RLD : Rapport TME Deuxième Partie

Saadaoui Aymane & Afandi Mohammad

Février 2022

Table des matières

1 GAN **2**

2 VAE **10**

3 Multi-agents **14**

4 Imitation Learning **18**

1 GAN

Dans ce TME, nous étudions les GANs (Generative Adverserial Networks) et effectuons des expérimentations sur deux jeux de données :

1. `CelebA` : une base de protraits de célébrités
2. `MNIST`

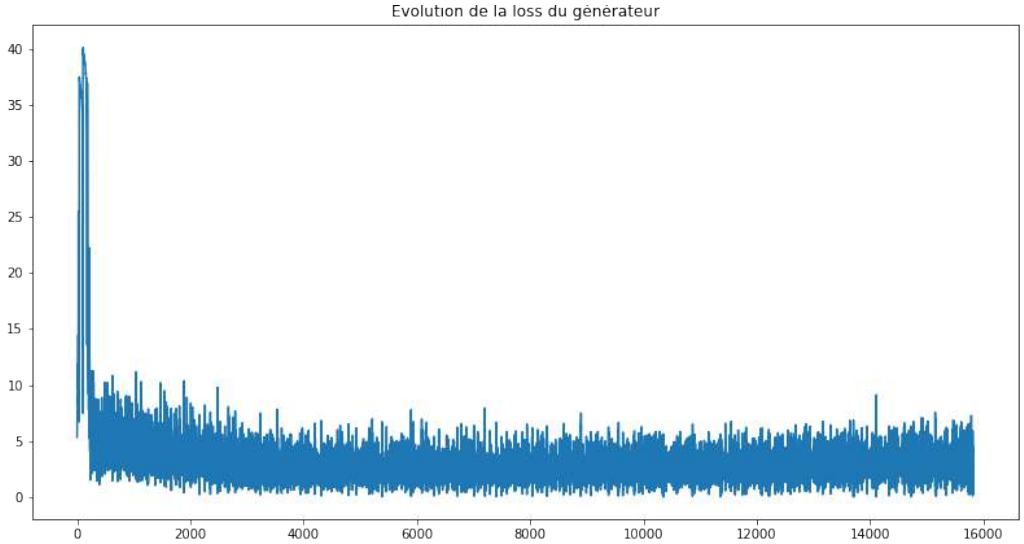
A noter qu'à cause de la limite de taille de fichiers sur Google Colab, les expériences ont été effectuées sur machine locale.

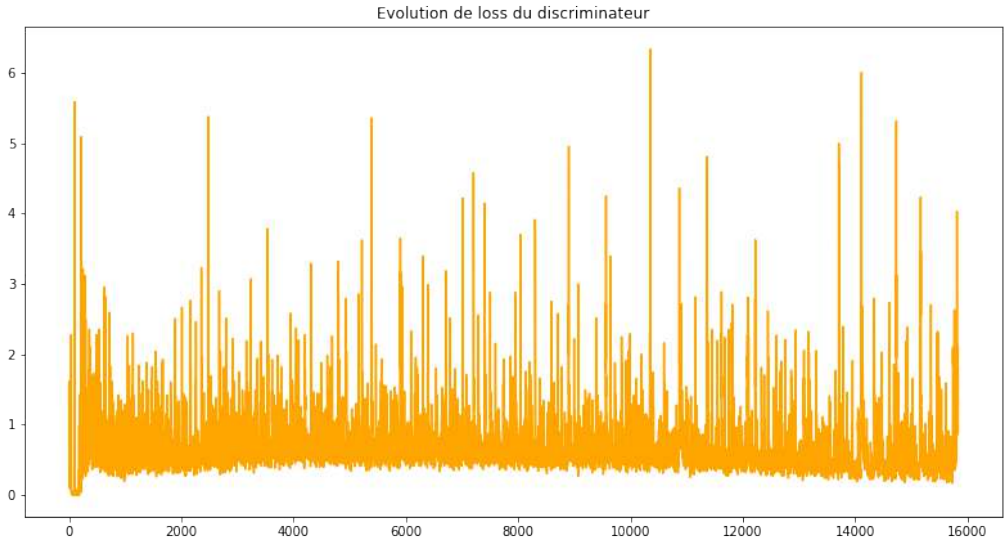
GAN

Dans un premier temps nous expérimentons les GAN sur `CelebA`. Il s'agit d'apprendre à générer des visages paraissant le plus réaliste possible à partir d'un ensemble de visage d'entraînement.
Le modèle GAN est constitué de deux modules :

- Le générateur G prend en entrée un bruit z d'un espace latent et sort une image ;
- Le discriminateur D prend en entrée une image et sort un scalaire entre 0 et 1 correspondant à la probabilité que l'image d'entrée soit bien issue de la distribution des données réelles plutôt que de celle des données générées.

La taille de batch est de 128, l'optimiseur est Adam avec $\beta_1 = 0,5$, $\beta_2 = 0,999$ et un learning rate de $0,002$. L'entraînement est effectué sur 9 epochs.





On constate que les losses ne convergent pas vraiment, il y a des oscillations tout le long de l'entraînement. Cela illustre les phénomènes antagonistes entre le générateur et le discriminateur.

Observons la génération au fil des epochs durant l'entraînement :

(a) Epoch 1 (b) Epoch 2 (c) Epoch 3

(d) Epoch 4 (e) Epoch 5 (f) Epoch 6

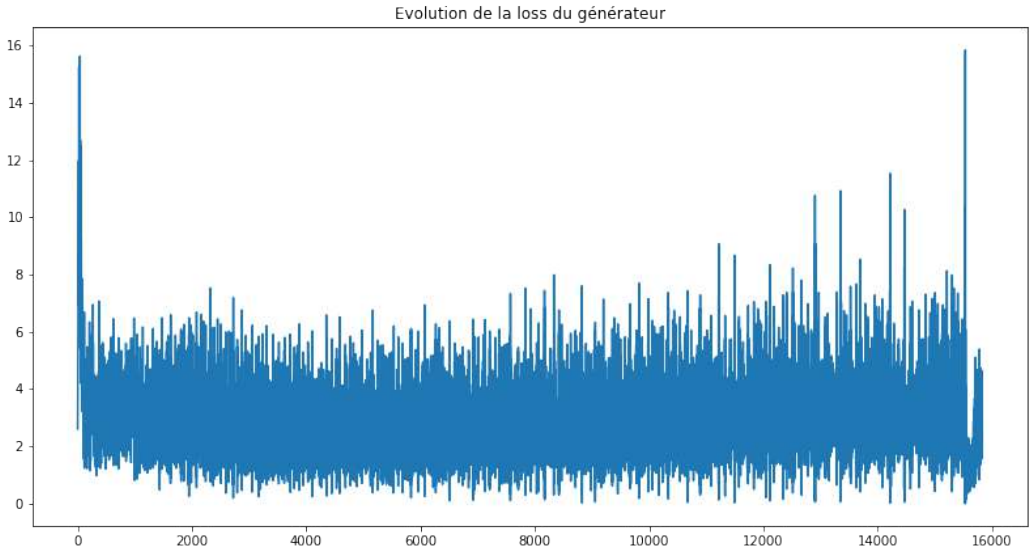
(g) Epoch 7 (h) Epoch 8 (i) Epoch 9

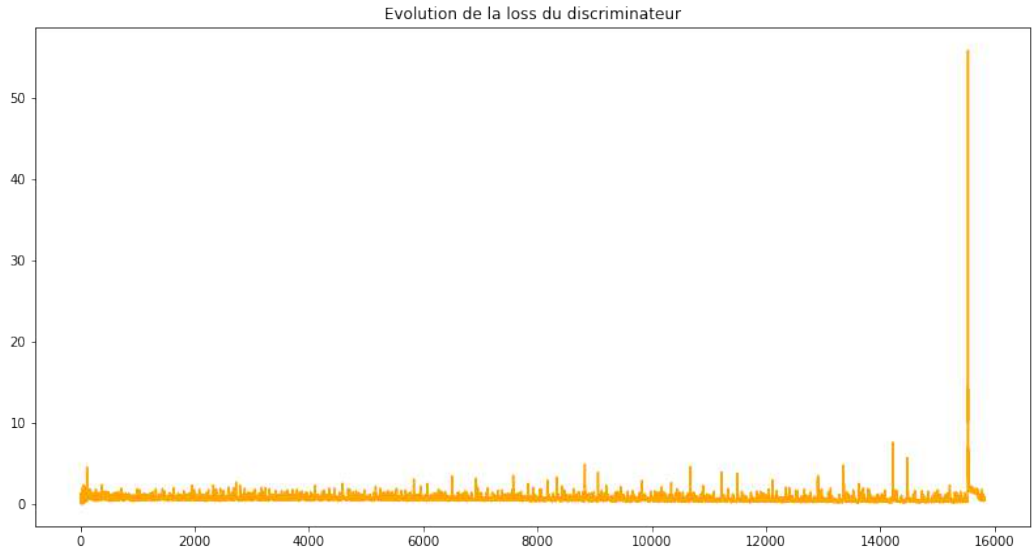
Figure 1: Images générées lors de l'entraînement (GAN)

A l'issue des 9 epochs les images générées ne sont pas parfaites mais on constate une nette amélioration au fil de l'entraînement.

DCGAN

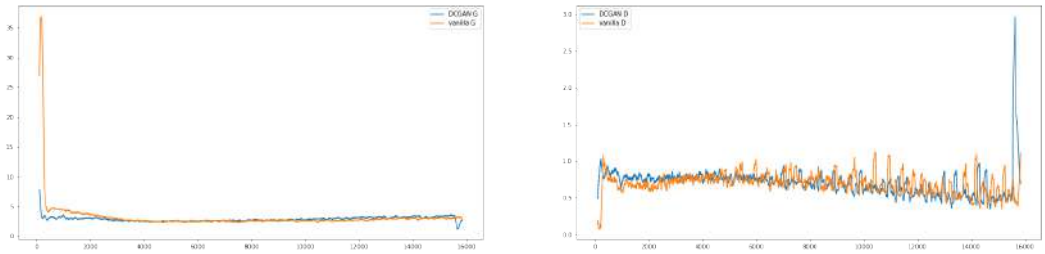
Nous implémentons ensuite un générateur DCGAN que nous allons appliquer à `CelebA`. Cette architecure ajoute une étape de projection de z depuis l'espace latent vers un tenseur $3D$ à 1024 canaux.





Là encore, les losses oscillent fortement. Il y a un pic pour la loss du discriminateur en fin d'entraînement que nous ne savons pas expliquer.

Nous pouvons comparer les losses pour le générateur et le discriminateur par rapport à ce que nous avons obtenu avec la version GAN vanilla :



Observons la génération au fil des epochs durant l'entraînement :

(a) Epoch 1 (b) Epoch 2 (c) Epoch 3

(d) Epoch 4 (e) Epoch 5 (f) Epoch 6

(g) Epoch 7 (h) Epoch 8 (i) Epoch 9

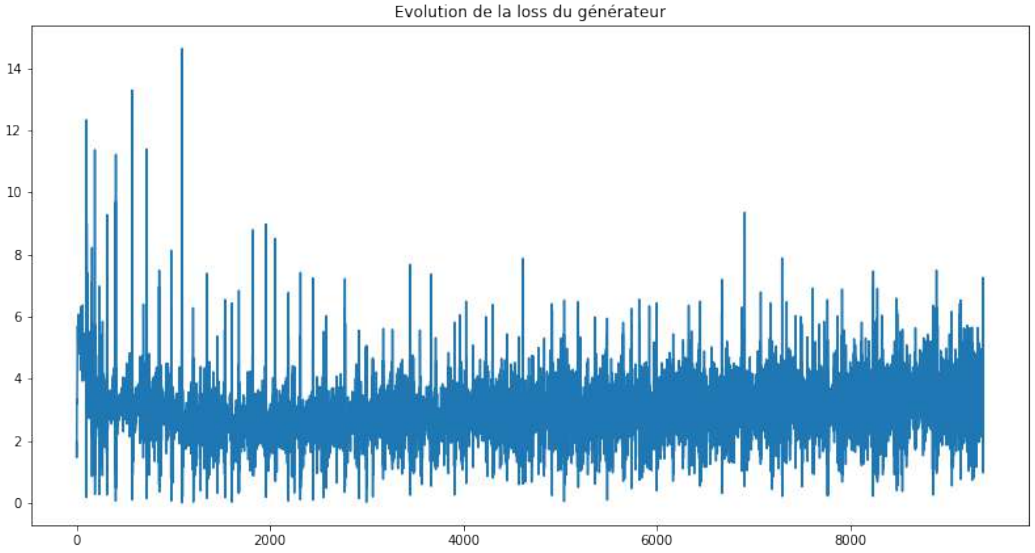
Figure 2: Images générées lors de l'entraînement (DCGAN)

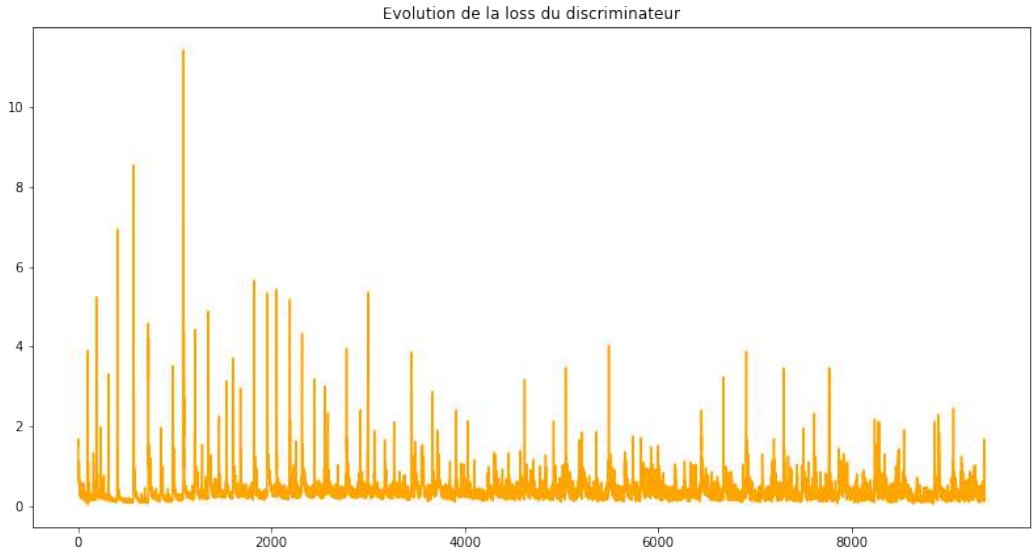
Nous ne constatons pas de différence significative au niveau de la qualité

des générations après 9 epochs par rapport au GAN Vanilla. En revanche, l'entraînement prend plus de temps : dans notre cas, il était environ 1.5x plus lent. En pratique, on privilégiera donc l'artchitecture classique dans la plupart des cas.

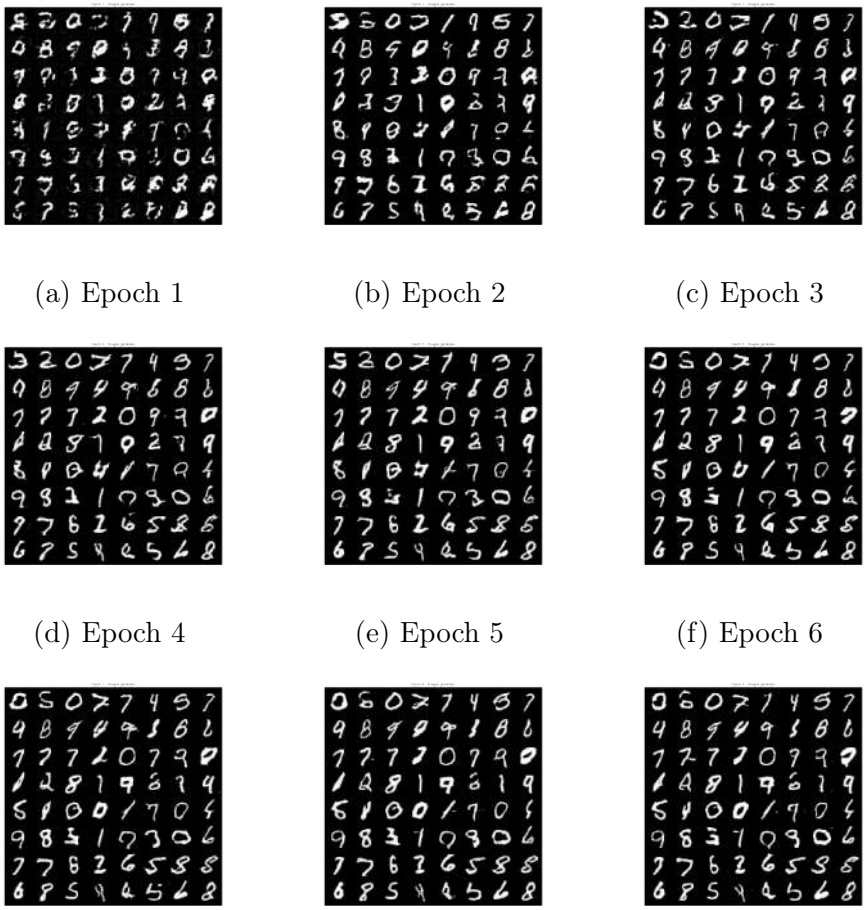
MNIST

Intéressons-nous désormais aux résultats qu'il est possible d'obtenir sur MNIST. Les images sont ici plus simples car en nuances de gris et de dimension plus petite (28×28). Nous utilisons un GAN simple auquel nous avons retiré un bloc de convolution et où nous avons réduit la taille de batch à 64.





Les losses montrent la même sorte d'évolution que précédemment.



(a) Epoch 1 (b) Epoch 2 (c) Epoch 3

(d) Epoch 4 (e) Epoch 5 (f) Epoch 6

(g) Epoch 7 (h) Epoch 8 (i) Epoch 9

Figure 3: Images générées lors de l'entraînement (GAN)

Les résultats de générations semblent satisfaisants pour MNIST avec un GAN simple. On constate toutefois que certaines images ne ressemblent pas vraiment à des chiffres réels. On aurait peut-être pu employer une autre architecture pour éviter ce problème, comme par exemple des GANs conditionnels. En effet, avec une telle architecture nous pourrions conditionner la génération par une étiquette correspondant à la classe par exemple.

2 VAE

Dans ce TME nous entraînons un auto-encodeur variationnel (VAE) sur le jeu de données MNIST. Le VAE est constitué de deux modules (réseaux de neurones) :

- Un encodeur qui apprend une représentation des images en entrée en tant que moyenne et covariance diagonale d'une loi normale dans un espace latent de dimension d.
- Un décodeur qui échantillonne un vecteur normal z de cette loi à partir duquel il apprend à reconstruire les images d'origine.

Les images de MNIST étant en nuances de gris, on peut les représenter par des vecteurs 2D de pixels prenant leurs valeurs entre 0 et 1. Par conséquent, on peut les assimiler à des réalisations de variables aléatoires suivant une loi de Bernoulli.

On veut maximimser l'espérance pour la densité de z de la log-vraisemblance des données qu'on a supposé suivre une distribution de Bernoulli. Cela revient donc à minimiser l'entropie croisée entre les probabilités sorties par le décodeur pixel par pixel et les véritables valeurs de ces probabilités sur les images réelles.

La fonction de coût est la somme d'un terme de reconstruction et d'une pénalisation KL.

Nous avons implémenté deux architectures :

- Une architecture linéaire avec encodeur $linaire \rightarrow ReLU \rightarrow linaire$ et décodeur $linaire \rightarrow ReLU \rightarrow linaire \rightarrow sigmode$;
- Une architecture convolution/déconvolution comme pour GAN.

Pour chacune de ces architectures nous avons implémenté 3 modèles différents selon la dimension de l'espace latent : 2, 16 et 64. L'entraînement a été effectué sur 15 epochs.
Nous pouvons observer la qualité de la reconstruction des images au cours de l'entraînement :

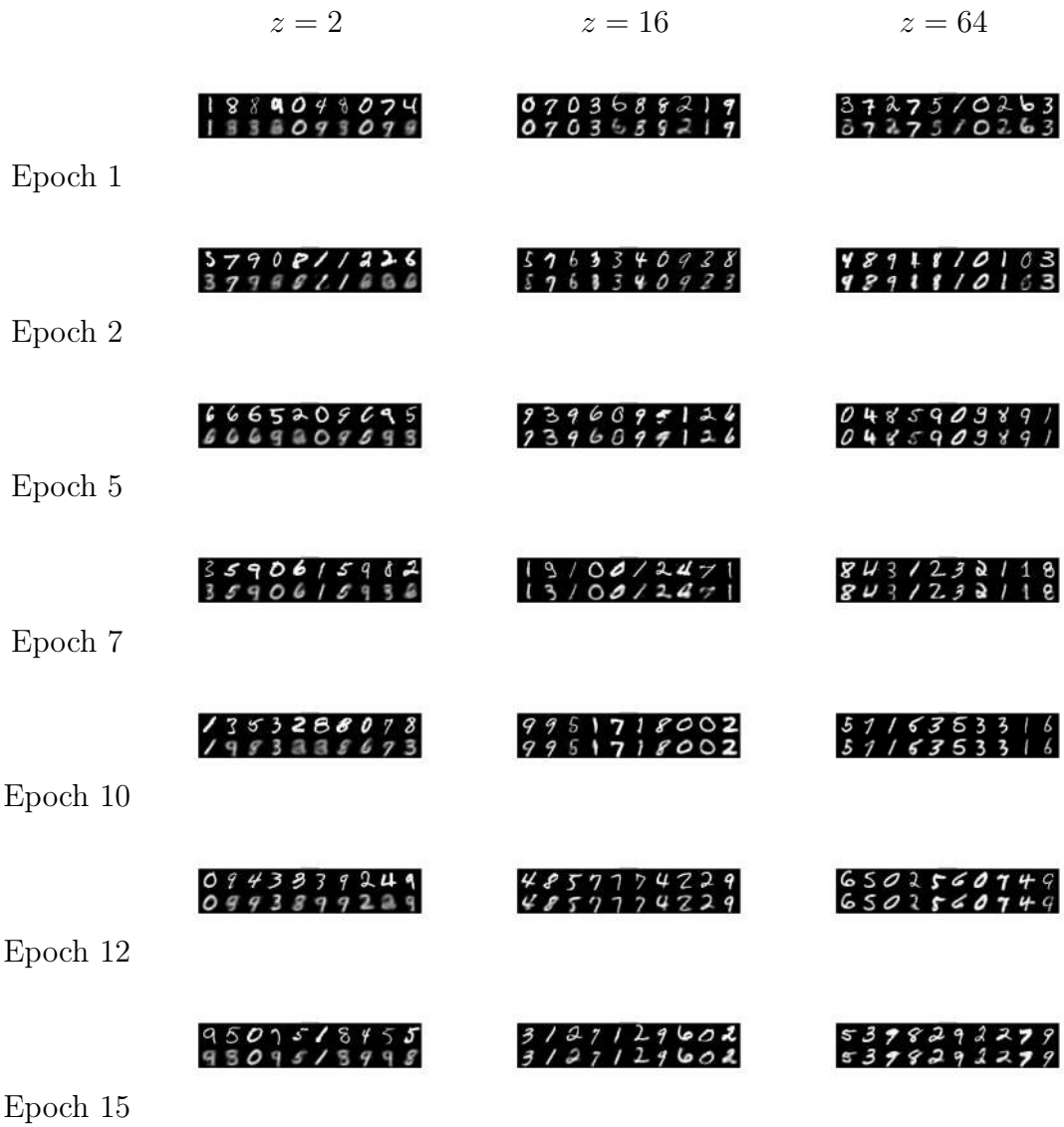


Figure 4: Reconstruction au cours de l'entraînement pour l'archi linéaire

On remarque plus la dimension de l'espace latent est grande, plus la reconstruction est nette. En dimension 2, la reconstruction est floue. Néanmoins, le gain n'est pas évident entre la dimension 16 et la dimension 64, on peut en conclure qu'il est intéressant d'augmenter la dimension de l'espace latent, mais que trop l'augmenter n'est pas forcément utile.

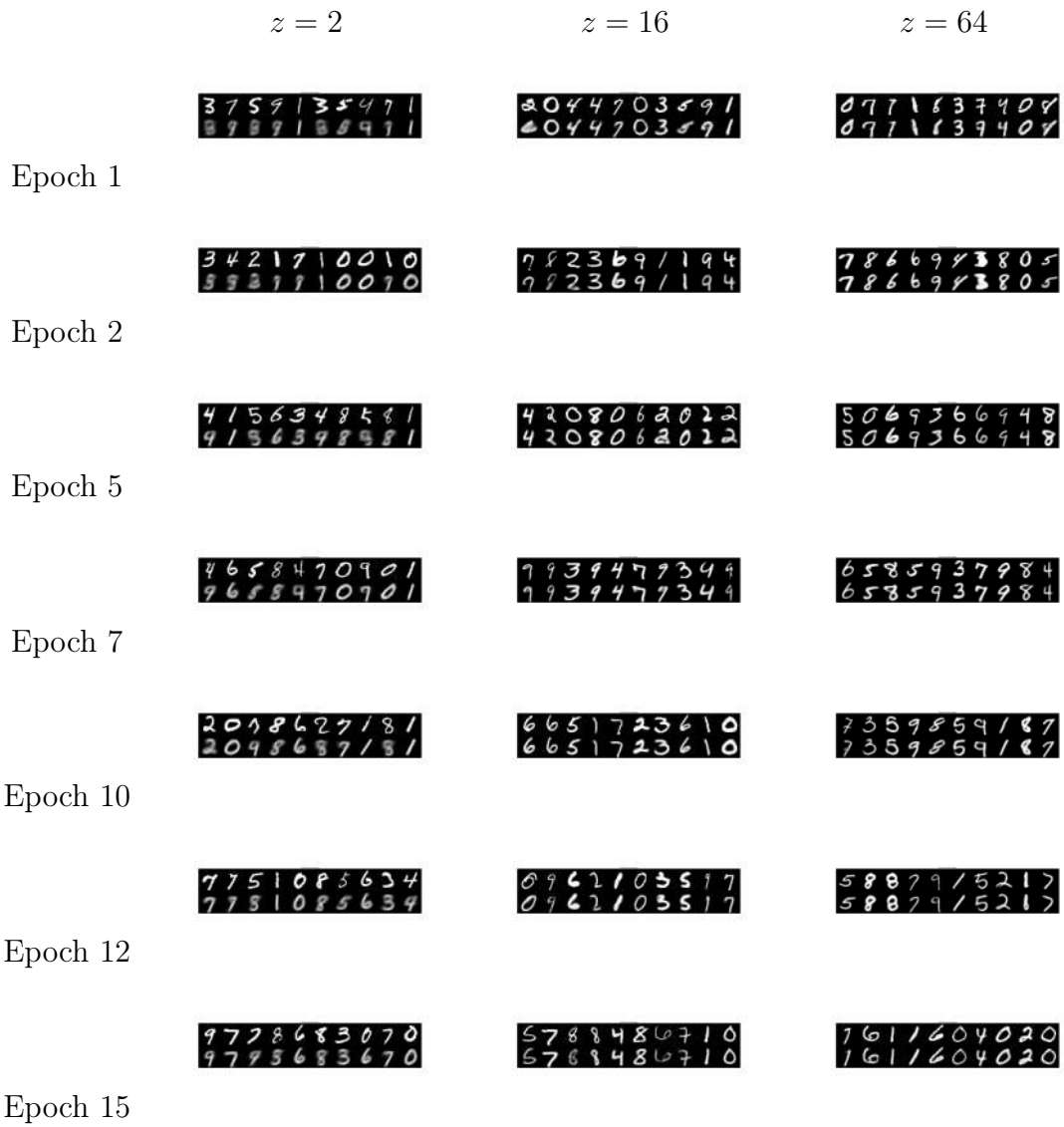


Figure 5: Reconstruction au cours de l'entraînement pour l'archi conv

Les reconstructions pour l'architecture convolution/déconvolution semblent être de qualité un peu meilleure que pour l'architecture linéaire. Cela vient sans doute de la prise en compte de la structure des régions qu'apporte la composante convolutive.

Nous pouvons nous intéresser aux embeddings dans l'espace latent des images de test en fonction de leur label en prenant les deux premières coordonnées du vecteur μ :

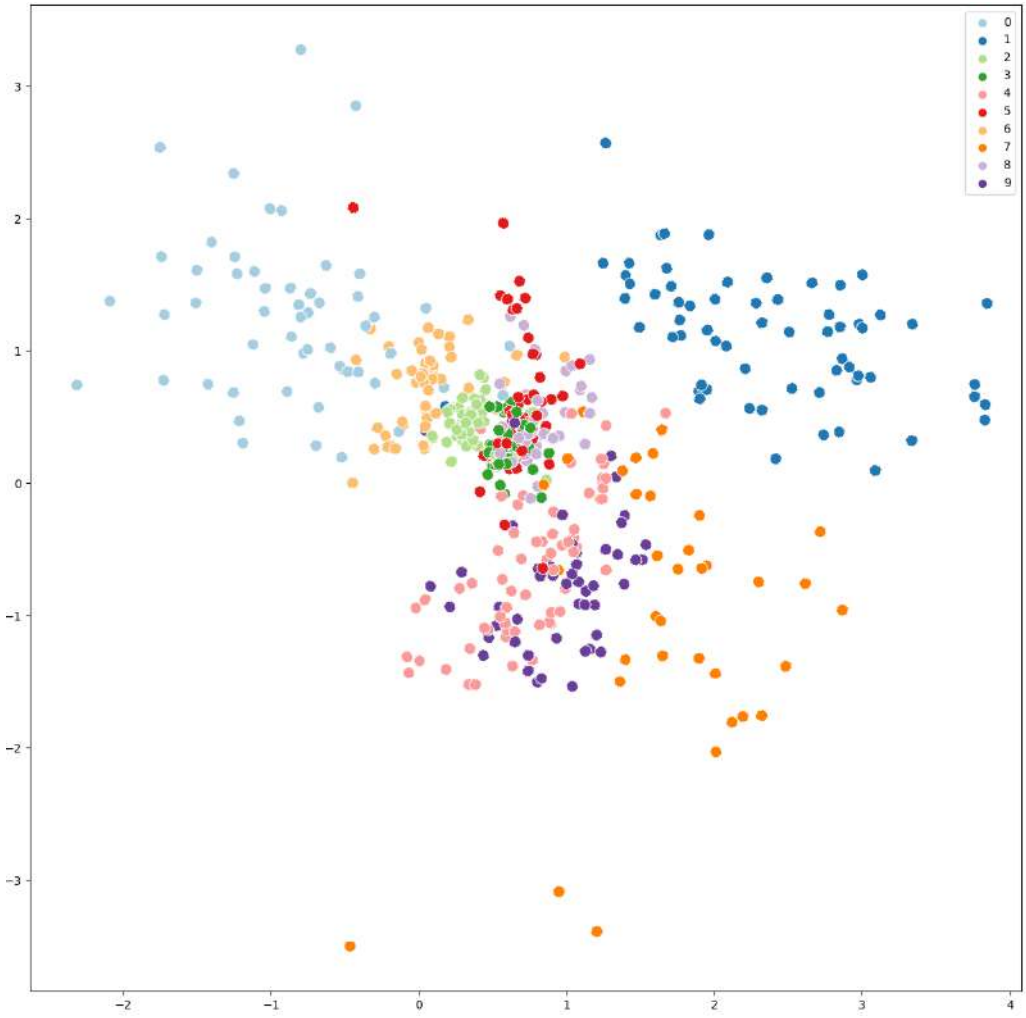


Figure 6: Embeddings dans l'espace latent

On peut voir sur cette représentation que certains labels forment des groupes plus distincts que d'autres. C'est le cas particulièrement pour 0 et 1, projetés bien à part du reste, et éventuellement 7. Les autres classes sont davantage confondues. Les nombres ayant des formes semblables ont tendance à être projetés proches les uns des autres.

3 Multi-agents

Dans ce TME, nous expérimentons l'apprentissage par renforcement multi-agents à l'aide de l'algorithme MADDPG sur l'environnement multiagents. Il s'agit d'un algorithme utilisant un apprentissage des critique centralisée et une execution decentralisée avec de multiples agents sur un environnement.

Chaque agent possède un réseau actor et crtic dont le critic sera conditionné par rapport aux actions des autres agents (Centralized). Cela permet de conserver des probabilités de transitions qui sont stationnaires.

On utilise MADDPG sur des environnements différents :

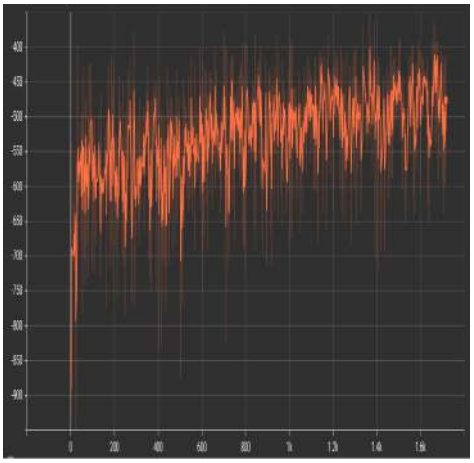
- un environnement coopératif, `simple_spreead`
- Un environnement adversaire, `simple_adversary`

Simple Spread

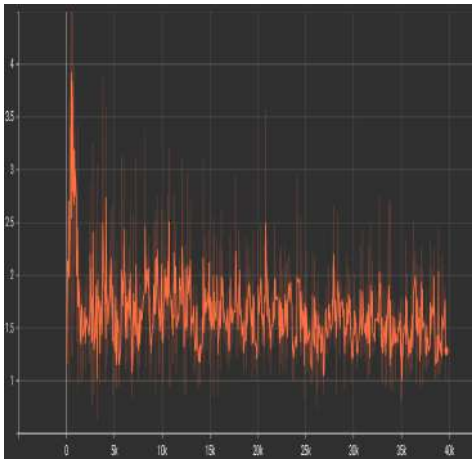
Il s'agit d'un environnement de navigation en coopération adapté à des problèmes de conduite autonome avec plusieurs agents par exemple.

Dans notre configuration on considère 3 agents et 3 points de repère. Les agents sont récompensés en fonction de la distance qui les sépare de chaque point de repère. Les agents sont pénalisés s'ils entrent en collision avec d'autres agents. Les agents doivent donc apprendre à couvrir tous les points de repère en évitant les collisions entre eux.

Tous nos agents sont identiques, et reçoivent la même récompense.
On observe le reward cumulé total de tout nos agents ainsi que la perte du critique, notre algorithme apprend à se placer sur les points de repère sans entrer en collision petit à petit.



(a) Rewards cumulé des 3 agents



(b) Loss du critic d'un des agents

Figure 7: Résultats sur l'environnement simple spread

Simple Adversary

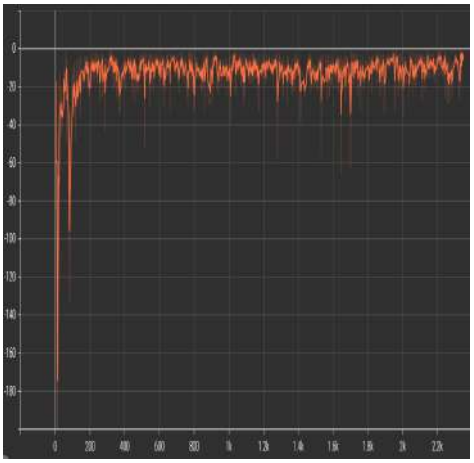
Il s'agit d'un environnement de compétition. Ici on possède un chasseur (rouge), 2 proies (vert) et 2 points de repère. Tous les agents observent la position des points de repère et des autres agents.

Les proies sont récompensés en fonction de la proximité de l'un d'entre eux avec le point de repère cible (spécifié pour chacune des proies), mais ils sont récompensés négativement si le chasseur est proche du point de repère cible.

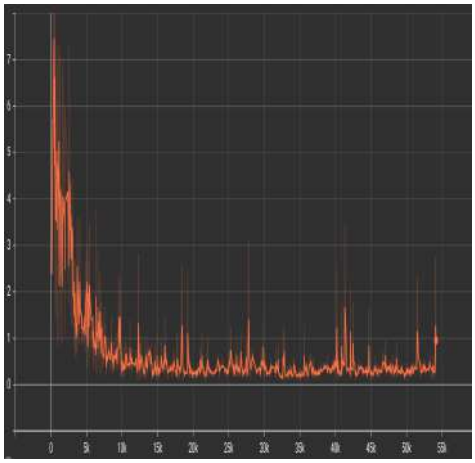
L'adversaire est récompensé en fonction de sa proximité avec la cible, mais il ne sait pas quel est le point de repère de la cible. Les proies doivent donc apprendre à se " séparer " et à couvrir tous les points de repère pour tromper l'adversaire.

Ici l'agent 1 correspond à l'adversaire et les agents 2 et 3 sont les proies.

On observe le reward cumulé total de tout nos agents ainsi que la perte du critique et le reward pour le chasseur et une proie. Les proies surpassent légerement le chasseur et arrivent à gagner face à lui.

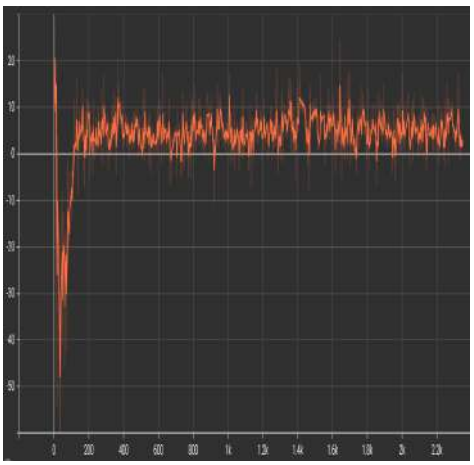


(a) Reward du chasseur

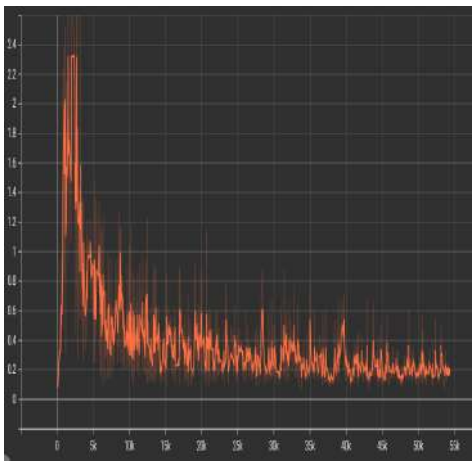


(b) Loss du critic du chasseur

Figure 8: Résultats du chasseur



(a) Reward de la proie



(b) Loss du critic de la proie

Figure 9: Résultats d'une des proies

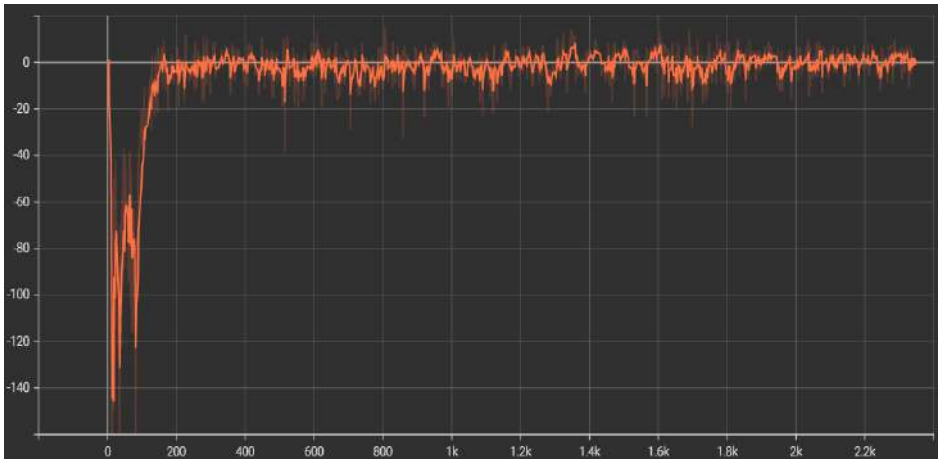


Figure 10: Rewards cumulés du chasseur et des proies

4 Imitation Learning

Dans ce TME, nous expérimentons l'apprentissage par imitation à l'aide de l'algorithme GAIL sur l'environnement Lunar-Lander version continue.

Modèle

On utilise GAIL avec une mise à jour de type Clipped PPO avec des rewards qui sont clippés ainsi qu'en ajoutant du bruit aux transitions avant chaque mise à jour du discriminateur.

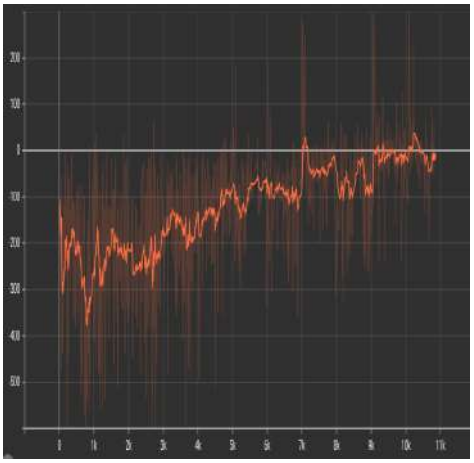
$\epsilon : N(0, 0.01)$
μ : 3e-4
γ : 0.99
β : 1e-3
Intervalle de clip : $[-100, 0]$

Le discriminateur est une réseau composé d'une couche cachée de 64 neurones avec activation ReLU et activation finale Sigmoid.
L'acteur est un réseau à 2 couches de 64 et 32 neurones avec activation ReLU et activation finale Sigmoid. L'optimisateur est Adam pour l'acteur et le discriminateur avec un learning rate commun μ.

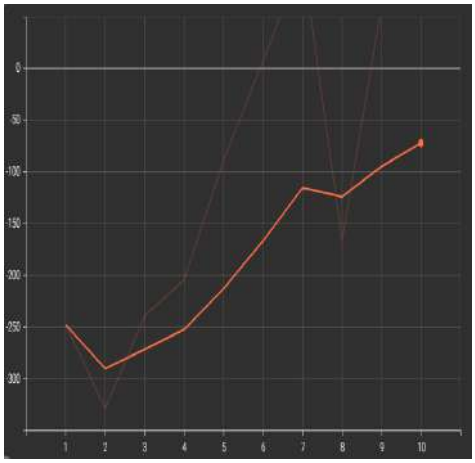
Expériences

Les rewards obtenus à l'aide de l'agorithme GAIL semblent bien meilleurs que ceux obtenus par Behavioral Cloning.

On obtient des résultats similaires à ceux du benchmark.

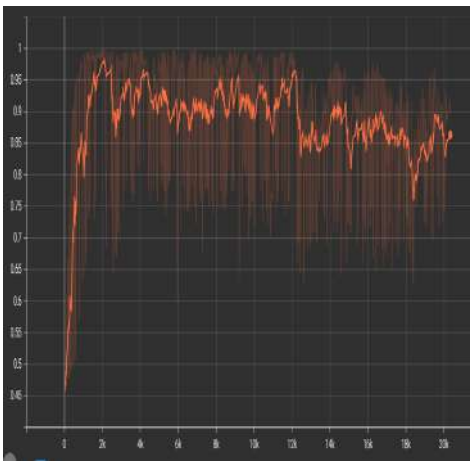


(a) Reward GAIL

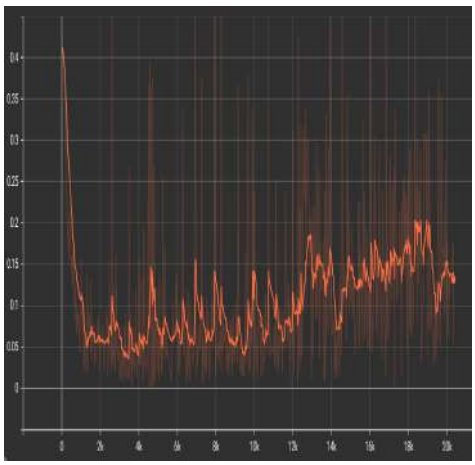


(b) Reward en test de GAIL

Figure 11: Rewards



(a) Expert



(b) Politique

Figure 12: Score du discriminateur