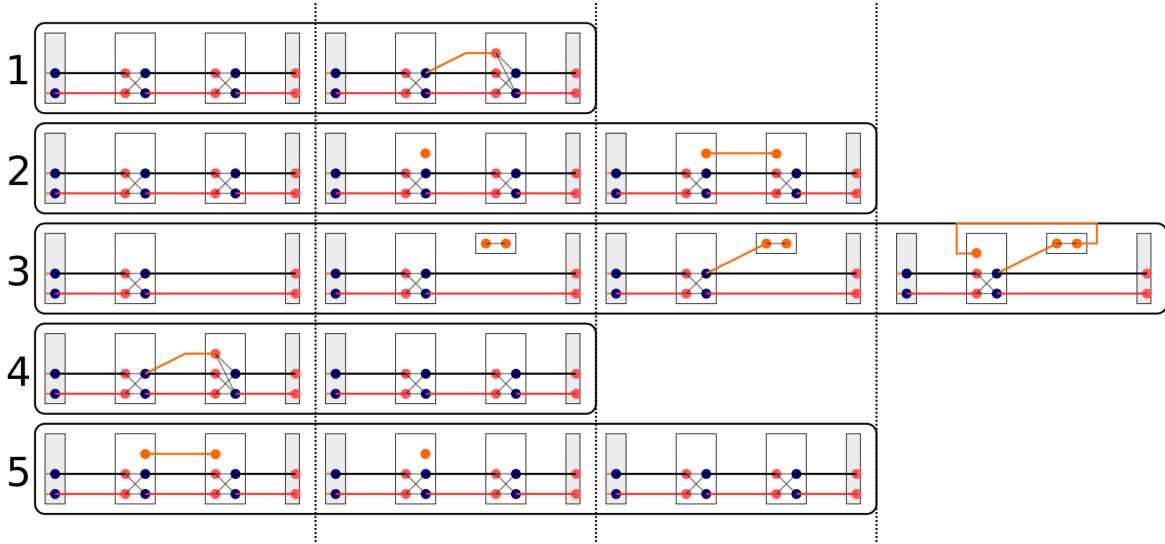


Figure 3: Algorithme visuel des mutations. Respectivement les 5 types de mutations citer en paragraphes. De gauche à droite, l'états initial vers l'état final. Les propriétés du graphe sont à titre indicatives, cette représentation très simplifiée ne reflète pas l'optimum d'une fonction. Les opérations $1 \leftrightarrow 4$ et $2 \leftrightarrow 5$ sont symétriques, l'opération 3, n'a pas de symétrie stable simple.

Dans notre modèle, 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 symétrie des opérations $1 \leftrightarrow 4$ et $2 \leftrightarrow 5$ [voir Fig 3], mais aussi de maintenir une structure vestigiale du réseau. En effet, au cours de l'évolution certaines structures sont maintenues, mais ne sont pas les choix optimaux pour remplir certaines fonctions. Par exemple, le chemin qu'emprunte le vaisseau sanguin entre la tête et le cœur chez les mammifères emprunte un chemin non optimal, pourtant il est maintenu car serait trop coûteux évolutivement pour changer de trajectoire [réf nécessaire]. La vallée de stabilité évolutive liée à ce choix critiquable est plus abordée 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’environnement d’entrainement. Cet environnement contient des regles et s’apparente à un jeu, la bibliotques la plus utilisé en python est *OpenGym* et sa structure à servi d’exemple pour realiser l’environnement 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 prédatons 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 environnement de jeu. L’agent contient les informations de **vue** et d’**action** qui va transmettre à l’environnement. L’environnement contient les information de la position de l’agent et son adversaire, ainsi que les regles de jeu et comptage de points. L’environnement 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 environnement 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 environnement 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 :

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

Enfin, l’environnement 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’environ 8 déplacement. Dans cette configuration, pour un calcul équilibré, 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.

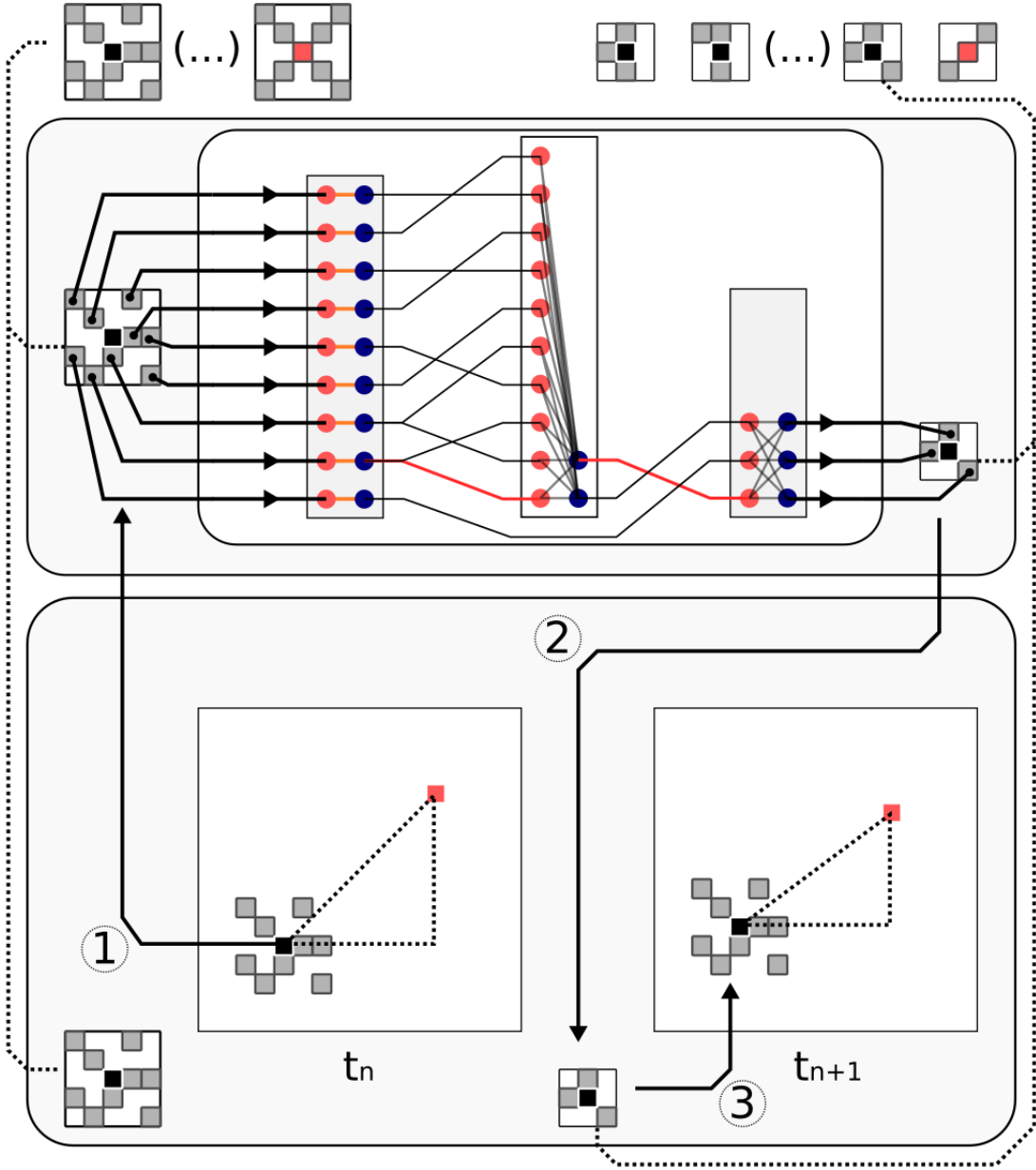


Figure 4: Les interactions entre un agent et un environnement de jeu. En haut, la génération aléatoire des positions relative possible des entree et sortie. Suivi des interactions entre l'agent en haut et l'environnement de jeu en bas. 1 : envoi de l'environnement des informations d'entrée (pixel) à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

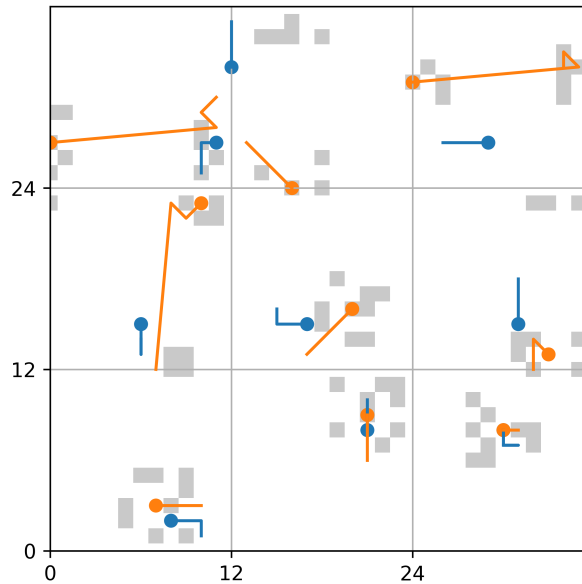
Les case des positions relative d'entrée et de sortie sont défini aléatoirement, mais comme le pourcentage des case utilisé sur la grille est au alentours de 35%, les combinaisons possibles sont de l'ordre factoriel. Pour la sortie, on a 3 positions parmi un total de 9 cases, ce qui donne $\binom{9}{3} = 84$ et comme le probleme est symetrique (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 $\binom{25}{9} = 2042975$ (probleme stabilité)

Les case en question sont choisi initialement aléatoirement, mais les configuration choisi au cours du cycle suivant depent de la probabilité des case ayant eu le meilleurs score : leurs statistique seront traité dans la partie resultats. On remarque un nombre de permutation en entrée tres elevé, mais pour la sortie/action, on observe plusieurs cas cyclique, des cas asymétrique (mais ne pose pas de probleme en cas de limite périodique) et des cas avec immobilisation possible, mais sans cycle. Comme on selectionne le meilleurs type d'entrée/sortie, il y a aussi une selection sur ce parametre (cas nouvel evenement aléatoire).

Les positions des entree/sortie sont selectionné aussi dans le processus evolutif (ecrit juste avant...).

Résultats

L'ensemble des experiences ce sont réalisé à la suite. On en a choisi 25 de facons à avoir 4 heredité et 5 nouvelles structure aléatoire à chaque cycle (comparaison efficacité neurone avec evenement aléatoire). 25 est arbitraire et principalement pour avoir une representation de l'environnement en $5*5*N*N$ taille. (representation par nombre premier sinon (mais pas optimal) : $n \% 2$ et $n \% \text{range}(3, \text{int}(\text{np.sqrt}(n))+1, 2)$)



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 :

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

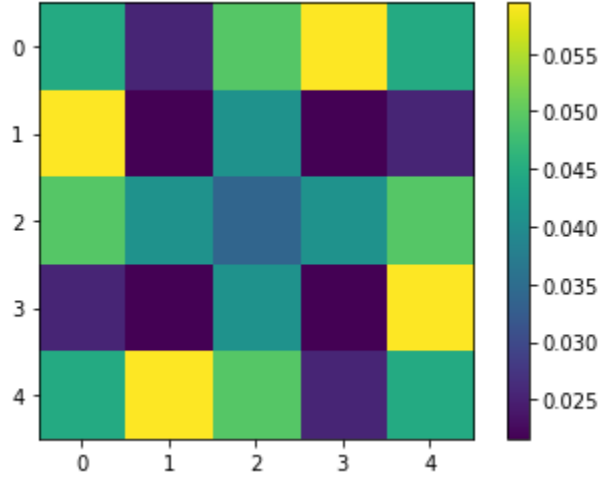
Je comprend pas d'ou ca sort : loi des grand nombre ? (inegalité de BT ?) Je preferai une demonstration de l'intervalle de confiance plutot qu'une formule toute faite...

4 Densité I/O

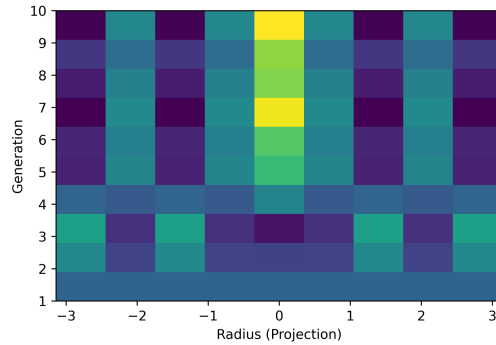
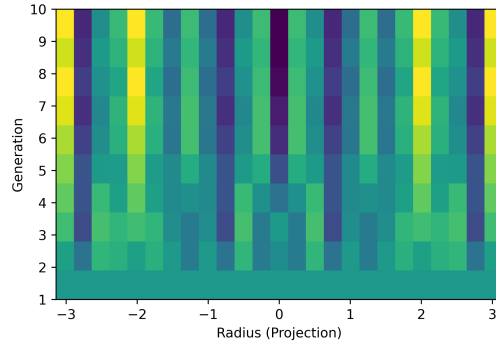
Au cours des cycles de reproduction, on met à jour la densité de probabilité de succès des I/O ayant le plus reussi pour le evenement aléatoire suivant (génération spontanée à chaque cycle). La convergence de la densité de probabilité par une variable aléatoire est defini par le theoreme de Transfert, 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 Discuter 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.

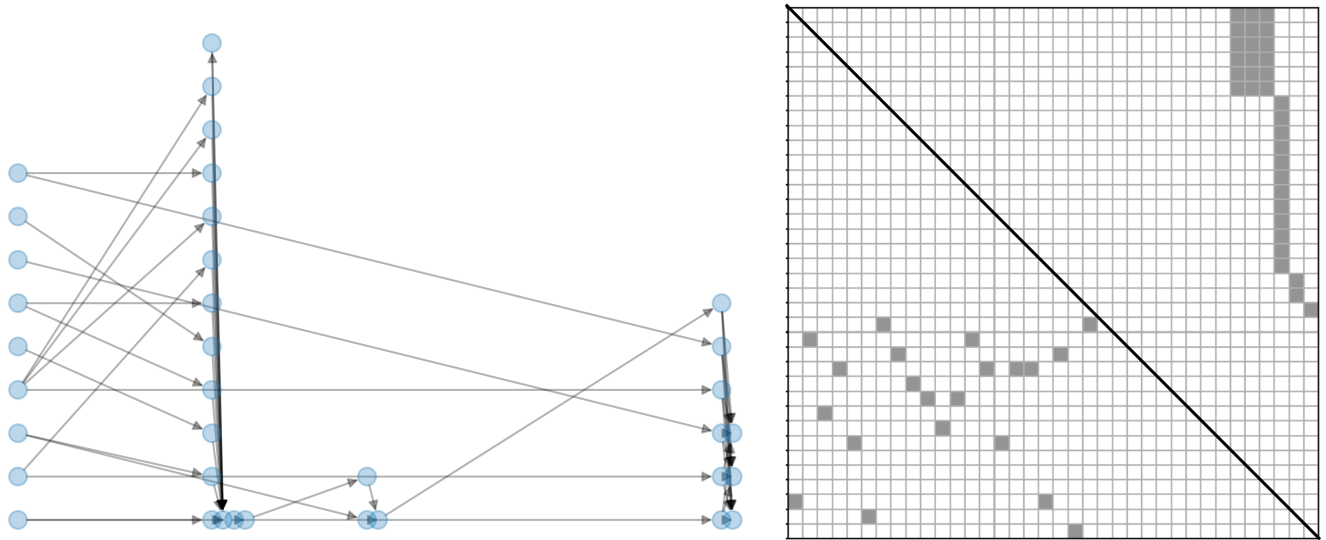


Comme les I/O sont stable (ou pas, je ne sais pas), on utilisera uniquement la densité finale pour les nouveau evenement aléatoire (ou pas). De toute facon, ce parametre(/variable) devient un invariant assez vite (comme on peut le voir dans la courbe evolution ecart : enfin c'est ce qu'a l'air de donner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

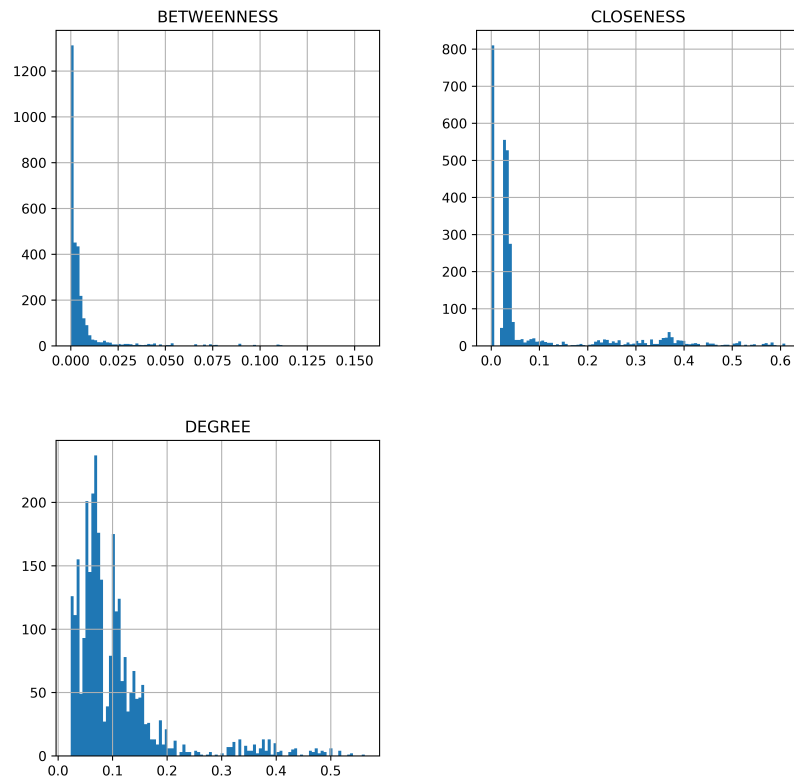
5 Connectivité des reseaux

Etudier quel “type” de structure est plus avantageuse pour ce probleme

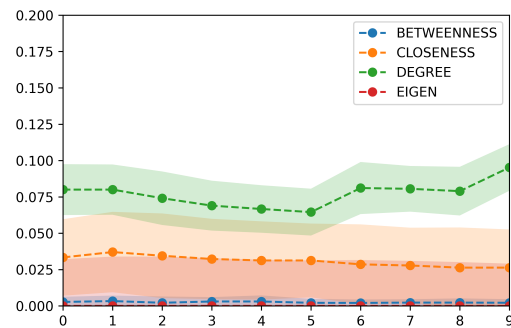
D’abord type de graphes à rapeller (dirigé évidemment), ensuite densité du graph, puis voisinage et degré.



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



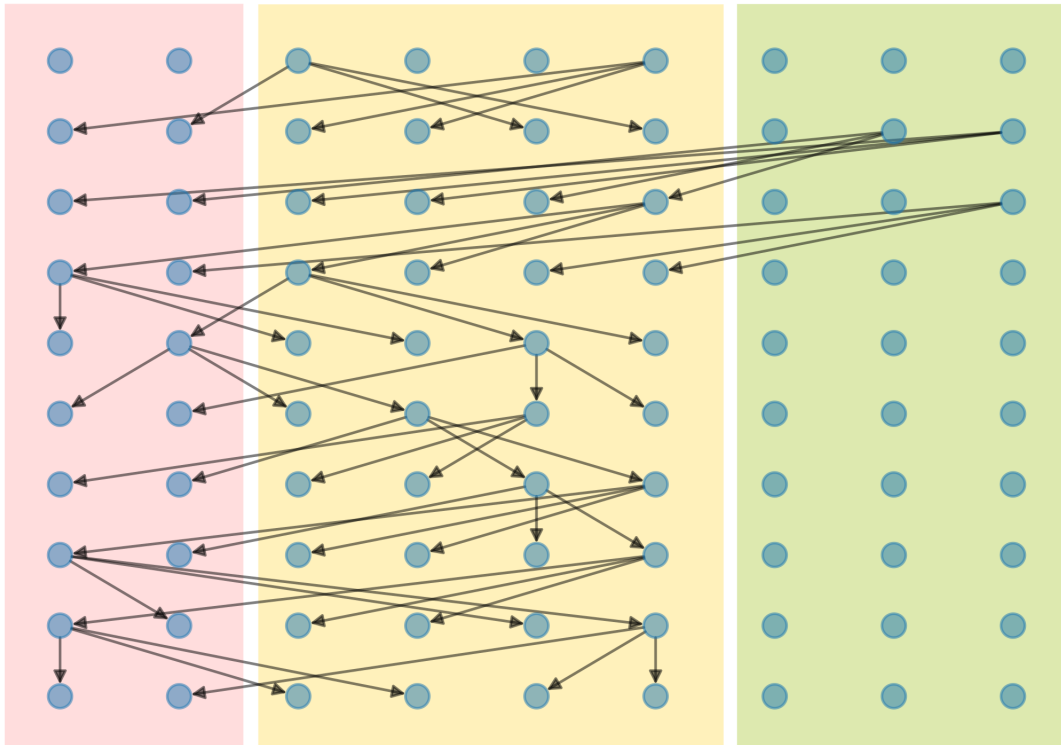
Convergence de la serie de graphe par l'evolution ?



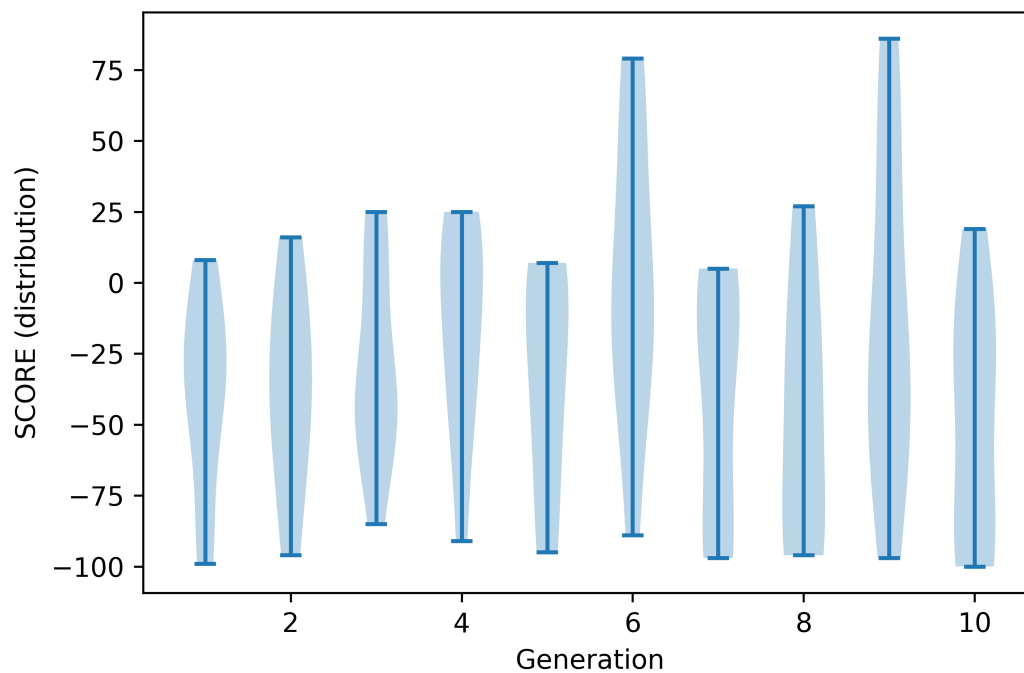
On n'observe pas de stabilité, mais les graphes les plus asymétriques marchent le mieux à court terme, mais pas à long terme (l'entraînement q-learning marche moins bien ?) → oui enfin, c'est par construction, et puis je ne sais pas comment calculer l'écart type entre deux graphes ??

6 Efficacité selective

Conclure un peu les deux données précédentes sur la convergence des I/O et de la structure du réseau.



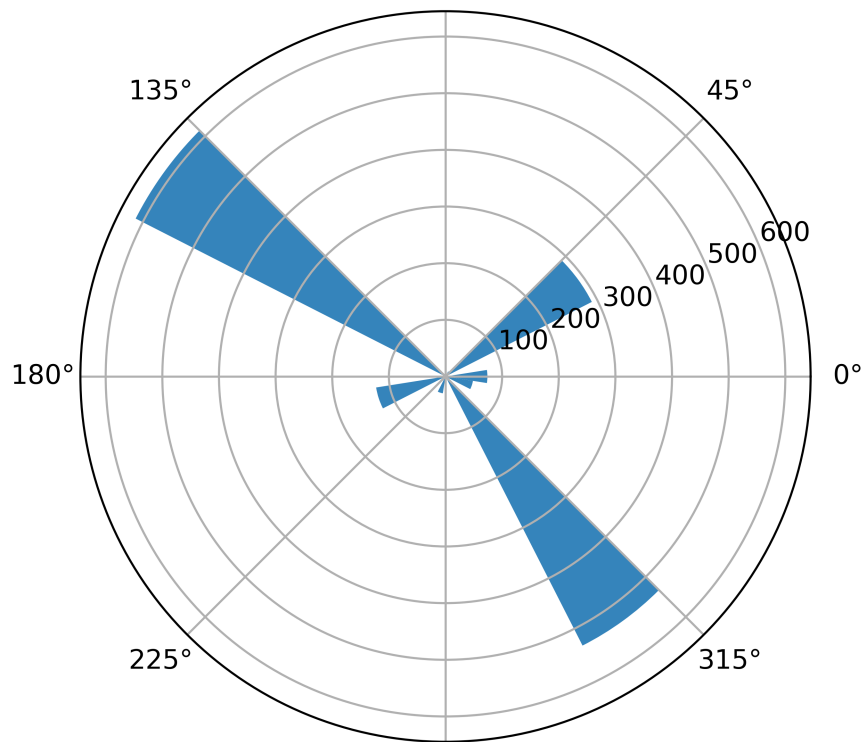
Idée ici est d'approfondir la vitesse d'apprentissage, un ce qu'un reseau tres adapté à un probleme apprend vite au debut, mais atteint un plafond moins optimale qu'un reseau moins adapté au debut, mais à la suite devient plus performant ?



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 observe quel type de trajectoire domine suivant condition limite périodique ou non.

Discussion

Paragraphe critique 1 : case carré ! pourquoi pas un cercle ? car carré problème de symétrie

Paragraphe critique 2 : réseau vestigiale n'est peut être pas optimal et empêche d'aller vers une vallée plus optimum (voir pb en bio-evo → illustration vallée d'optimalité)

Paragraphe critique 3 : Artefact de trajectoire, le réseau est avantagé si trajectoire uniquement en diagonal (+ CLP)

Paragraphe ouverture 1 : Optimisation de la structure du réseau en même temps que l'apprentissage. (citer les autres méthodes de "self-learning", NEAT, GNN, etc, mais ici dans le plus élémentaire)

Paragraphe ouverture 2 : 1 seul environnement partagé (coop vs compét)

Paragraphe ouverture 3 : Utilisation hors contexte sur base MNIST : comparaison efficacité réseau simple et réseau structuré.

IMAGE BATCH_SIZE comparaison et réseau classique (temps d'adaptation par réseau) (en fonction des cycles d'entraînement) → (comparé entre cycle d'hérédité et cycle d'entraînement)

References

- [1] Stuart Bartlett and Michael L. Wong. Defining life 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.