



---

## TME RLD 9-14

---

*Auteur*  
Youva ADDAD

*Enseignant*  
Sylvain LAMPRIER  
Benjamin PIWOWARSKI  
Nicolas BASKIOTIS

# Table des matières

---

<b>1 Generative Adversarial Networks</b>	<b>2</b>
<b>2 Variational autoencoders</b>	<b>7</b>
<b>3 Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments</b>	<b>12</b>
3.1 Cooperative navigation (Simple Spread) :	12
3.2 Physical deception (Simple Adversary) :	13
3.3 Predator Prey (Simple Tag) :	14
3.4 Amélioration :	15
<b>4 Imitation Learning</b>	<b>18</b>
4.1 Behavioral Cloning :	18
4.2 Generative Adversarial Imitation Learning :	18
<b>5 Automatic Curriculum RL</b>	<b>20</b>
5.1 DQN avec buts :	20
5.2 Hindsight Experience Replay :	22
<b>6 Modèles de flux (Flow-based)</b>	<b>24</b>
6.1 Transformation affine :	24
6.2 Glow :	27

# 1 Generative Adversarial Networks

Il y a deux composants dans un GAN un générateur et un discriminateur. Le générateur est un modèle dirigé par des variables latentes qui génère de manière déterministe des échantillons  $\mathbf{x}$  depuis  $\mathbf{z}$ , et le discriminateur est une fonction dont le travail est de distinguer entre les échantillons du jeu de données réel et du générateur. Le générateur et le discriminateur jouent tous les deux un jeu minimax à deux joueurs, le générateur minimise un objectif  $p_{\text{data}} = p_g$  et le discriminateur maximise l'objectif  $p_{\text{data}} \neq p$ , Formellement, l'objectif GAN se formule de la manière suivante :

$$\min_G \max_D V(D, G) = E_{x \sim p_{\text{data}}(x)}[\log D(x)] + E_{z \sim p_z(z)}[\log(1 - D(G(z)))] \quad (1)$$

nous allons donc essayer d'utiliser les datasets CelebA et MNIST afin d'entrainer un générateur afin de générer des images ressemblantes à leurs dataset respectivement.



FIGURE 1 – Samples du dataset CelebA

Dans un premier temps donc, nous avons utilisé l'architecture donner avec un  $lr\_d = 0.0002$ ,  $lr\_g = 0.0005$ , un  $\beta_1 = 0.5$ , un  $\beta_2 = 0.999$  qui seront donc couplé avec *Adam* optimizer ainsi que 20 epochs pour l'entraînement, nous effectuons une classification binaire, nous attribuons la probabilité 1 aux points de données de l'ensemble d'apprentissage  $\mathbf{x} \sim p_{\text{data}}$ , et attribuons la probabilité 0 aux échantillons générés  $\mathbf{x} \sim p_g$ .

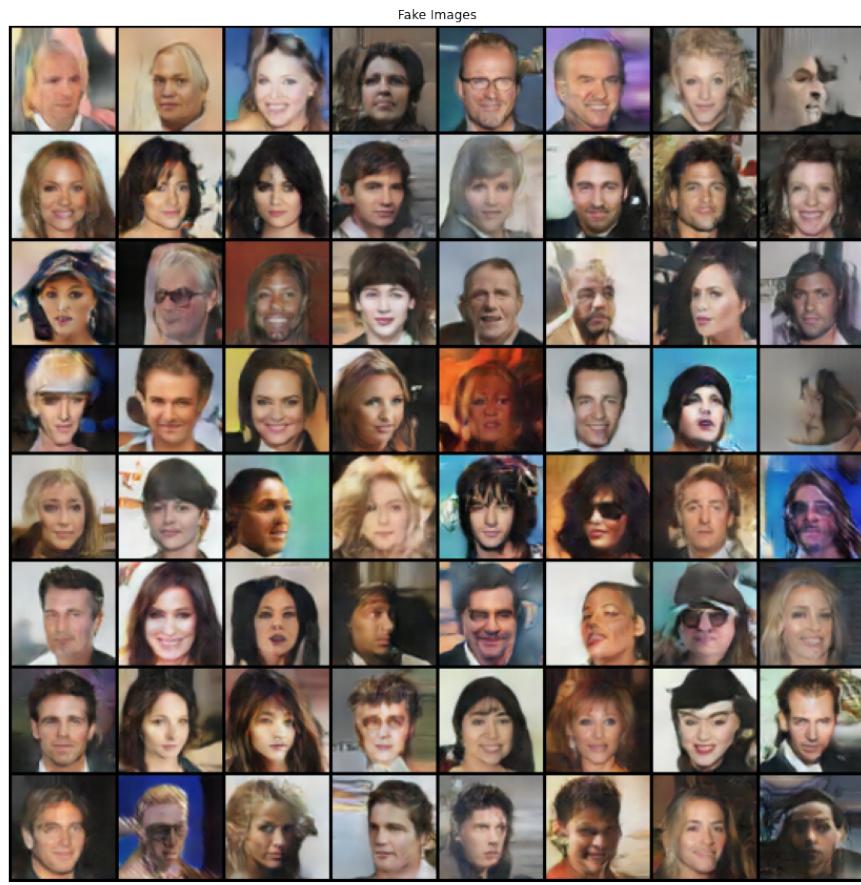


FIGURE 2 – Samples générés sur CelebA

Nous avons donc obtenu des résultats variés, nous pouvons clairement distinguer les visages mais la qualité de la génération n'est pas des plus parfaites.

Nous allons donc essayer d'améliorer l'architecture du Générateur, nous allons rajouter donc une couche de taille 5 aux transposed convolution, et un padding de 2 et un output\_padding de 1 afin d'obtenir les sorties de la taille désirée, tout en gardant les mêmes paramètres d'apprentissage que précédemment.

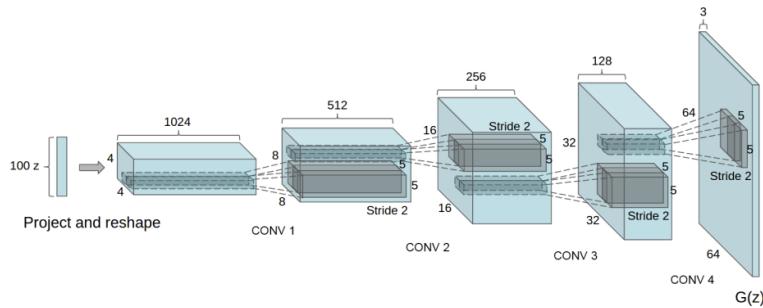


FIGURE 3 – Architecture du Générateur



FIGURE 4 – Samples générés sur CelebA avec la nouvelle architecture du générateur

Nous avons obtenu un résultat un peu meilleur que précédemment, cela reste tout de même insuffisant.

Nous allons essayer une architecture adaptée sur le dataset MNIST, afin de voir les résultats sur un dataset moins difficile à apprendre, nous avons omis le dernier block pour le générateur et le discriminateur afin d'avoir une taille de 32, sachant que les images MNIST, elles-mêmes ont été redimensionnées pour coller à une taille de 32.



FIGURE 5 – Samples générer sur Mnist

Même remarque que précédemment, nous pouvons distinguer nettement les chiffres, ainsi que leurs diversités, mais nous pouvons de plus dire qu'il existe des chiffres ne ressemblent pas à des chiffres déjà existants.

Finalement, nous avons changé l'architecture du générateur afin de pouvoir utiliser des convolutions, pour cela, nous allons utiliser un upsample avec un mode 'nearest' suivi d'une convolution.



FIGURE 6 – Samples générer sur Mnist avec la nouvelle architecture pour le générateur

Nous obtenons donc de meilleurs résultats que précédemment, des chiffres divers et variés, avec une qualité nettement meilleur, cela en gardant les mêmes paramètres que sur CelebA.

## 2 Variational autoencoders

Un auto-encodeur variationnel est un modèle génératif. les VAE sont destinés à compresser les informations d'entrée dans une distribution latente multivariée contrainte (encodage) pour les reconstruire aussi précisément que possible (décodage), les VAE marier modèles graphiques et deep learning et cherche à modeliser la distribution des données.

- Un encodeur  $\mathcal{E} : R^n \rightarrow R^d$
- Un décodeur  $\mathcal{D} : R^d \rightarrow R^n$

On approche le posterior par une distribution  $q_\phi(\mathbf{z} | \mathbf{x})$  paramétrée par  $\phi$ . Sous réserve que  $q_\phi(\mathbf{z} | \mathbf{x})$  soit bien construite, cette distribution (connue) donne accès à des valeurs de  $\mathbf{z}$  étant vraisemblablement à l'origine d'un  $\mathbf{x}$ , avec  $\phi$  nous indiquons les paramètres de ce modèle d'inférence, aussi appelés les paramètres variationnels, nous optimisons les paramètres variationnels  $\phi$  tels que :

$$q_\phi(\mathbf{z} | \mathbf{x}) \approx p_\theta(\mathbf{z} | \mathbf{x}) \quad (2)$$

Nous souhaitons avoir une distribution la plus proche possible du vrai posterior  $p_\theta(\mathbf{z} | \mathbf{x})$   
 $\hat{\phi} = \underset{\phi}{\operatorname{argmin}} \text{KL}(q_\phi(\mathbf{z} | \mathbf{x}) \| p_\theta(\mathbf{z} | \mathbf{x}))$

$$\begin{aligned} \text{KL}(q_\phi(\mathbf{z} | \mathbf{x}) \| p_\theta(\mathbf{z} | \mathbf{x})) &= E_{q_\phi(\mathbf{z} | \mathbf{x})} [\log q_\phi(\mathbf{z} | \mathbf{x}) - \log p_\theta(\mathbf{z} | \mathbf{x})] \\ &= E_{q_\phi(\mathbf{z} | \mathbf{x})} [\log q_\phi(\mathbf{z} | \mathbf{x}) - \log p_\theta(\mathbf{x} | \mathbf{z}) - \log p_\theta(\mathbf{x})] + \log p_\theta(\mathbf{x}) \\ &= -E_{q_\phi(\mathbf{z} | \mathbf{x})} [\log p_\theta(\mathbf{x} | \mathbf{z})] + \text{KL}(q_\phi(\mathbf{z} | \mathbf{x}) \| p_\theta(\mathbf{z})) + \log p_\theta(\mathbf{x}) \end{aligned} \quad (3)$$

L'objectif d'optimisation de l'auto-encodeur variationnel, comme dans d'autres méthodes variationnelles, est l'evidence lower bound, abrégée en ELBO.

$$\begin{aligned} \text{KL}(q_\phi(\mathbf{z} | \mathbf{x}) \| p_\theta(\mathbf{z} | \mathbf{x})) &= -E_{q_\phi(\mathbf{z} | \mathbf{x})} [\log p_\theta(\mathbf{x} | \mathbf{z})] + \text{KL}(q_\phi(\mathbf{z} | \mathbf{x}) \| p_\theta(\mathbf{z})) + \log p_\theta(\mathbf{x}) \\ &= -\mathcal{L}(\theta, \phi; \mathbf{x}) + \log p_\theta(\mathbf{x}) \end{aligned} \quad (4)$$

Minimiser la divergence de Kullback-Leibler  $\text{KL}(q_\phi(\mathbf{z} | \mathbf{x}) \| p_\theta(\mathbf{z} | \mathbf{x}))$  revient à maximiser l'ELBO  $\mathcal{L}(\theta, \phi; \mathbf{x})$ , L'evidence lower bound (ELBO) est définie par :

$$\mathcal{L}(\theta, \phi; \mathbf{x}) = E_{q_\phi(\mathbf{z} | \mathbf{x})} [\log p_\theta(\mathbf{x} | \mathbf{z})] - \text{KL}(q_\phi(\mathbf{z} | \mathbf{x}) \| p_\theta(\mathbf{z}))$$

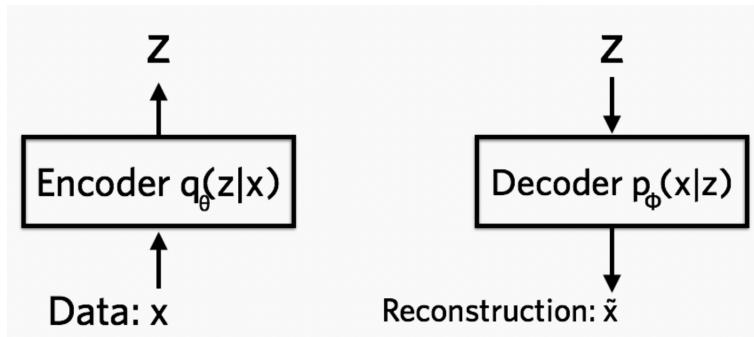


FIGURE 7 – Architecture du VAE

Afin d'effectuer une descente de gradient pour apprendre les réseaux, on utilise un reparametrization trick pour se débarrasser du tirage stochastique de l'encodeur : une variable aléatoire  $z$  suivant une loi normale  $\mathcal{N}(\mu, \sigma)$  peut être obtenue par  $z = \mu + \sigma \odot \epsilon$  avec  $\epsilon \sim \mathcal{N}(0, 1)$ .

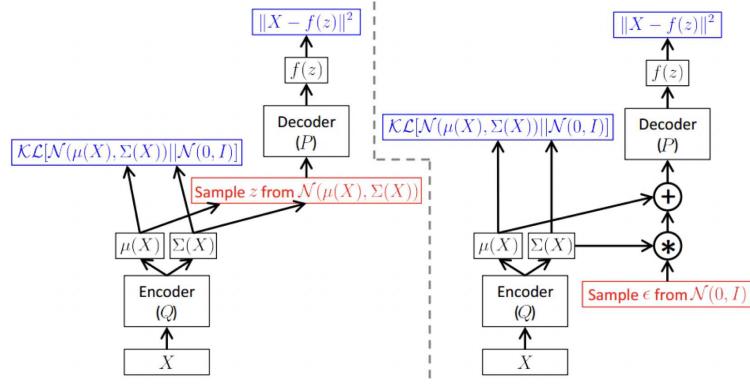
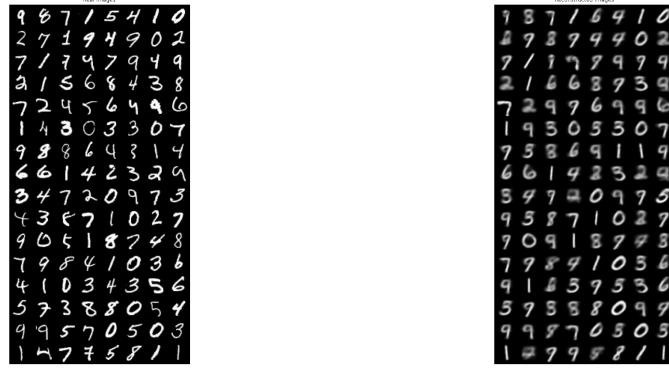


FIGURE 8 – Architecture générale du VAE

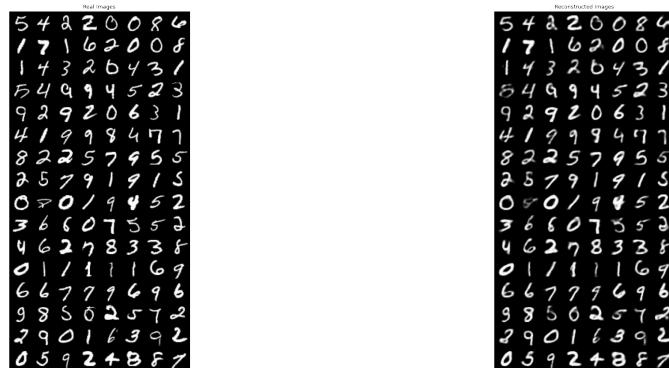
Afin d'expérimenter l'achitecture nous avons utilisé le dataset MNIST avec un batch\_size de 128, un  $\beta_1 = 0.9$ , un  $\text{beta}_2 = 0.999$ , un  $lr = 0.001$ , un  $nb\_epochs = 20$  coupler avec Adam optimizer ainsi que un  $display\_freq = 1000$  afin d'afficher les images générer au fur et à mesure, pour les dimensions latentes nous avons utilisé 2 et 32 avec des couches cachées [256], [256,128], [256,128,64].



(a) Un échantillon d'images réelles

(b) Un échantillon d'images générées

FIGURE 9 – Génération d'images avec une 256 comme dimension pour les couches cachées et 2 pour l'espace latent

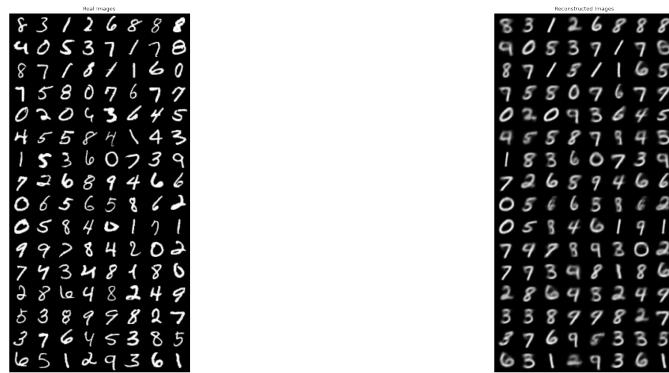


(a) Un échantillon d'images réelles

(b) Un échantillon d'images générées

FIGURE 10 – Génération d'images avec une 256 comme dimension pour les couches cachées et 32 pour l'espace latent

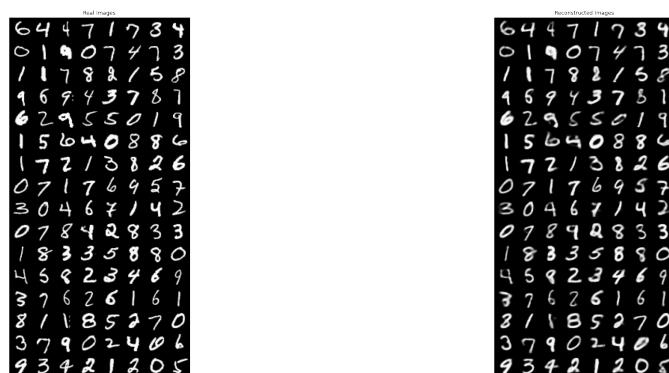
Nous pouvons directement remarquer que pour une dimension latente de 2 les images générées sont un peu bruitées, un des problèmes connus des VAE, par contre pour une dimension latente de 32 les images générées sont quasiment parfaites.



(a) Un échantillon d'images réelles

(b) Un échantillon d'images générées

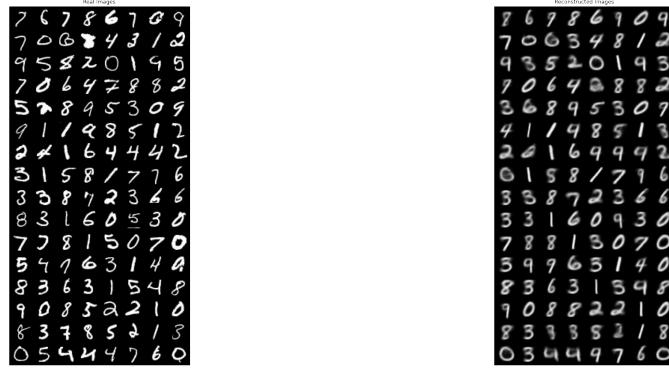
FIGURE 11 – Génération d'images avec [256,128] comme dimension pour les couches cachées et 2 pour l'espace latent



(a) Un échantillon d'images réelles

(b) Un échantillon d'images générées

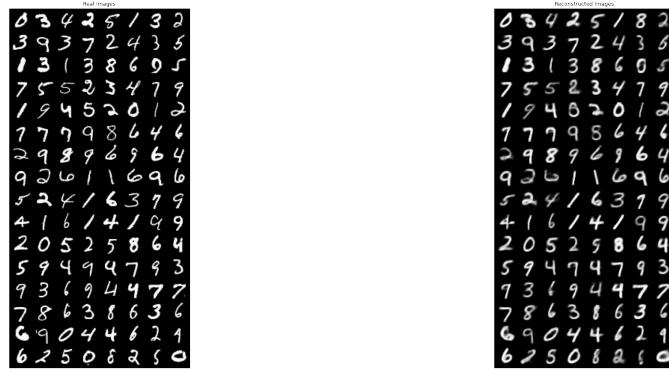
FIGURE 12 – Génération d'images avec [256,128] comme dimension pour les couches cachées et 32 pour l'espace latent



(a) Un échantillon d'images réelles

(b) Un échantillon d'images générées

FIGURE 13 – Génération d'images avec [256,128,64] comme dimension pour les couches cachées et 2 pour l'espace latent



(a) Un échantillon d'images réelles

(b) Un échantillon d'images générées

FIGURE 14 – Génération d'images avec [256,128,64] comme dimension pour les couches cachées et 32 pour l'espace latent

La même remarque s'applique ici, le nombre de couches cachées n'a apriori pas d'influence comme nous pouvons le voir sur les figures, mais pour un espace latent de 32 les images sont quasiement parfaite, nettement meilleur que pour un espace latent de 2, ce qui est complètement compréhensible, 2 comme espace latent est une forte compression.

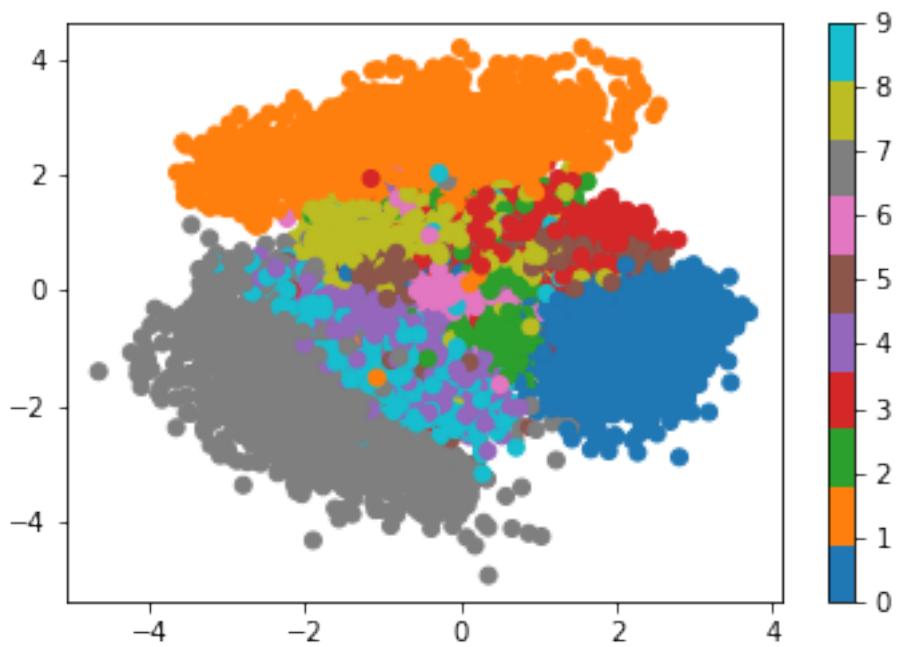


FIGURE 15 – Représentation de l'espace latent

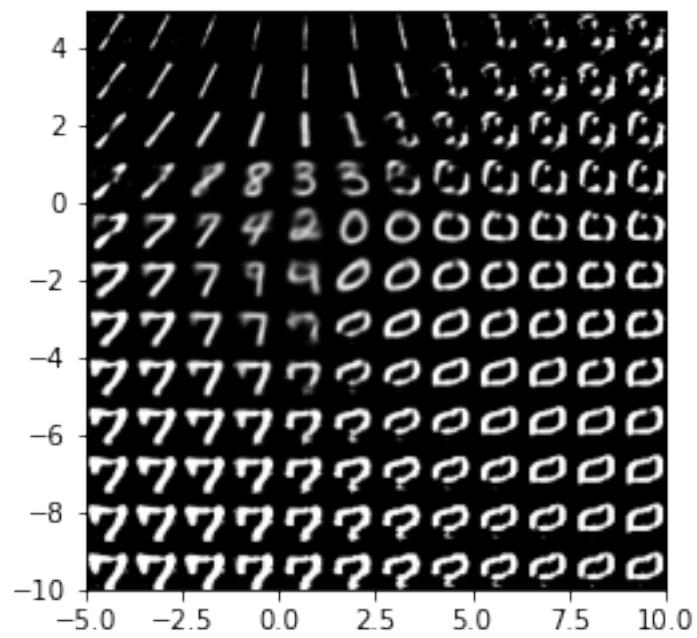


FIGURE 16 – Affichage de l'espace latent

### 3 Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments

Multi-agent DDPG (MADDPG) étend DDPG à un environnement où plusieurs agents se coordonnent pour accomplir des tâches avec uniquement des informations locales. Du point de vue d'un agent, l'environnement est non stationnaire car les politiques des autres agents sont rapidement mises à jour et restent inconnues. MADDPG est un modèle actor-critic repensé particulièrement pour gérer un tel environnement changeant et les interactions entre agents.

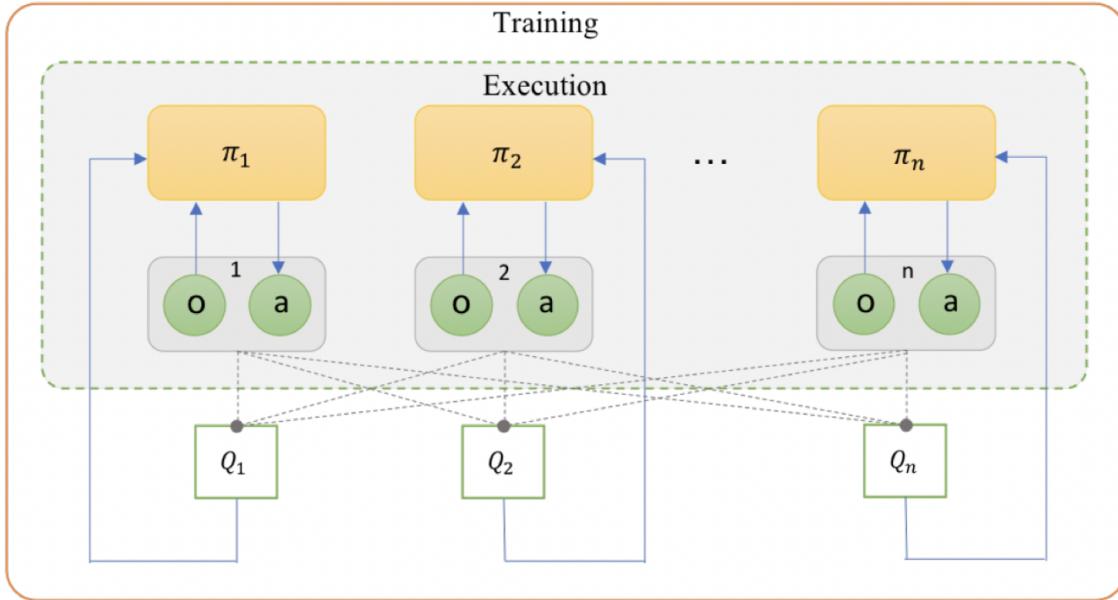


FIGURE 17 – Apprentissage de MADDPG

Considérons un jeu avec  $N$  agents avec des politiques paramétrées par  $\theta = \{\theta_1, \dots, \theta_N\}$  et soit  $\pi = \{\pi_1, \dots, \pi_N\}$  l'ensemble de toutes les politiques, nous pouvons alors écrire le gradient du reward espéré pour l'agent  $i$  comme :

$$\nabla_{\theta_i} J(\theta_i) = E_{s \sim p^\mu, a_i \sim \pi_i} [\nabla_{\theta_i} \log \pi_i(a_i | o_i) Q_i^\pi(\mathbf{x}, a_1, \dots, a_N)] \quad (5)$$

Le critique dans MADDPG apprend une fonction de valeur d'action centralisée  $Q_i^\pi(\mathbf{x}, a_1, \dots, a_N)$  qui prend en entrée les actions de tous les agents  $a_1, \dots, a_N$  en plus de certaines informations d'état  $\mathbf{x}$  avec se composent des observations de tous les agents  $\mathbf{x} = (o_1, \dots, o_N)$ .

Nous pouvons étendre l'idée ci-dessus pour travailler avec des politiques déterministes, si l'on considère maintenant  $N$  politiques continues  $\mu_{\theta_i}$  le gradient peut s'écrire :

$$\nabla_{\theta_i} J(\mu_i) = E_{\mathbf{x}, a \sim \mathcal{D}} \left[ \nabla_{\theta_i} \mu_i(a_i | o_i) \nabla_{a_i} Q_i^\mu(\mathbf{x}, a_1, \dots, a_N) \Big|_{a_i = \mu_i(o_i)} \right] \quad (6)$$

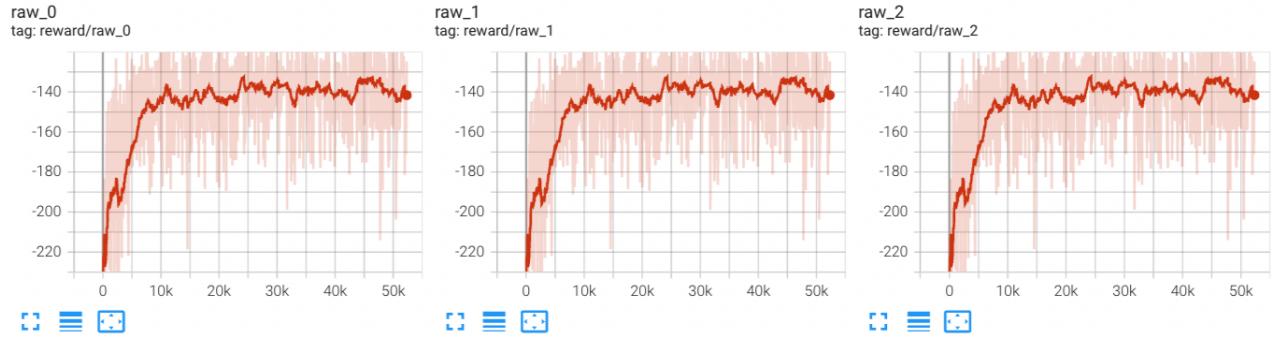
Où  $\mathcal{D}$  est un replay buffer, contenant plusieurs échantillons d'épisodes  $(\mathbf{x}, a_1, \dots, a_N, r_1, \dots, r_N, \mathbf{x}')$ , La fonction action-valeur centralisée  $Q_i^\mu$  est mis à jour comme :

$$\mathcal{L}(\theta_i) = E_{\mathbf{x}, a, r, \mathbf{x}'} \left[ (Q_i^\mu(\mathbf{x}, a_1, \dots, a_N) - y)^2 \right], \quad y = r_i + \gamma Q_i^{\mu'}(\mathbf{x}', a'_1, \dots, a'_N) \Big|_{a'_j = \mu'_j(o_j)} \quad (7)$$

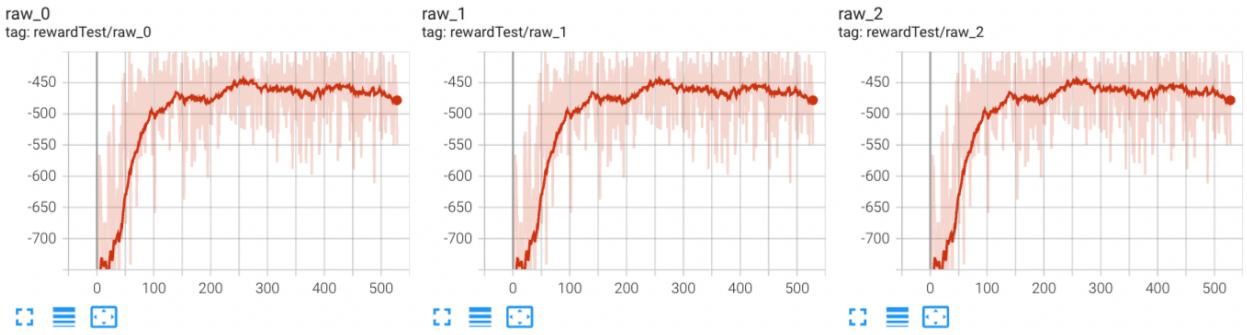
Où  $\mu' = \{\mu_{\theta'_1}, \dots, \mu_{\theta'_N}\}$  est l'ensemble des politiques cibles.

#### 3.1 Cooperative navigation (Simple Spread) :

Nous avons donc tester notre implementation sur Simple spread avec un  $batchsize = 128$ , un buffer de  $capacit = 1000$ ,  $freqOptim = 10$ ,  $startEvents = 10$  le nombre d'évenements purement aléatoires (avec noise uniquement) en début d'apprentissage,  $N = 3$  le nombre d'agent,  $\gamma = 0.95$ ,  $layers = [128]$ ,  $lr = 0.001$  pour tous,  $lrq = 0.01$  pour tous,  $maxLengthTest = 100$ ,  $maxLengthTrain = 25$ ,  $polyakP = 0.9$  et  $polyakQ = 0.9$ , un processus d'exploration de Ornstein-Uhlenbeck avec  $\sigma = 0.2$  réinitialiser à chaque fin d'épisode, et pas de clipping du reward.



(a) Reward en Train



(b) Reward en Test

FIGURE 18 – Reward en Train et en Test sur l’environnement Simple Spread pour chacun des agents

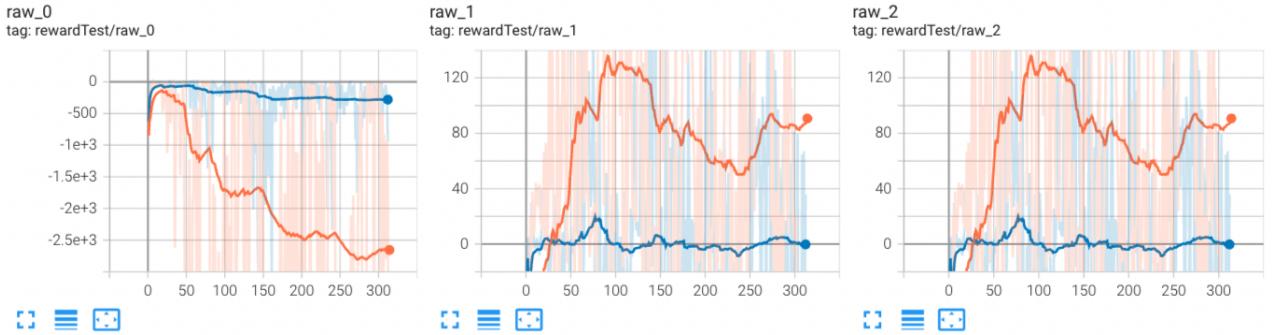
Nous pouvons remarquer la convergence en train vers une valeur en moyenne qui est entre -130 et -140, et -450 en test. L’application de ces méthodes à des environnements compétitifs (adversaire) est également un défi du point de vue de l’optimisation, comme en témoigne l’instabilité.

### 3.2 Physical deception (Simple Adversary) :

Nous avons donc tester notre implementation sur Simple Adversary avec un  $batchsize = 1024$ , un buffer de  $capacit = 10000$ ,  $freqOptim = 10$ ,  $startEvents = 1000$  le nombre d’evenements purement aléatoires (avec noise uniquement) en début d’apprentissage,  $N = 3$  le nombre d’agent,  $\gamma = 0.95$ ,  $layers = [128]$ ,  $lr = 0.01$  pour tous,  $lrq = 0.01$  pour tous,  $maxLengthTest = 100$ ,  $maxLengthTrain = 25$ ,  $polyakP = 0.99$  et  $polyakQ = 0.99$ , un processus d’exploration de Ornstein-Uhlenbeck avec  $\sigma = 0.2$  réinitialiser à chaque fin d’épisode, et pas de clipping du reward.



(a) Reward en Train (en orange avec la configuration de simple spread en blue actuel)



(b) Reward en Test (en orange avec la configuration de simple spread en blue actuel)

FIGURE 19 – Reward en Train et en Test sur l'environnement Simple Adversary pour chacun des agents

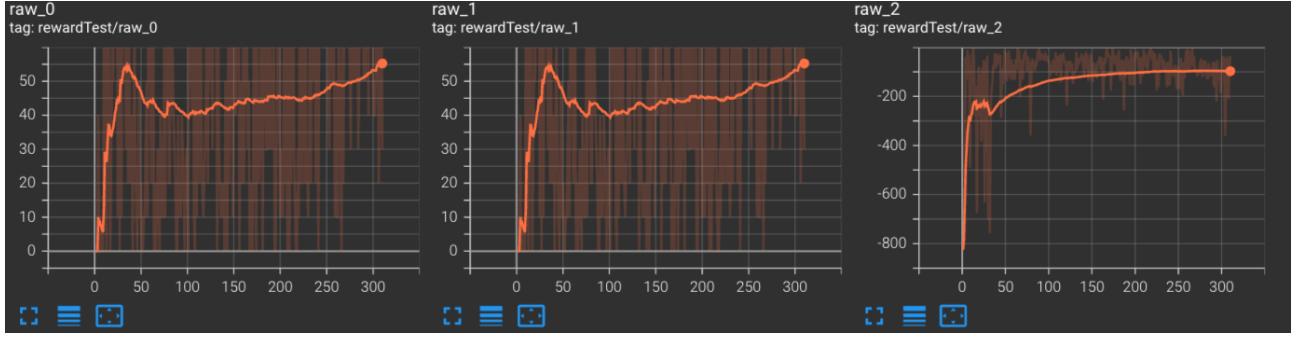
Nous avons pu atteindre une convergence (en blue) avec la configuration citer précédemment, le résultat reste tout de même instable du à l'aspect adversaire des environnements et de la tâche, cette environnement est beaucoup plus difficile à faire converger que le précédent, nous arrivons à atteindre un reward de -10 pour l'adversaire et 4 pour les bons en train, en test nous arrivons à atteindre -50 pour l'adversaire et 0 pour les bons.

### 3.3 Predator Prey (Simple Tag) :

Nous avons donc tester notre implementation sur Simple Tag avec un  $batchsize = 1024$ , un buffer de  $capacit = 1000000$ ,  $freqOptim = 10$ ,  $startEvents = 1000$  le nombre d'évenements purement aléatoires (avec noise uniquement) en début d'apprentissage,  $N = 3$  le nombre d'agent,  $\gamma = 0.95$ ,  $layers = [128]$ ,  $lr = 0.01$  pour tous,  $lrq = 0.01$  pour tous,  $maxLengthTest = 100$ ,  $maxLengthTrain = 25$ ,  $polyakP = 0.99$  et  $polyakQ = 0.99$ , un processus d'exploration de Ornstein-Uhlenbeck avec  $\sigma = 0.2$  réinitialiser à chaque fin d'épisode, et pas de clipping du reward.



(a) Reward en Train



(b) Reward en Test

FIGURE 20 – Reward en Train et en Test sur l'environnement Simple Tag pour chacun des agents

La convergence n'a pas encore été atteinte, mais nous pouvons tout de même remarquer une net tendance d'amélioration, ceci dit nous avons donc un reward moyen de 12 pour les 2 prédateurs en train et -16 pour la proie, 60 en test pour les 2 prédateurs et -100 pour la proie.

### 3.4 Amélioration :

Pour obtenir des politiques robustes aux changements des autres agents (e.g., éviter de sur-apprendre sur les politiques des agents concurrents), MADDPG propose également de considérer des ensemble de politiques par agent : chaque agent possède  $K$  politiques  $\mu_i^{(1)}, \dots, \mu_i^{(K)}$

On vise alors à maximiser pour chaque agent  $i$  :  $E_{k \sim \text{unif}(1, K), s \sim d^\mu, a_i \sim \mu_i^{(k)}} [R_i(s, a_i)]$  Gradient correspondant (avec  $k$  replay buffers par agent) :

$$\nabla_{\theta_i^{(k)}} J(\mu_i) = E_{x, a \sim D_i^{(k)}} \left[ \nabla_{\theta_i^{(k)}} \mu_i^{(k)}(o_i) \nabla_{a_i} Q_i^\mu(x, a_1, \dots, a_N) \Big|_{a_i = \mu_i^{(k)}(o_i)} \right]$$

Nous avons donc tester notre implementation sur Simple Tag avec un  $batchsize = 1024$ , un buffer de  $capacit = 1000000$ ,  $freqOptim = 10$ ,  $startEvents = 1000$  le nombre d'évenements purement aléatoires (avec noise uniquement) en début d'apprentissage,  $N = 3$  le nombre d'agent,  $\gamma = 0.95$ ,  $layers = [128]$ ,  $lr = 0.01$  pour tous,  $lrq = 0.01$  pour tous,  $maxLengthTest = 100$ ,  $maxLengthTrain = 25$ ,  $polyakP = 0.99$  et  $polyakQ = 0.99$ , un processus d'exploration de Ornstein-Uhlenbeck avec  $\sigma = 0.2$  réinitialiser à chaque fin d'épisode, et pas de clipping du reward, et cette fois avec 3 sous-politiques pour chaque agent.



(a) Reward en Train (en bleu avec 3 sous-politique en orange avec une sous-politique)

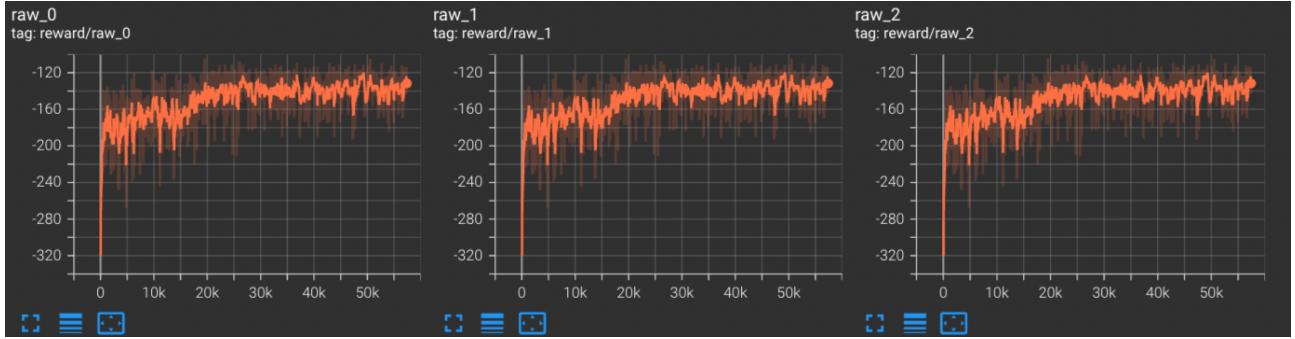


(b) Reward en Test (en bleu avec 3 sous-politique en orange avec une sous-politique)

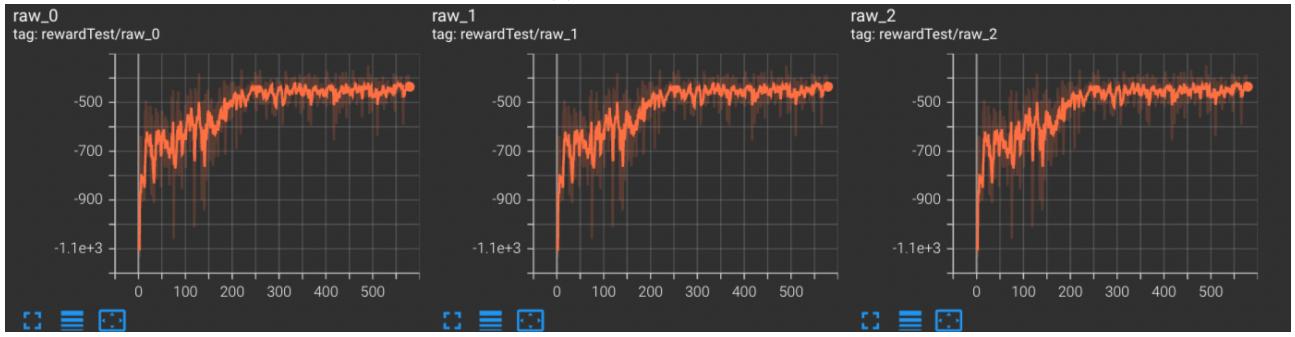
FIGURE 21 – Reward en Train et en Test sur l'environnement Simple Tag pour chacun des agents (en bleu avec 3 sous-politique en orange avec une sous-politique)

Nous remarquons donc qu'avec l'amélioration la réduction du sur-apprentissage, mais le reward est presque similaire dans la tâche de simple tag.

Nous avons donc tester notre implémentation sur Simple spread avec un  $batchsize = 128$ , un buffer de  $capacit = 1000$ ,  $freqOptim = 10$ ,  $startEvents = 10$  le nombre d'événements purement aléatoires (avec noise uniquement) en début d'apprentissage,  $N = 3$  le nombre d'agent,  $\gamma = 0.95$ ,  $layers = [128]$ ,  $lr = 0.001$  pour tous,  $lrq = 0.01$  pour tous,  $maxLengthTest = 100$ ,  $maxLengthTrain = 25$ ,  $polyakP = 0.9$  et  $polyakQ = 0.9$ , un processus d'exploration de Ornstein-Uhlenbeck avec  $\sigma = 0.2$  réinitialiser à chaque fin d'épisode, et pas de clipping du reward, en rajoutant 3 sous-politique pour chaque agent.



(a) Reward en Train



(b) Reward en Test

FIGURE 22 – Reward en Train et en Test sur l'environnement Simple Spread

Nous avons obtenu un résultat plus stable et meilleur que précédemment approchant donc les -120 reward pour les 3 agents en train et -400 en test, l'amélioration est donc bénéfique pour la tâche de multi agent.

## 4 Imitation Learning

### 4.1 Behavioral Cloning :

Il s'agit de maximiser la probabilité de choisir, pour chaque exemple  $i$  de l'ensemble expert  $\mathcal{E}$ , l'action  $a_i$  associée à l'état  $s_i$  selon la politique de l'agent  $\pi_\theta$  :

$$\max_{\theta} \sum_{(a_i, s_i) \in \mathcal{E}} \log \pi_\theta(a_i | s_i) \quad (8)$$

Nous avons donc testé notre implémentation sur LunarLander avec un réseau de neurones à 2 couches cachées [64,32] avec activation Tanh, et un softmax à la fin, une fréquence d'optimisation de 100 événements, un batchsize de 298 soit la taille du dataset expert,  $lr = 0.003$  et un affichage sans moyenne et sans maxLength pour le train et le test.

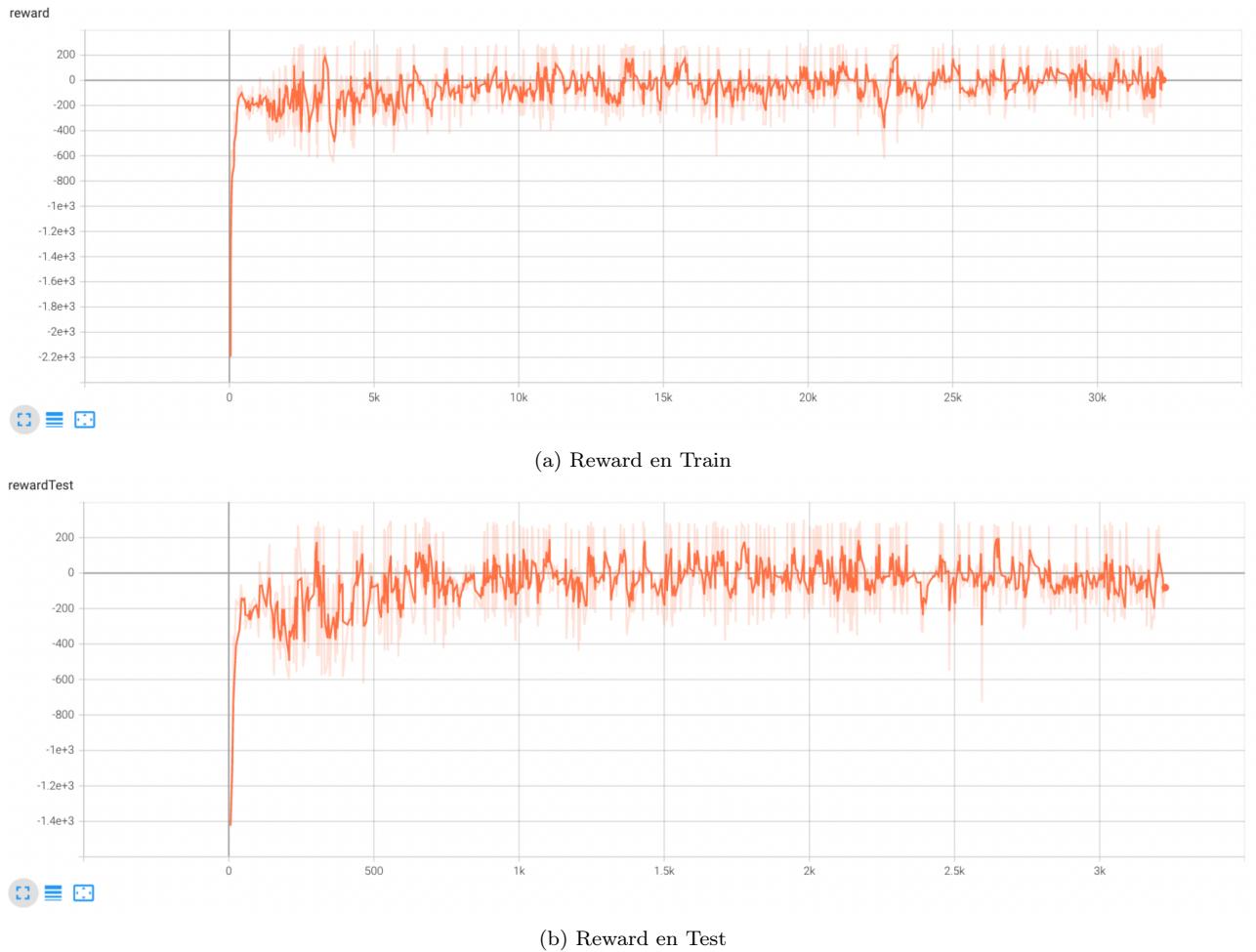


FIGURE 23 – Reward en Train et en Test sur l'environnement LunarLander avec Behavioral Cloning

On note donc un reward autour de 0 en train et en test, avec Behavioral Cloning on arrive à peine à atteindre des rewards acceptables, et il n'arrive pas à bien apprendre sur les données expertes.

### 4.2 Generative Adversarial Imitation Learning :

GAIL envisage d'apprendre une politique à partir d'un exemple de comportement d'expert, sans interaction avec l'expert ni accès à un signal de renforcement. GAIL extrait une politique depuis les données comme si elle était obtenue par reinforcement learning suivant un apprentissage par renforcement inverse. Il définit l'Imitation Learning comme un problème adverse où l'on cherche à générer des trajectoires indiscernables des trajectoires de l'expert selon un discriminateur appris simultanément.

Nous allons apprendre la politique en utilisant l'algorithme PPO avec :

$$\hat{E}_{\tau_i} [\nabla_{\theta} \log \pi_{\theta}(a | s) Q(s, a)] - \lambda \nabla_{\theta} H(\pi_{\theta}) \quad (9)$$

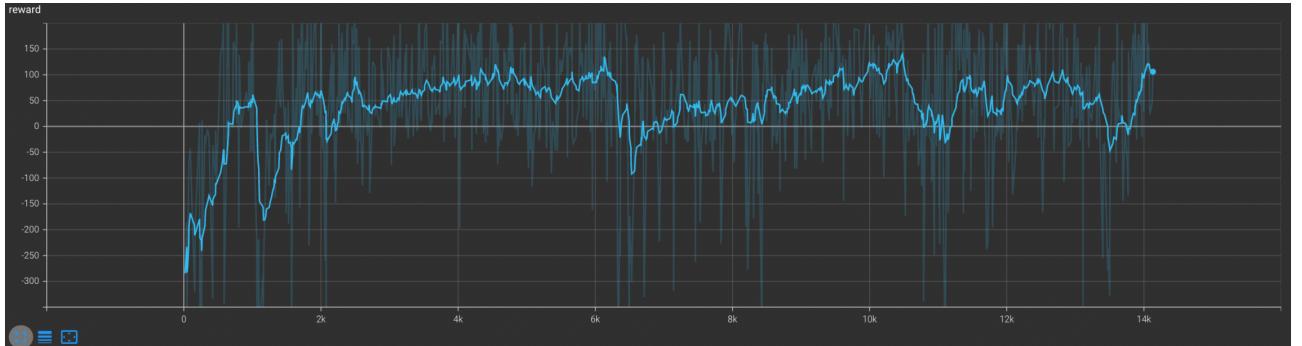
where  $Q(\bar{s}, \bar{a}) = \hat{E}_{\tau_i} [\log(D_{w_{i+1}}(s, a)) | s_0 = \bar{s}, a_0 = \bar{a}]$

Et le descriminateur :

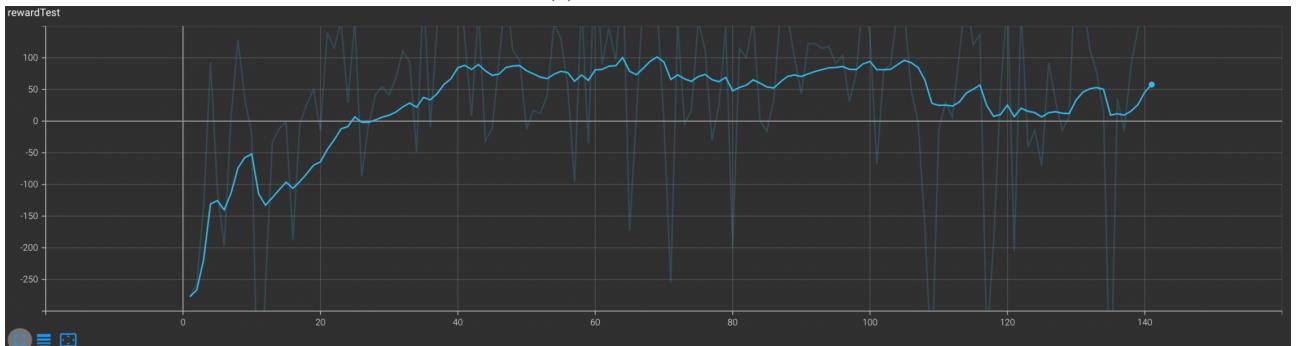
$$\hat{E}_{\tau_i} [\nabla_w \log (D_w(s, a))] + \hat{E}_{\tau_E} [\nabla_w \log (1 - D_w(s, a))] \quad (10)$$

Avec les trajectoires de l'expert  $\tau_E \sim \pi_E$ , la politique intiale et les paramètres du descriminateur  $\theta_0, w_0$ , ainsi que  $R_t = \frac{1}{T-t} \sum_{t'=t}^T r_{t'} = \frac{1}{T-t} \sum_{t'=t}^T \log D_{\omega}(s_{t'}, a_{t'})$ . À noter que nous avons choisis  $r_t = -\log(1 - D(s_t, a_t))$ , et un retour cumulé discounté.

Nous nous intéressons au problème LunarLander (version discrète), dont des décisions expertes sont disponibles (fichier expert.pkl), pour cela nous avons utilisé les paramètres suivant :  $lr\_pi = 0.0003$  le learning rate de la politique,  $lr\_q = 0.0003$  le learning rate pour la critic,  $lr\_disc = 0.0003$  le learning rate du descrimintor, [100, 100] dimensions des couches cachées du discriminator, l'actor ainsi que la critic,  $eps\_clip = 0.2$  l'epsilon pour le clipping,  $\gamma = 0.99$ ,  $\lambda = 0.95$ ,  $numberOptimAC = 10$  nombre de step d'optimisation de l'actor et de la critic,  $numberOptimDiscriminator = 10$  le nombre de step d'optimisation du discriminator,  $batchsize = 256$  mais moduler à la taille du buffer le max entre la taille du buffer et 256,  $sigma\_noise = 0.2$  ajoutés à toutes les dimensions des couples (s,a),  $reg\_entropy = 0.001$  le  $\lambda$  pour l'entropy,  $maxReward = 100$  pour le clipping de  $\log D_{\omega}(s_t, a_t)$ ,  $maxLength$  de 500 en train et en test, une mise à jour tout les fin de trajectoire. De plus nous avons appliqué un weight decay sur l'ensemble des paramètres ainsi que un clipping de 10 sur l'actor, le critic et le discriminator pour une meilleure stabilité.



(a) Reward en Train



(b) Reward en Test

FIGURE 24 – Reward en Train et en Test sur l'environnement LunarLunder avec GAIL

Nous pouvons dire que malgré une instabilité qui est due à l'aspect adversaire de la tâche, néanmoins nous arrivons à atteindre un reward avoisinant les 300, beaucoup intéressant que pour Behavioral Cloning, nous avons de plus vu le besoin de stabilisé l'apprentissage. De plus le choix des paramètres semble crucial pour l'apprentissage, ainsi que avoir le même nombre d'exemples pour les données expertes et les données générer.

## 5 Automatic Curriculum RL

### 5.1 DQN avec buts :

Dans cette version du DQN, nous allons mettre en œuvre une version de DQN considérant des buts (Universal Value Function Approximators). Il s'agit alors de considérer une fonction  $Q : \mathcal{A} \times \mathcal{S} \times \mathcal{G} \rightarrow R$ , où  $\mathcal{A}$  est l'ensemble des actions,  $\mathcal{S}$  l'ensemble des états et  $\mathcal{G}$  l'ensemble des buts à considérer. Une manière d'étendre simplement le code de DQN à ce cadre est de se ramener à  $Q_\phi : \mathcal{A} \times \mathcal{S}' \rightarrow R$  en considérant  $\mathcal{S}'$  un ensemble d'états-but élément correspond à la concaténation d'un état de  $\mathcal{S}$  avec un but de  $\mathcal{G}$ .

Nous allons considéré dans un premier temps le fichier de carte *plan2Multi.txt*. Ce fichier possède des buts proches de l'agent de départ, et tout au long du chemin menant aux buts les plus éloignés, ce qui permet d'avoir des récompenses pour progresser.

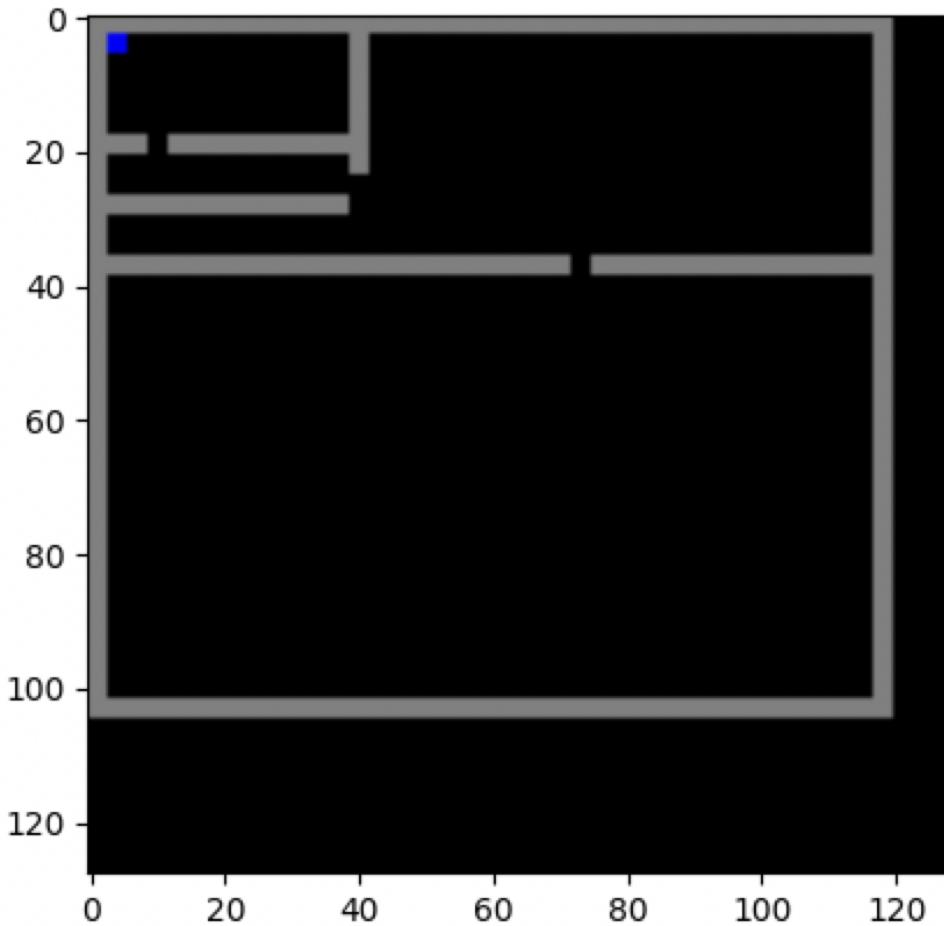
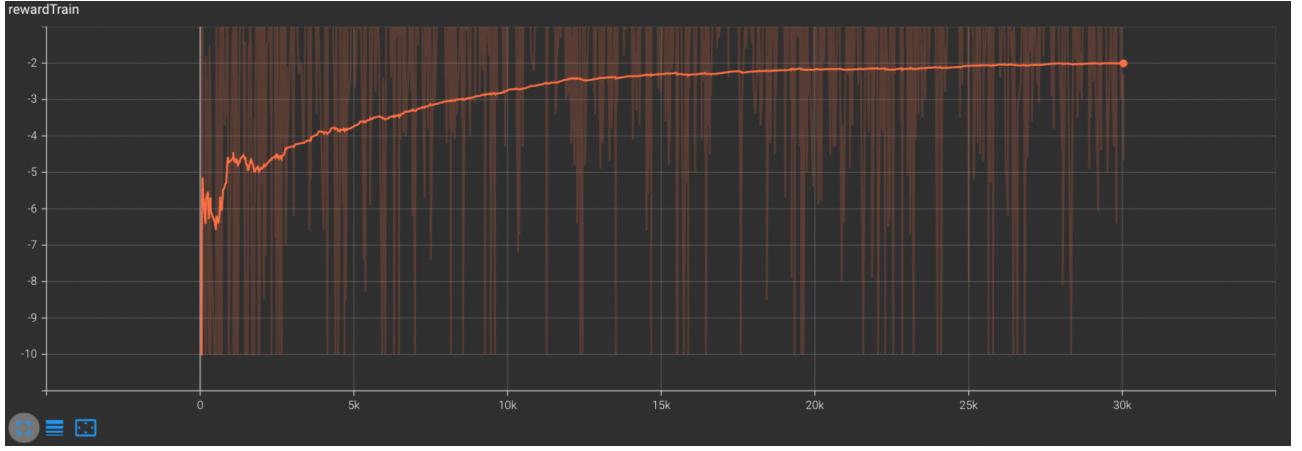
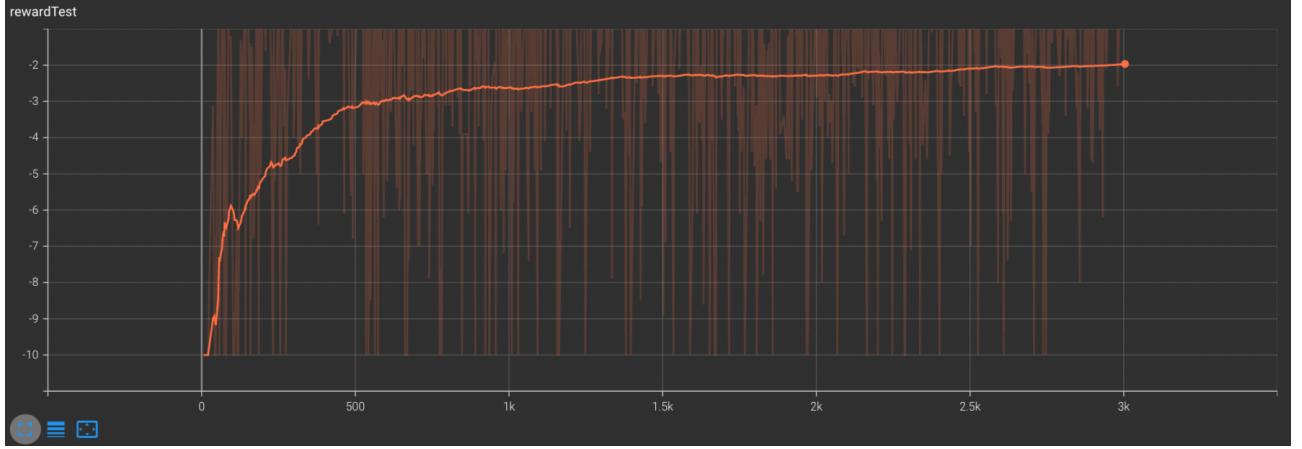


FIGURE 25 – Plan2Multi

Nous avons donc testé cette version du DQN avec le Plan2Multi, en utilisant les paramètres suivant  $freqOptim = 10$  la fréquence d'optimisation,  $maxLengthTest = 100$ ,  $maxLengthTrain = 100$ ,  $lr = 0.001$ ,  $\gamma = 0.99$ ,  $batch\_size = 1000$ ,  $\epsilon = 0.2$  pour l'exploration  $\epsilon\_greedy$ ,  $capacity = 1E6$ , avec un réseau à deux couches cachées de 200 neurones chacune et activations tanh.



(a) Reward en Train



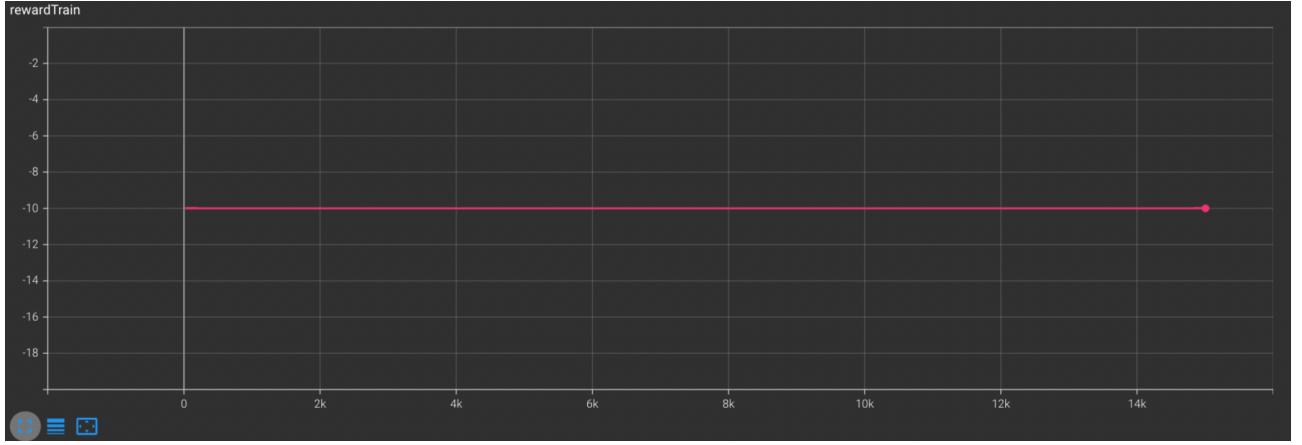
(b) Reward en Test

FIGURE 26 – Reward en Train et en Test sur le Plan2Multi en utilisant DQN avec goal

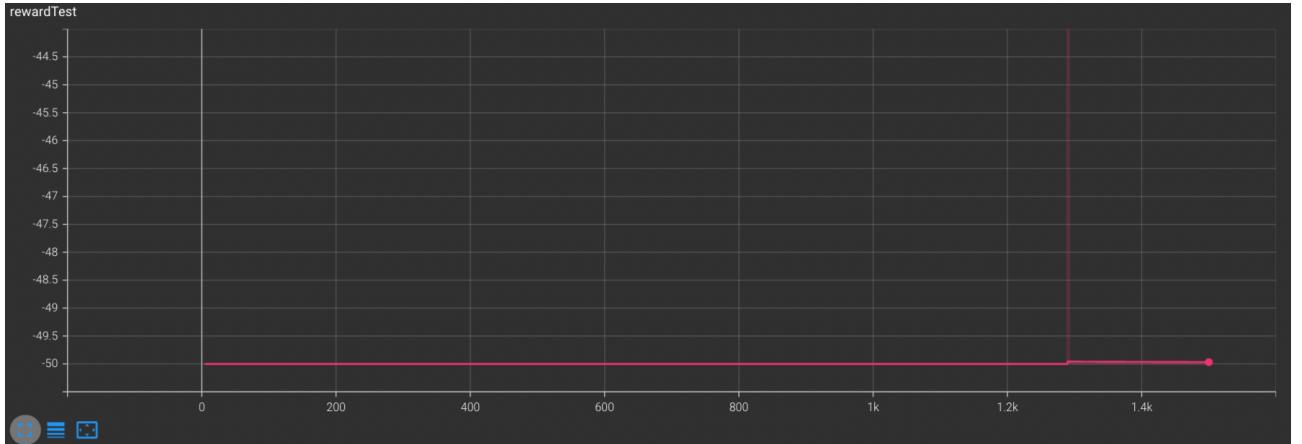
Nous observons un bon apprentissage qui mène à un agent quasi-parfait pour atteindre chacun des buts, y compris le plus éloigné, nous arrivons parfaitement à apprendre ce plan. Un reward approchant le -2 en train et en test, nous remarquons tout de même les oscillations qui sont dus à la probabilité d'exploration qui est de 0.2.

Les choses se compliquent lorsque l'environnement ne fournit pas de buts tout au long du chemin pour découvrir les objectifs les plus éloignés. On considère maintenant le fichier plan2.txt, qui charge la même carte que plan2Multi.txt mais avec un seul but visé, celui le plus difficile à atteindre pour l'agent (en cas [33,38]).

En utilisant les paramètres suivant  $freqOptim = 10$  la fréquence d'optimisation,  $maxLengthTest = 500$ ,  $maxLengthTrain = 100$ ,  $lr = 0.001$ ,  $\gamma = 0.99$ ,  $batch\_size = 1000$ ,  $\epsilon = 0.2$  pour l'exploration  $\epsilon\_greedy$ ,  $capacity = 1E6$ , avec un réseau à deux couches cachées de 200 neurones chacune et activations tanh.



(a) Reward en Train



(b) Reward en Test

FIGURE 27 – Reward en Train et en Test sur le Plan2 en utilisant DQN avec goal

Dans ce cas, une progression nulle, aucune amélioration, DQN est très limité dans ce cas, il peine à apprendre.

## 5.2 Hindsight Experience Replay :

L'idée derrière Hindsight Experience Replay (HER) après avoir expérimenté quelque épisode  $s_0, s_1, \dots, s_T$ , il s'agit d'ajouter dans le Replay Buffer en plus des transitions de chaque épisode associées au but de l'environnement visé par l'épisode, les mêmes transitions mais cette fois associées à un état atteint dans la trajectoire de l'agent (généralement le dernier état rencontré par l'agent dans l'épisode). Cet état but ayant été atteint dans l'épisode (puisque'il en fait partie), cela permet d'effectuer les mises à jour du réseau  $Q_\phi$  avec des retours non nuls, la transition de l'épisode vers cet état étant alors associée à une récompense positive (+1). L'idée est d'exploiter les connaissances sur des sous-but déjà atteints pour généraliser aux buts réels de l'environnement non encore rencontrés, avec  $G$  ne contenant que l'état final de chaque épisode, une récompense associée de +1 lorsque la transition mène à cet état de  $G$  et toutes les autres récompenses à -0.1.

Nous avons donc testé HER avec le Plan2, en utilisant les paramètres suivant  $freqOptim = 10$  la fréquence d'optimisation,  $maxLengthTest = 100$ ,  $maxLengthTrain = 100$ ,  $lr = 0.001$ ,  $\gamma = 0.99$ ,  $batch\_size = 1000$ ,  $\epsilon = 0.2$  pour l'exploration  $\epsilon\_greedy$ ,  $capacity = 1E6$ , avec un réseau à deux couches cachées de 200 neurones chacune et activations tanh.

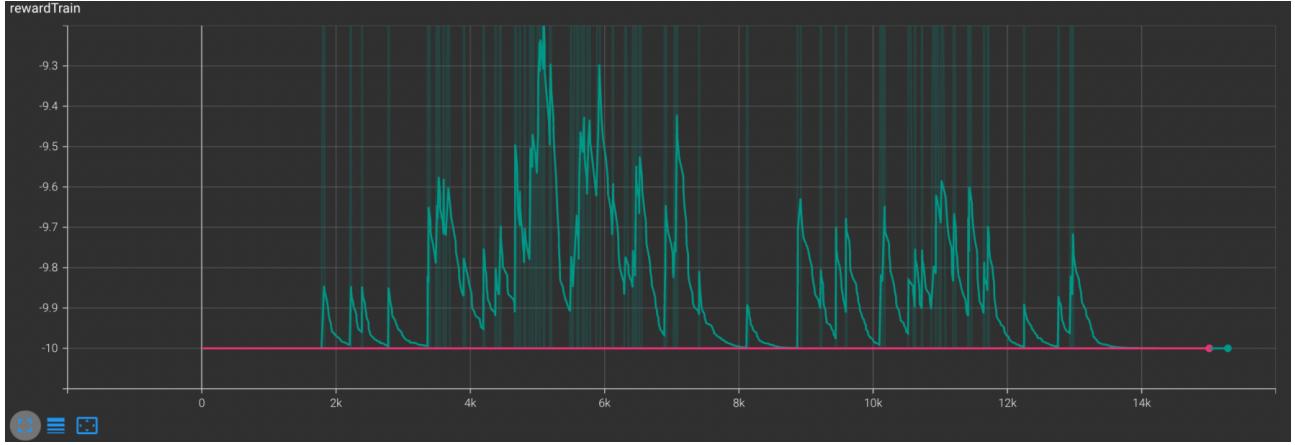


FIGURE 28 – Reward en Train et en Test sur le Plan2 en utilisant HER (en vert HER en rose DQN)

Nous avons réussi à apprendre avec le Plan2, l'instabilité et lier certainement au fait du choix de l'exploration, dans l'ensemble un résultat acceptable vu la tâche. Nous remarquons aussi la différence avec le DQN goal classique qui reste uniforme alors que le HER arrive à atteindre des rewards.

## 6 Modèles de flux (Flow-based)

Nous introduisons les normalizing flows, un type de méthode qui combine les avantages des modèles autorégressifs et auto-encodeurs variationnels, permettant à la fois l'apprentissage des caractéristiques et l'estimation de la vraisemblance.

Changement de variables :  $Z$  et  $X$  sont des variables aléatoires qui sont liées par une application  $f : R^n \rightarrow R^n$  tel que  $X = f(Z)$  et  $Z = f^{-1}(X)$ .

$$p_X(x) = p_Z(f^{-1}(x)) \left| \det \left( \frac{\partial f^{-1}(x)}{\partial x} \right) \right| \quad (11)$$

En utilisant le changement de variables, la vraisemblance marginale  $p(x)$  est donnée par :

$$p_X(x; \theta) = p_Z(f_\theta^{-1}(x)) \left| \det \left( \frac{\partial f_\theta^{-1}(x)}{\partial x} \right) \right| \quad (12)$$

Le nom normalizing flow peut être interprété comme suit :

- "Normaliser" signifie que le changement de variables donne une densité normalisée après application d'une transformation inversible.
- "Flux" signifie que les transformations inversibles peuvent être composées les unes avec les autres pour créer des transformations inversibles plus complexes.

$$\begin{aligned} \log p(\mathbf{x}) &= \log p(\mathbf{z}_K) \\ &= \log p_0(\mathbf{z}_0) - \sum_{i=0}^{K-1} \log \left| \det \frac{df_{i+1}}{d\mathbf{z}_i} \right| \\ &= \log p_0(f_1^{-1} \circ \dots \circ f_K^{-1}(\mathbf{z}_K)) - \sum_{i=0}^{K-1} \log \left| \det \frac{df_{i+1}}{d\mathbf{z}_i} \right| \end{aligned} \quad (13)$$

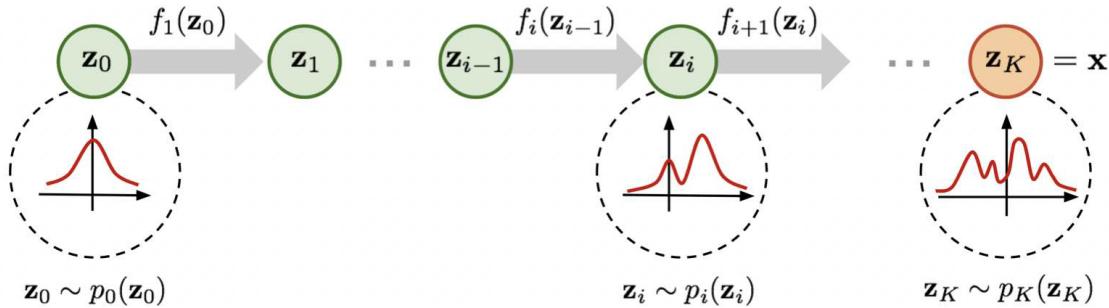


FIGURE 29 – Flow (source Lil'Log)

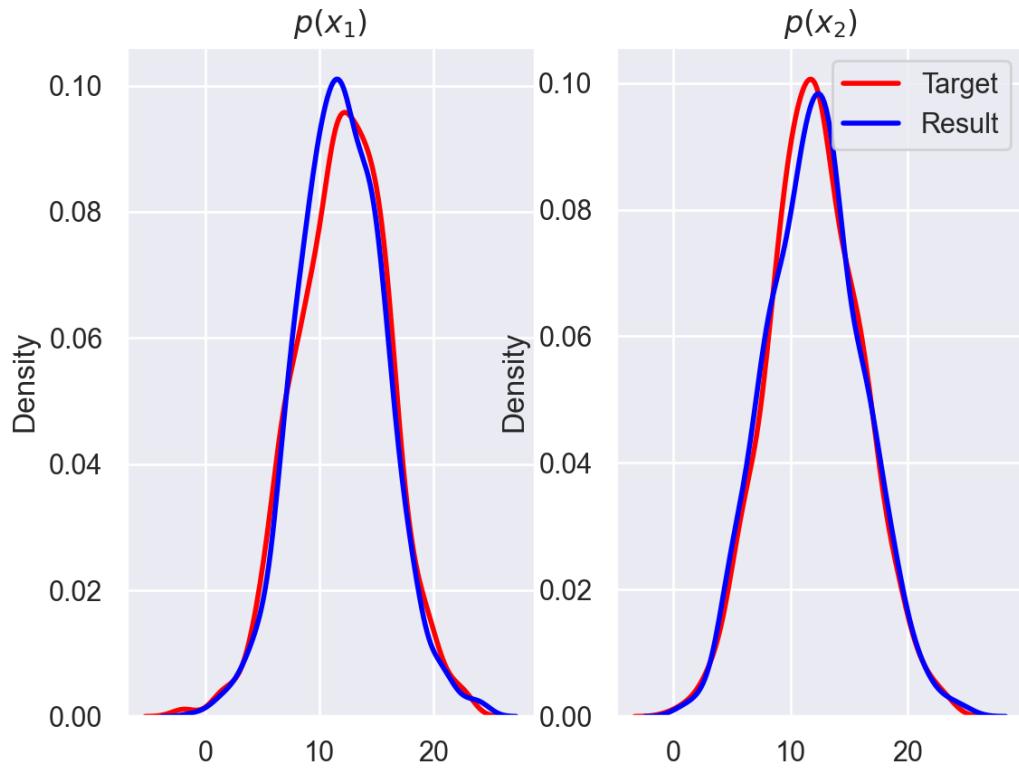
La méthode check est utilisée la simple définition des fonctions inverses. Si  $f$  est une bijection d'un ensemble  $X$  vers un ensemble  $Y$ , cela veut dire (par définition des bijections) que tout élément  $y$  de  $Y$  possède un antécédent et un seul par  $f$ . On peut donc définir une application  $g$  allant de  $Y$  vers  $X$ , qui à  $y$  associe son unique antécédent, c'est-à-dire que :  $g \circ f = \text{Id}_X$  et  $f \circ g = \text{Id}_Y$ .

### 6.1 Transformation affine :

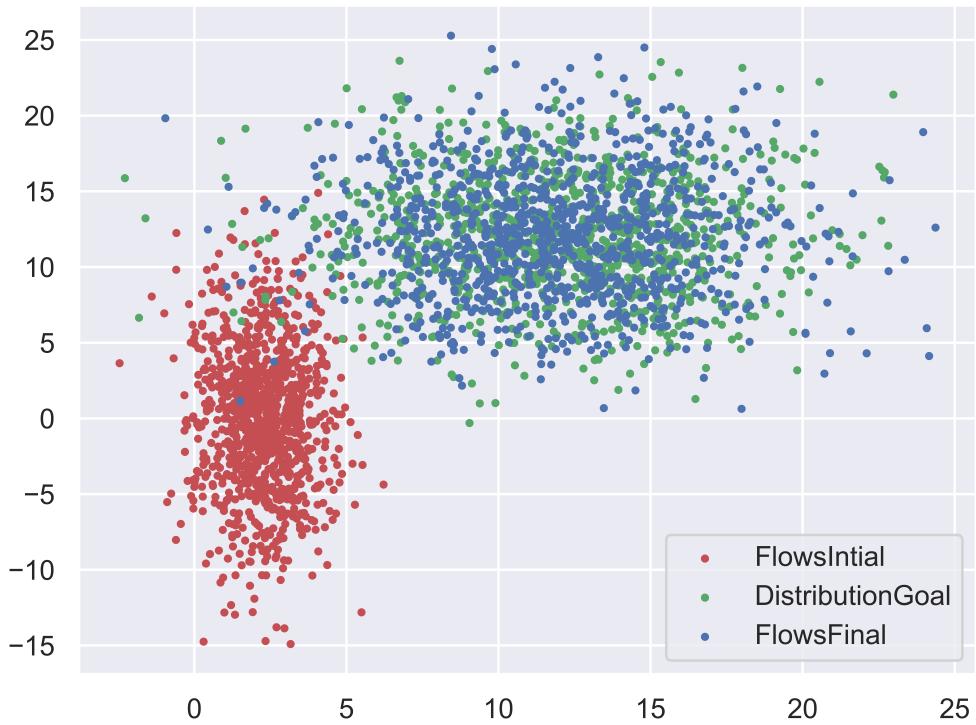
La première transformation est simple : il s'agit d'une transformation affine, permettant de changer la variance et la moyenne d'une variable aléatoire.

Nous avons donc testé le Model avec une TransformationAffine seul et 4 TransformationsAffine successive avec les paramètres suivants :  $dim = 2$ ,  $batchsize = 64$ ,  $lr = 0.001$  avec comme loss la NLL  $\mathcal{L}(\mathcal{D}) = -\frac{1}{|\mathcal{D}|} \sum_{\mathbf{x} \in \mathcal{D}} \log p(\mathbf{x})$

$$f(\mathbf{y}; \mathbf{s}, \mathbf{t}) = \mathbf{y} \odot \exp(\mathbf{s}) + \mathbf{t} \quad (14)$$

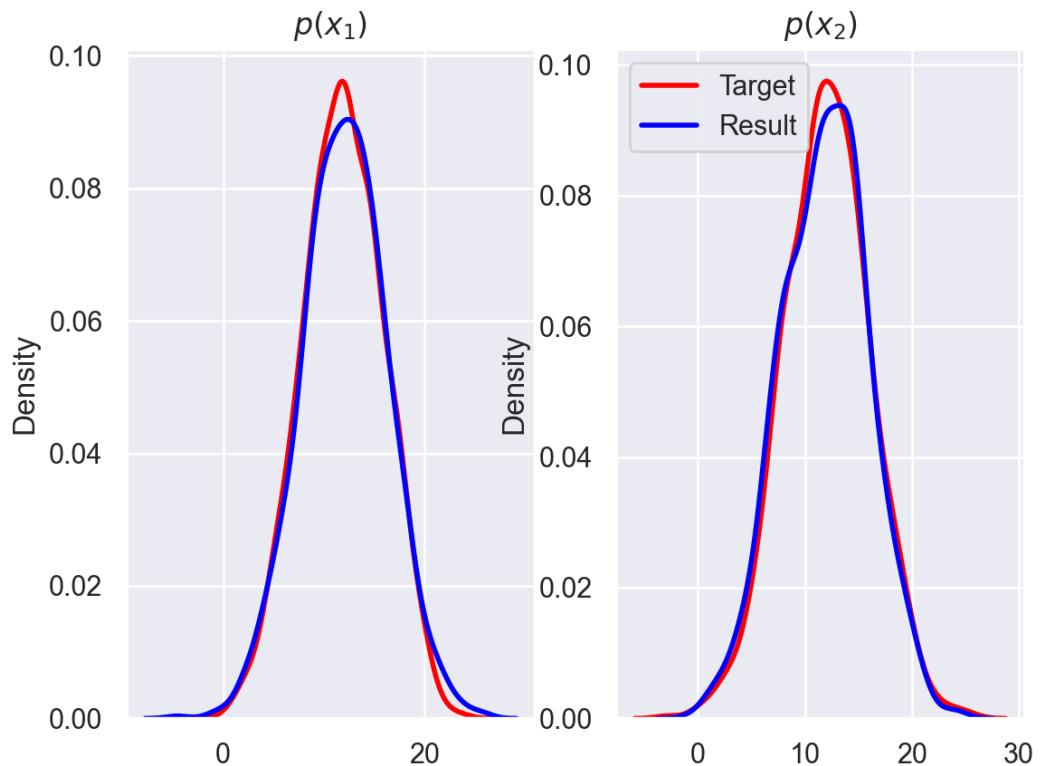


(a) Comparaison des distributions (en rouge  $\mathcal{N}(12, 4)$  en bleu le résultat après le flow)

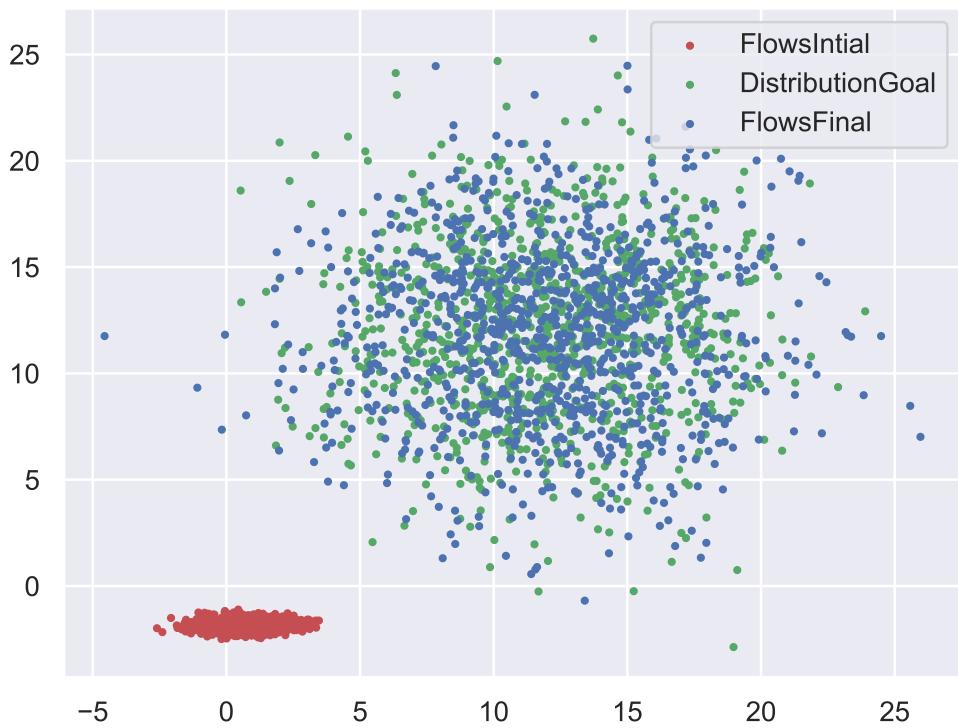


(b) Comparaison des exemples générés (en rouge résultat Flow à l'initial, en vert  $\mathcal{N}(12, 4)$  et en bleu le résultat obtenu)

FIGURE 30 – Comparaison de la génération avec un model Transformation Affine seul



(a) Comparaison des distributions (en rouge  $\mathcal{N}(12, 4)$  en bleu le résultat après le flow)



(b) Comparaison des exemples générés (en rouge résultat Flow à l'initial, en vert  $\mathcal{N}(12, 4)$  et en bleu le résultat obtenu)

FIGURE 31 – Comparaison de la génération avec un model 4 Transformation Affine

Les résultats obtenus sont parfaitement concordants à une normal dont nous avons choisi les paramètres, nous pouvons remarquer sans surprise qu'à l'initiale (en rouge) la distribution ne ressemble à rien et au fil des itérations elle se colle de plus en plus.

## 6.2 Glow :

Les transformations affines ne permettent pas de faire grand chose ; pour apprendre des distributions de probabilité plus complexes, nous allons implémenter le modèle Glow qui repose sur trois opérateurs.

Description	Function	Reverse Function	Log-determinant
Actnorm. See Section 3.1.	$\forall i, j : \mathbf{y}_{i,j} = \mathbf{s} \odot \mathbf{x}_{i,j} + \mathbf{b}$	$\forall i, j : \mathbf{x}_{i,j} = (\mathbf{y}_{i,j} - \mathbf{b})/\mathbf{s}$	$h \cdot w \cdot \text{sum}(\log  \mathbf{s} )$
Invertible $1 \times 1$ convolution. $\mathbf{W} : [c \times c]$ . See Section 3.2.	$\forall i, j : \mathbf{y}_{i,j} = \mathbf{W}\mathbf{x}_{i,j}$	$\forall i, j : \mathbf{x}_{i,j} = \mathbf{W}^{-1}\mathbf{y}_{i,j}$	$h \cdot w \cdot \log  \det(\mathbf{W}) $ or $h \cdot w \cdot \text{sum}(\log  \mathbf{s} )$ (see eq. (10))
Affine coupling layer. See Section 3.3 and (Dinh et al., 2014)	$\mathbf{x}_a, \mathbf{x}_b = \text{split}(\mathbf{x})$ $(\log \mathbf{s}, \mathbf{t}) = \text{NN}(\mathbf{x}_b)$ $\mathbf{s} = \exp(\log \mathbf{s})$ $\mathbf{y}_a = \mathbf{s} \odot \mathbf{x}_a + \mathbf{t}$ $\mathbf{y}_b = \mathbf{x}_b$ $\mathbf{y} = \text{concat}(\mathbf{y}_a, \mathbf{y}_b)$	$\mathbf{y}_a, \mathbf{y}_b = \text{split}(\mathbf{y})$ $(\log \mathbf{s}, \mathbf{t}) = \text{NN}(\mathbf{y}_b)$ $\mathbf{s} = \exp(\log \mathbf{s})$ $\mathbf{x}_a = (\mathbf{y}_a - \mathbf{t})/\mathbf{s}$ $\mathbf{x}_b = \mathbf{y}_b$ $\mathbf{x} = \text{concat}(\mathbf{x}_a, \mathbf{x}_b)$	$\text{sum}(\log( \mathbf{s} ))$

FIGURE 32 – sous-étapes dans Glow

Nous avons testé notre modèle avec les paramètres :  $batchsize = 128$ ,  $lr = 1e-5$ ,  $wd = 1e-2$  pour le weight decay,  $hidden\_dim = 32$ , une loi normale centrée réduite comme prior.

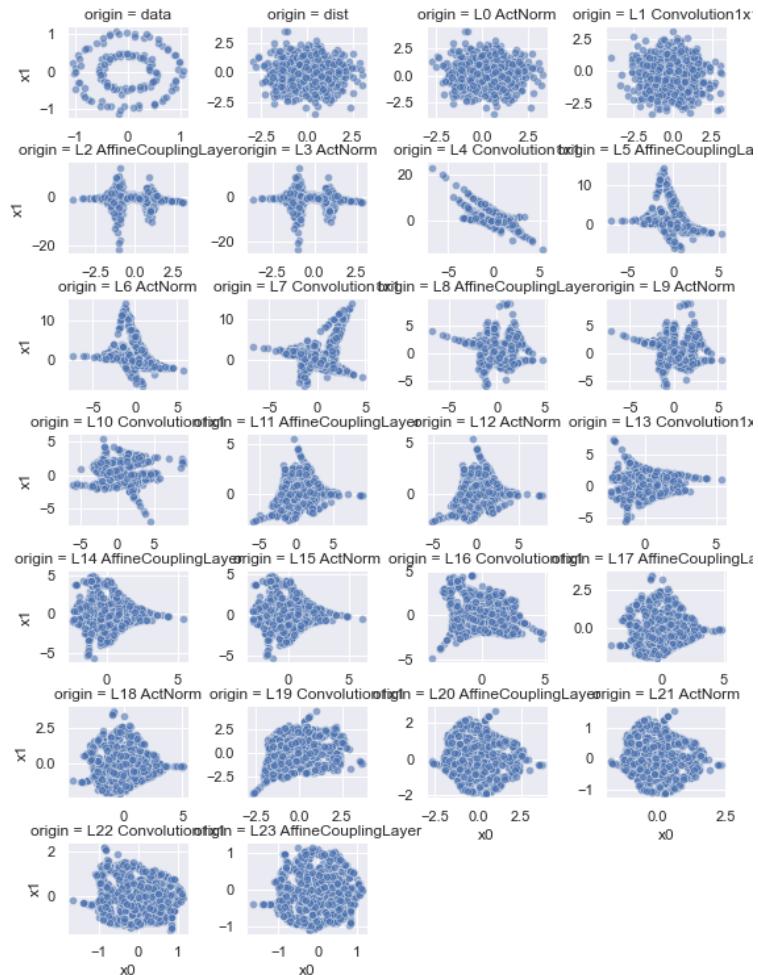


FIGURE 33 – L'enchainement du Flow

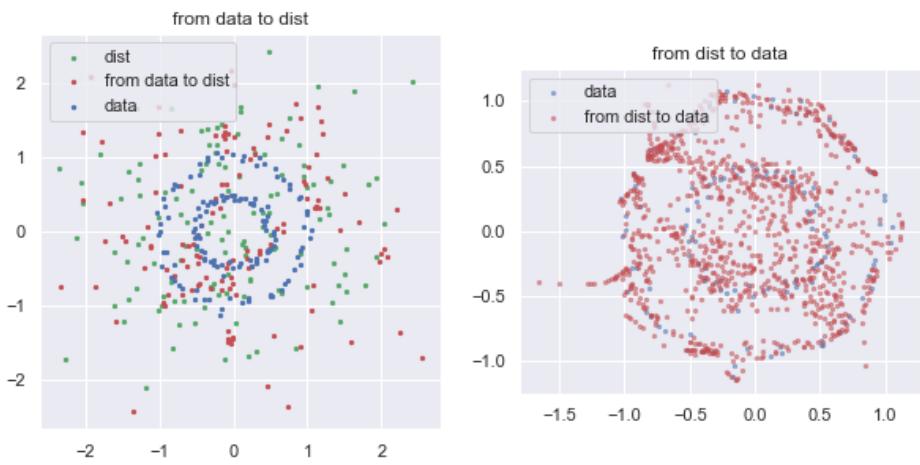


FIGURE 34 – Comparaison des données générées et les données réelles

Les données générées tendent à ressembler à notre target, à savoir les données cercles (figure 34 à droite), l'apprentissage n'est pas parfait mais nous remarquons les données qui se colle sur les contours.