

---

**Apprentissage par renforcement**

**Rendu de TP n°2**

**TPS 9 à 14**

---

## Table des matières

<b>1 TP9 : GAN</b>	<b>3</b>
1.1 Présentation de l'architecture GAN . . . . .	3
1.2 Expérimentations . . . . .	4
1.2.1 GAN . . . . .	4
1.2.2 DCGAN sur celebA . . . . .	6
1.2.3 DCGAN sur MNIST : . . . . .	7
<b>2 TP10 : Autoencodeur variationnel</b>	<b>9</b>
2.1 Principe de fonctionnement et objectif d'un VAE . . . . .	9
2.2 Expérimentations . . . . .	10
2.2.1 Dimension de l'espace latent = 1 . . . . .	10
2.2.2 Dimension de l'espace latent = 2 . . . . .	11
2.2.3 Dimension de l'espace latent = 15 . . . . .	11
2.3 Qualité de reconstruction de l'algorithme . . . . .	13
<b>3 TP11 : MADDPG</b>	<b>14</b>
3.1 Coopérative navigation . . . . .	15
3.2 Physical deception . . . . .	15
3.3 Predator-prey . . . . .	17
<b>4 TP12 : Imitation Learning</b>	<b>19</b>
4.1 Behavioral Cloning . . . . .	19
4.2 GAIL . . . . .	19
<b>5 TP13 : Implicit Curriculum RL</b>	<b>22</b>
5.1 Goal-base DQN . . . . .	22
5.2 HER . . . . .	23
5.3 Goal Sampling . . . . .	25
<b>6 TP14 : Flow</b>	<b>28</b>
6.1 Transformation affine . . . . .	28
6.1.1 Calculs préliminaire et code . . . . .	28
6.1.2 Résultats expérimentaux . . . . .	29
6.2 Modèle GLOW . . . . .	30
6.2.1 Calculs préliminaires . . . . .	30
6.2.2 Expérimentations . . . . .	31

## Introduction

Ce rapport est un compte-rendu des travaux pratiques (2ème partie) liés à l'UE *Reinforcement Learning and advanced Deep Learning*. Il se compose de deux parties.

- La première concerne l'apprentissage par renforcement, et plus particulièrement de méthodes avancées de l'état de l'art du domaine. L'objectif général de ces TPs est l'apprentissage de diverses tâches par un agent placé dans un environnement donné, et avec lequel il interagit. Dans ce contexte, nous ne disposons que de peu d'informations sur l'environnement permettant de définir une politique de décision pour l'agent. Le but de l'apprentissage par renforcement est donc d'apprendre à effectuer une tâche donnée à partir de la perception du monde que nous avons à disposition et en se basant sur les retours de l'environnement selon les actions choisies. Les TPs associés à cette partie sont les TPs 3 (Apprentissage multi-Agent), 4 (Imitation learning) et 5 (Curriculum learning).
- La seconde concerne des techniques d'apprentissage profond et plus particulièrement sur les modèles génératifs. 3 modèles sont étudiés : les GAN 1, les VAE 2 et le modèle de flots 6.

Des expérimentations sont présentées ici, en insistant sur les résultats quantitatifs obtenus. Il s'agit d'un rapport expérimental. Plus de détails sur la théorie, et en particulier sur les pseudos-codes sont disponibles sur le site du cours d'apprentissage par renforcement de l'université de la Sorbonne, à l'adresse :<https://dac.lip6.fr/master/rld-2021-2022/>.

Le code associé à ce rendu permet de reproduire les expériences présentées. Le dossier est divisé en différentes parties, chacune d'entre elle correspondant à un TP en particulier.

Le premier (1) implémente un **GAN**, différente implémentation sur les jeux de données **MNIST** et **CelebA** sont proposées ainsi que 2 architectures différentes. Tout est présenté sous la forme d'un **notebook**. C'est également le cas pour le code associé à la section 2 qui présente également un modèle génératif : les **VAE**. Les trois sections suivantes (3 à 5) présentent des algorithmes de renforcements. Chacun d'entre eux sont associés à un dossier avec son propre code car ils ne s'exécutent pas sur les même environnements. Ainsi, chaque algorithme est implémenté sur un fichier à son nom, avec la boucle d'apprentissage associée. Finalement, le dernier TP (6) implémente un modèle génératif de flots. Les différentes composantes du modèles sont implémentées sur différents fichiers, et la boucle d'apprentissage les regroupant est disponible sur les fichiers **main\_exercice1.py** et **main\_exercice2.py**. Les courbes et images présentées dans ce rapport sont issues de **tensorboard** et sont disponibles dans les dossiers **XP**.

# 1 TP9 : GAN

## 1.1 Présentation de l'architecture GAN

Les GAN, inventés en 2014 (Generative Adversarial Network), sont des réseaux de neurones qui permettent de générer des données artificiellement selon la loi  $\mathcal{P}(X)$  des données d'apprentissage. Un GAN est composé de deux réseaux de neurones en compétition :

- Le générateur : Il crée des données ressemblant le plus possible aux données d'entraînement
- Le discriminateur : Il cherche à distinguer les images réelles des images créées par le générateur.

Chaque réseau s'améliore donc dans ses tâches respectives, le générateur cherche à tromper le discriminateur et le discriminateur à ne pas être trompé.

Sur un plan technique, les GAN transforment une entrée  $z \sim \mathcal{N}(0, I_n)$  en une image  $\hat{x} = G(z)$ . Quant à lui, le discriminateur vaut  $D(\hat{x}) = 0$  si  $\hat{x} = G(z)$ , 1 si  $\hat{x} \in \mathcal{D}\text{ata}$ . Concrètement, pour l'optimisation des réseaux on utilise la perte dérivée des équations :

- (1)  $\max_G \mathbb{E}[\log D(G(z))]$
- (2)  $\max_D \mathbb{E}[\log D(x^*)] + \mathbb{E}[\log(1 - D(G(z)))]$

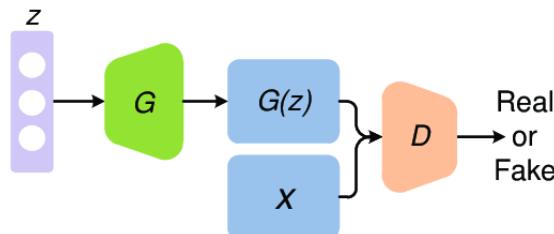


FIGURE 1 – L'architecture GAN

Les GAN ont servi en particulier dans la génération d'images, et c'est à cette fin que nous les utiliseront dans ce TP, à partir de la base de données `celebA` (portraits d'acteurs célèbres, exemples sur les figures 2a à 2c).



FIGURE 2 – Trois portraits issus de la base de données celebA

## 1.2 Expérimentations

### 1.2.1 GAN

On entraîne un GAN sur celebA pour reproduire des portraits d'acteurs. On choisit les hyperparamètres suivants :

- pas d'apprentissage du générateur : 0.001
- pas d'apprentissage du discriminateur : 0.001
- nombres d'époques : 20
- optimiseur du générateur : Adam
- optimiseur du discriminateur : Adam
- dimension de  $z$  : 100

Avant l'entraînement, les images générées par le générateurs correspondent à du bruit (les paramètres étant initialisés aléatoirement) comme présenté sur la figure 3.

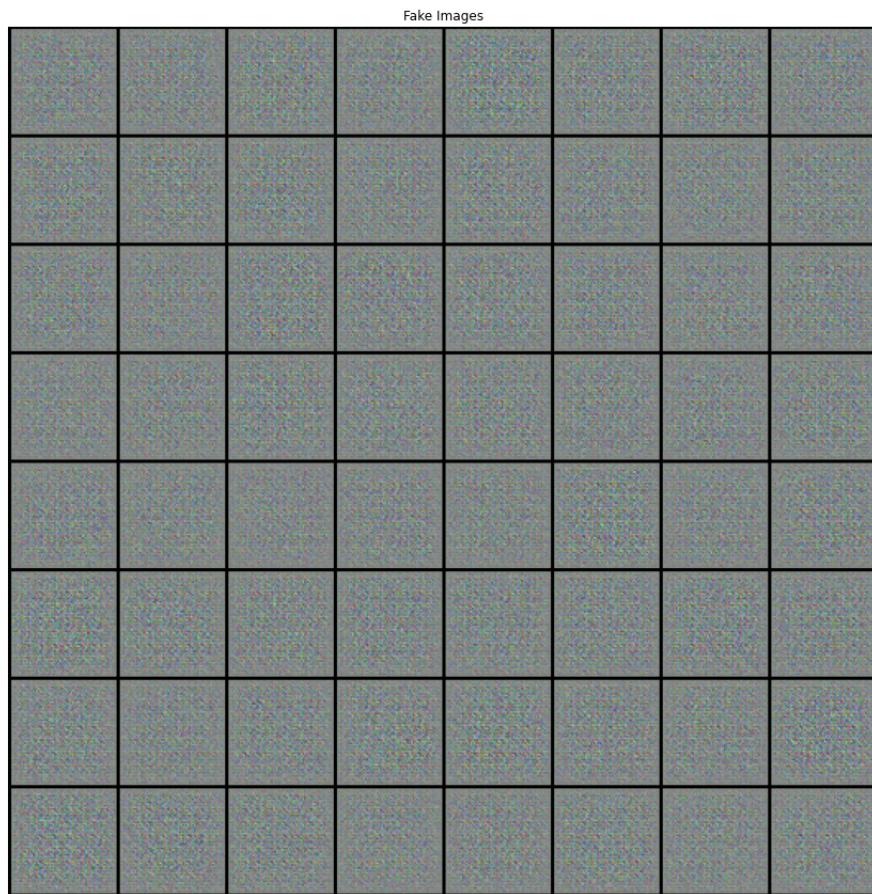


FIGURE 3 – Données générées par  $G$  avant tout entraînement

Au cours de l'entraînement, la qualité de reconstruction s'améliore grandement, comme présenté sur les figures 4a à 4d.

Les figures 5a à 7d présentent les fonctions de pertes du générateur et du discriminateur au cours de cet entraînement.

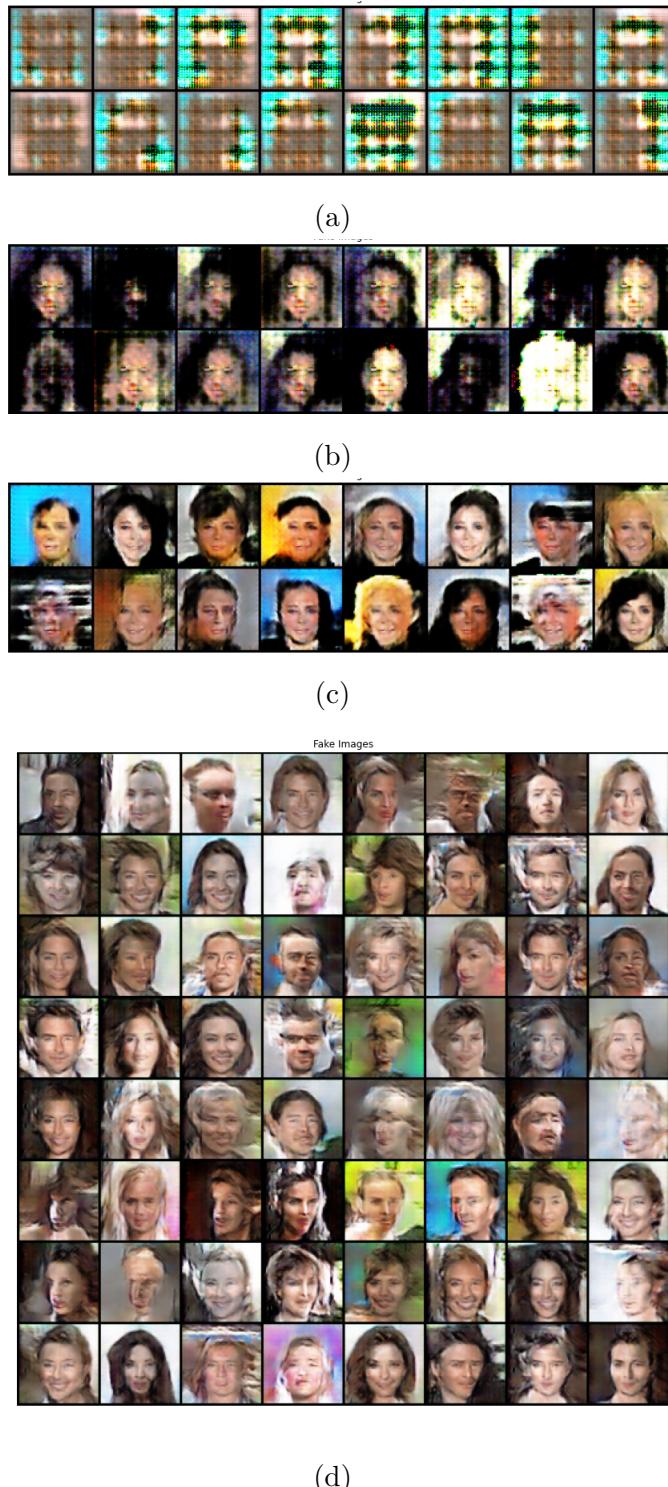


FIGURE 4 – Génération de portraits par GAN pour, de haut en bas, 1, 6, 15, 20 époques

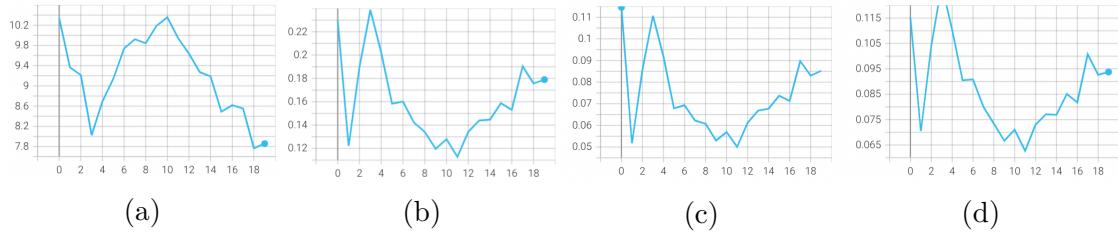


FIGURE 5 – De gauche à droite, de haut en bas, fonction de perte sur le générateur, sur le discriminateur, sur le discriminateur sur de fausses images, sur le discriminateur sur de vraies images.

On observe sur l'allure des pertes la compétition entre le générateur et le discriminateur, qui se traduit par une symétrie approximative des allures des courbes. Les pertes de première et de seconde espèces (comparaison des deux derniers graphiques) sont semblables. Cela signifie que pour le discriminateur il importe autant de considérer réelle une image fausse, que fausse une image réelle.

### 1.2.2 DCGAN sur celebA

Avec les couches de convolutions, DCGAN est particulièrement adapté au traitement d'images, on s'attend donc à obtenir de meilleurs résultats qu'avec un GAN.

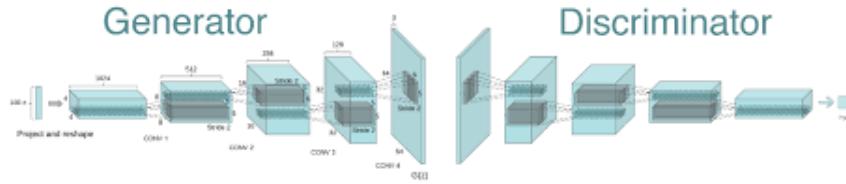


FIGURE 6 – Architecture DCGAN

L'entraînement (15 epochs) du DCGAN nous donne les résultats présentés sur les figures 8a à 8c pour les images reconstruites et 7a à 7d pour les fonctions de perte.

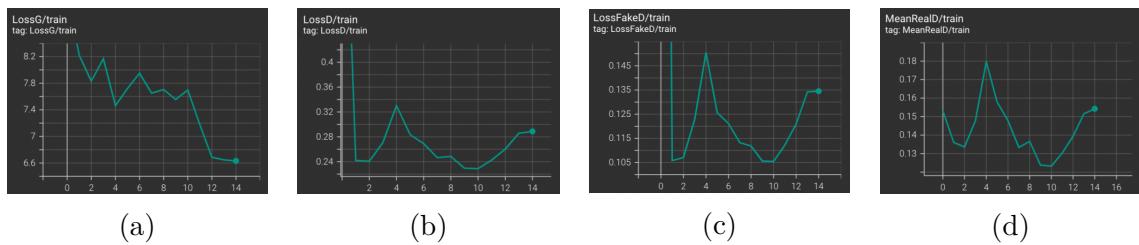


FIGURE 7 – De gauche à droite, de haut en bas, fonction de perte sur le générateur, sur le discriminateur, sur le discriminateur sur de fausses images, sur le discriminateur sur de vraies images.

Effectivement visuellement même avec 15 époques la reconstruction des visages semble meilleure qu'avec GAN.

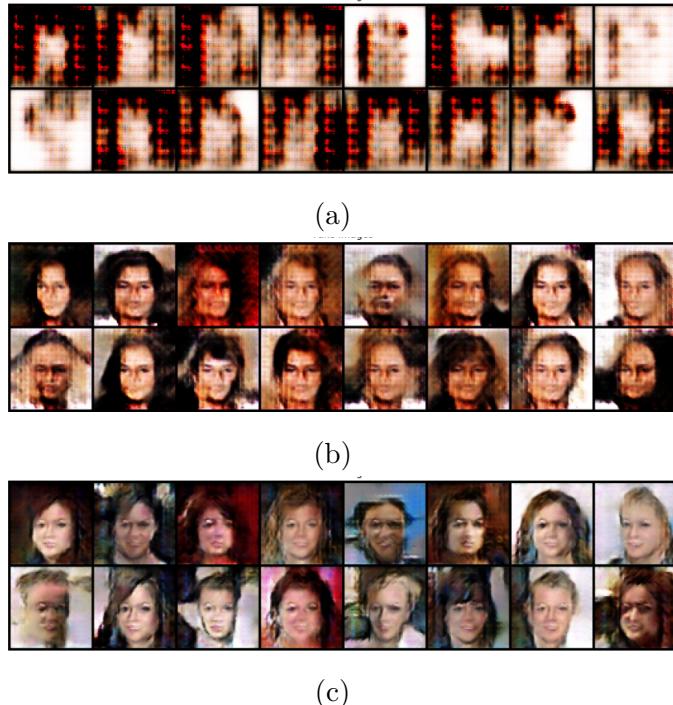


FIGURE 8 – Génération de portraits par DCGAN pour, de haut en bas, 1, 10, 15 époques

### 1.2.3 DCGAN sur MNIST :

Cette fois-ci, on entraîne un DCGAN, sur le jeu de données MNIST (chiffres écrits à la main). Pour nos expériences, on choisit les hyperparamètres suivants :

- pas d'apprentissage du générateur : 0.001
  - pas d'apprentissage du discriminateur : 0.001
  - nombres d'époques : 30
  - optimiseur du générateur : Adam
  - optimiseur du discriminateur : Adam
  - dimension de  $z$  : 100

Des images reconstruites sont présentées sur les figures 10a à 10c, et les fonctions de perte associées à cet entraînement sur les figures 9a à 9d.

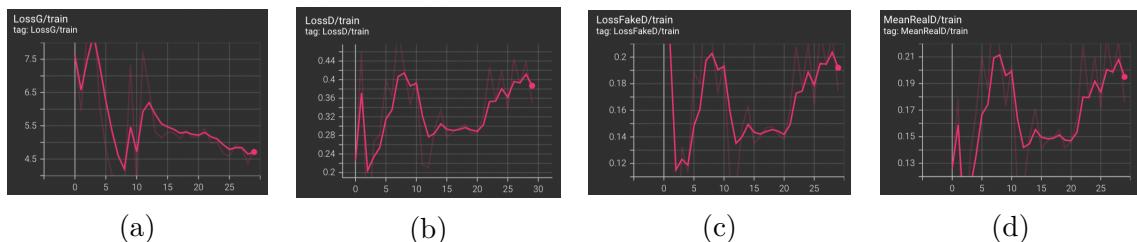


FIGURE 9 – De gauche à droite, de haut en bas, fonction de perte sur le générateur, sur le discriminateur, sur le discriminateur sur de fausses images, sur le discriminateur sur de vraies images.

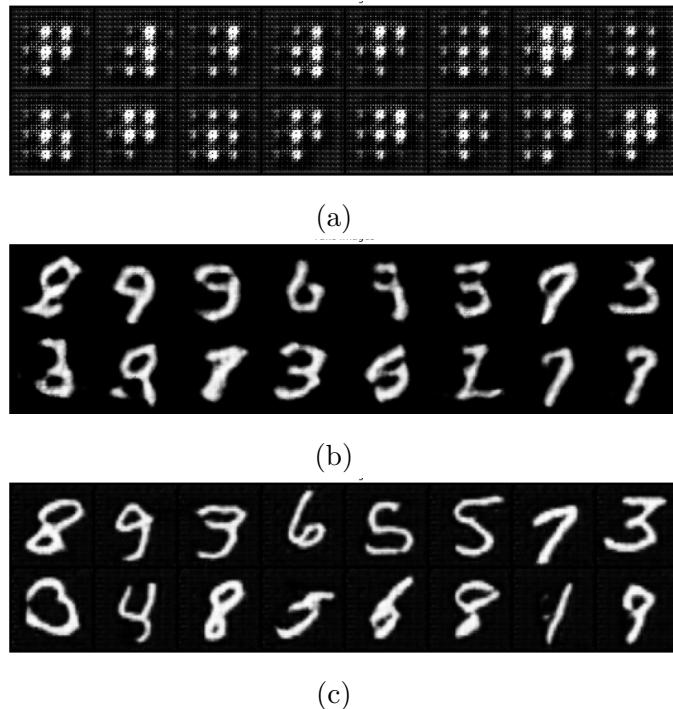


FIGURE 10 – De gauche à droite : reconstruction de MNIST par DCGAN après 1, 10 et 30 époques.

La génération est ici très bonne, cela est probablement imputable à la plus grande simplicité des données d'entraînements. MNIST est en noir et blanc, mais surtout, générer un portrait humain convaincant est bien plus difficile que générer un chiffre convaincant.

## 2 TP10 : Autoencodeur variationnel

### 2.1 Principe de fonctionnement et objectif d'un VAE

Un auto-encodeur variationnel est un modèle génératif : on cherche à reconstituer la loi qui engendre les données,  $\mathcal{P}(X)$ . On voit parfois les variables latentes  $z$  comme des *codes*. L'idée est de supposer que l'on peut retrouver la loi des données à partir d'une variable latente  $z$  suivant une loi simple (typiquement une gaussienne multivariée). En un certain sens on peut voir  $x \mapsto q_\phi(z|x)$  comme un *encodeur probabiliste* car à une valeur  $x$  de l'espace d'entrée on associe une densité sur l'espace des variables latentes. Et de la même manière nous pouvons voir  $p_\theta(x|z)$  comme un *décodeur probabiliste* (voir figure 11).

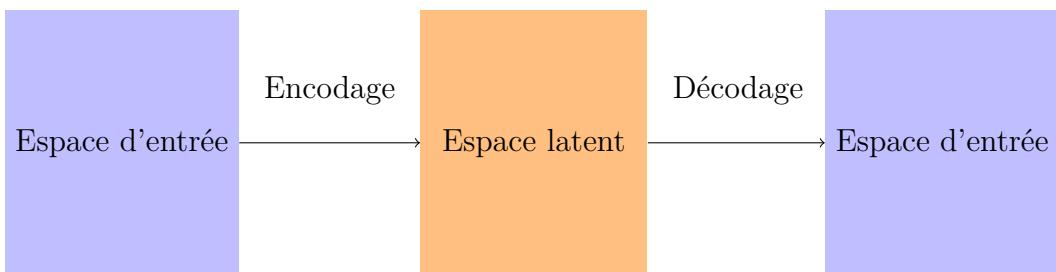


FIGURE 11 – Principe d'un VAE.

Dans un autoencodeur variationnel, on utilise  $q_\phi(z|x)$  comme encodeur et  $p_\theta(x|z)$  comme décodeur.  $q_\phi$  et  $p_\theta$  (figure 12) sont des réseaux de neurones que l'on optimise avec une fonction de coût comprenant deux termes :

- Un terme de reconstruction (log-vraisemblance) qui permet de retrouver en sortie  $\hat{x}$  ressemblant à une donnée simulée par  $\mathcal{P}(X)$
- Un terme de pénalisation pour contraindre l'encodeur (par divergence KL) à ressembler à la loi a priori sur les  $z$

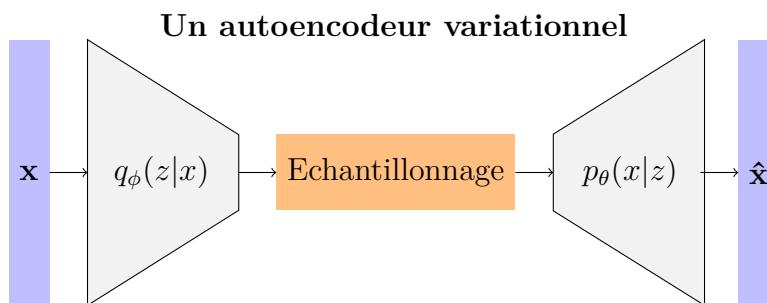


FIGURE 12

Dans ce TP on s'intéresse en particulier à reconstruire la loi générant des données MNIST. Puisqu'on ne peut rétropropager le gradient à travers un noeud stochastique, on utilise l'astuce de reparamétrisation : pour générer  $z \sim \mathcal{N}(\mu, \sigma)$ , on écrit  $z = \mu + \sigma \odot \varepsilon$  avec  $\varepsilon \sim \mathcal{N}(0, 1)$  de sorte que la partie stochastique de  $z$  n'est plus paramétrisée, on n'a plus besoin de rétropropager dessus.

## 2.2 Expérimentations

Pour chacune de nos expérimentations nous avons choisi les hyperparamètres suivants :

- Un pas d'apprentissage de 0.0001
- L'optimiseur Adam
- Des dimensions intermédiaires de 128 dans les réseaux de neurones correspondant à l'encodeur et au décodeur
- Un nombre d'époques valant 50 sauf pour la dernière expérimentation où l'on a choisi 100

Dans la suite on fait varier la dimension de l'espace latent afin d'étudier son influence sur la qualité de génération de VAE.

### 2.2.1 Dimension de l'espace latent = 1

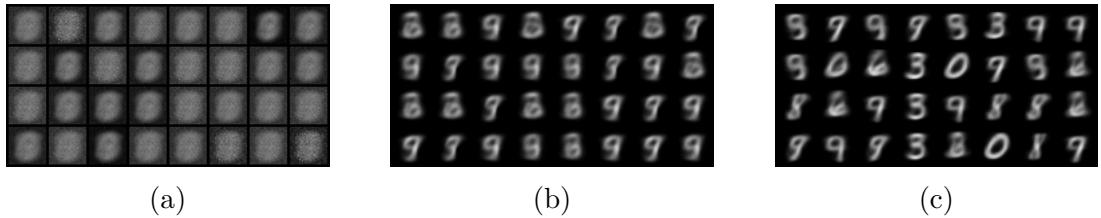


FIGURE 13 – De gauche à droite : reconstruction d'images par VAE après 0, 10 et 50 époques.

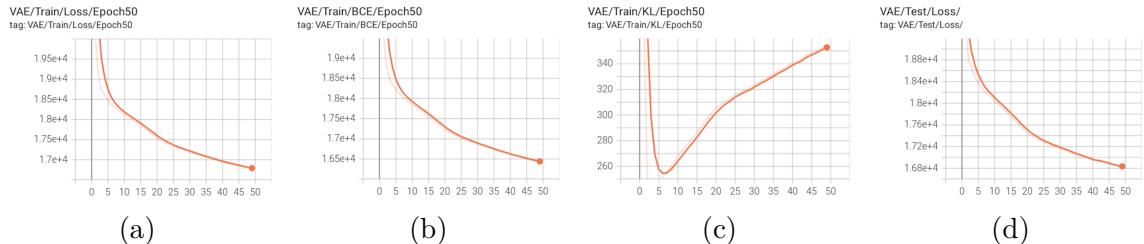


FIGURE 14 – Fonctions de perte en entraînement, perte associée aux termes de reconstruction et pénalisation et loss en test et pour un espace latent de dimension 1.

Après 50 époques on observe une génération d'assez bonne qualité, la plupart des chiffres sont classifiables sans ambiguïté par un œil humain (figures 13a à 13c). La perte globale est strictement décroissante, cependant bien que la perte de reconstruction soit strictement décroissante également, la perte de pénalisation devient quant à elle croissante après 10 époques. Le modèle favorise donc la reconstruction (figure 14).

Sur la figure 15, on présente la représentation de l'influence de  $z$  sur la génération d'un chiffre. En particulier on observe que des  $z$  proches induisent des chiffres générés proches, représentant le même nombre.



FIGURE 15 – Images générées dans un espace latent de dimension 1 (sur le segment  $[-1; 1]$ ).

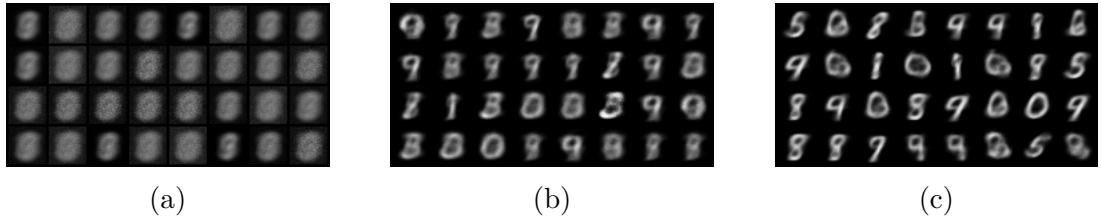


FIGURE 16 – De gauche à droite : reconstruction d’images par VAE après 0, 10 et 50 époques.

### 2.2.2 Dimension de l’espace latent = 2

La génération obtenue semble légèrement meilleure que celle pour  $\dim(z) = 1$  (figures 16a à 16c). Et effectivement la perte obtenue est 10% plus petite qu’avec  $\dim(z) = 1$  comme on le constate sur la figure 17.

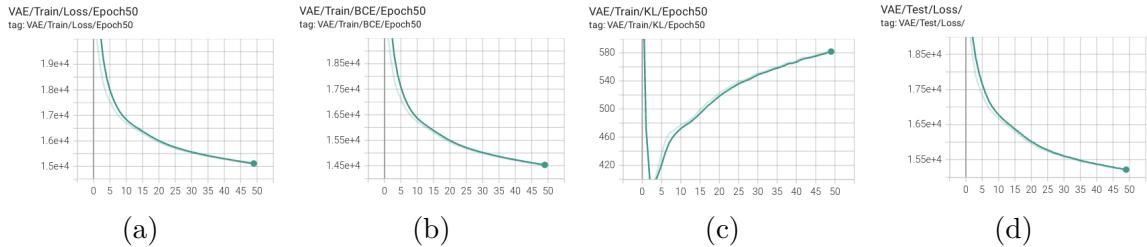


FIGURE 17 – Fonctions de perte, en entraînement, en test et perte associée aux termes de reconstruction et pénalisation pour un espace latent de dimension 2.

En dimension 2 on peut également observer la représentation de l’influence de  $z$  sur la génération d’un chiffre (voir figure 18). En particulier on observe que des  $z$  proches induisent des chiffres générés proches, représentant le même nombre.

Dans des dimensions supérieures à 2 il est plus difficile de représenter l’influence d’un  $z$  donné sur la génération des données.

### 2.2.3 Dimension de l’espace latent = 15

Malgré une perte plus faible dans cette expérience (figure 20) (même avec seulement 50 époques), visuellement, la reconstruction semble s’être déteriorée par rapport à  $\dim(z) = 2$  (figures 19a à 19c). En fait il est parfois difficile de déterminer la meilleure dimension pour l’espace latent, car nous disposons de peu de résultats théoriques sur ce choix.

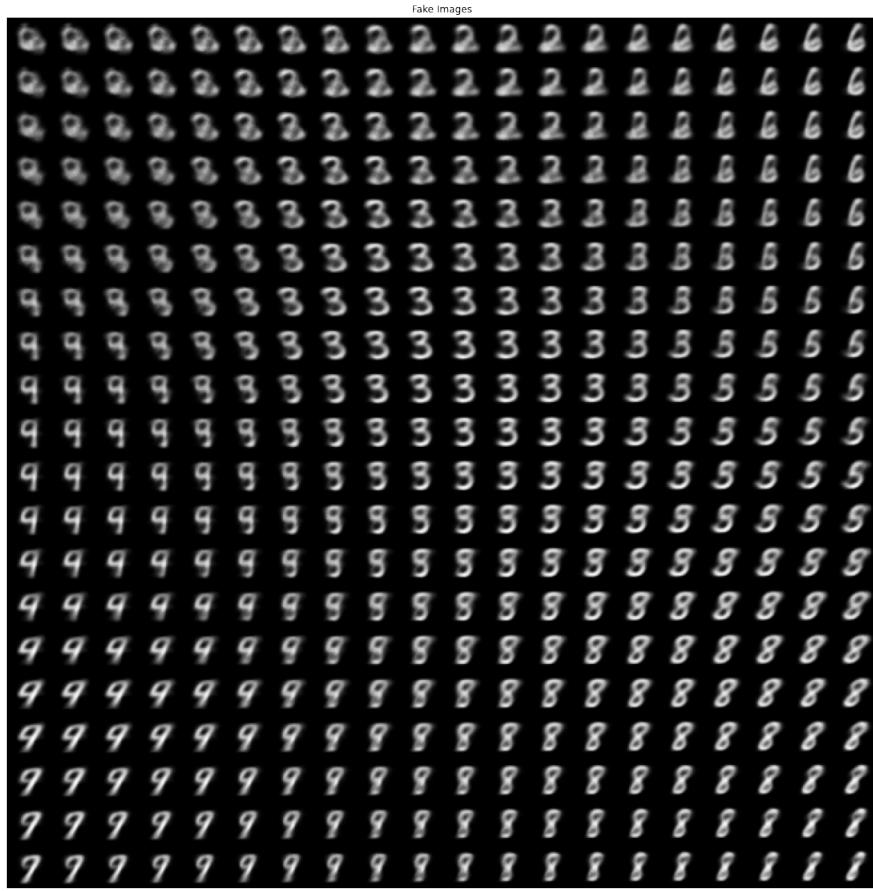


FIGURE 18 – Images générées dans un espace latent de dimension 2 (sur le carré  $[-1; 1]^2$ ).

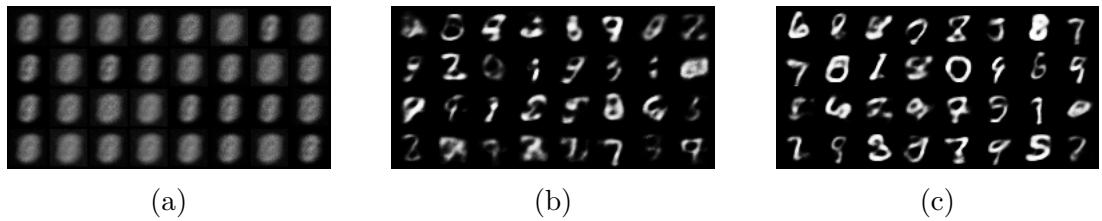


FIGURE 19 – De gauche à droite : reconstruction d’images par VAE après 0, 20 et 100 époques.

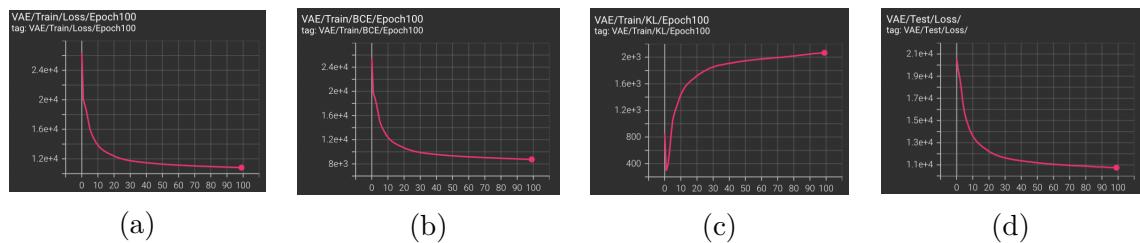


FIGURE 20 – Fonctions de perte en entraînement, perte associée aux termes de reconstruction et pénalisation et loss en test et pour un espace latent de dimension 15.

### 2.3 Qualité de reconstruction de l'algorithme

La qualité des reconstruction proposée par notre VAE semble suffisante pour tromper un ordinateur récent ! En effet, les chiffres sont reconnus et même identifiés comme un numéro de téléphone (voir figure 21).

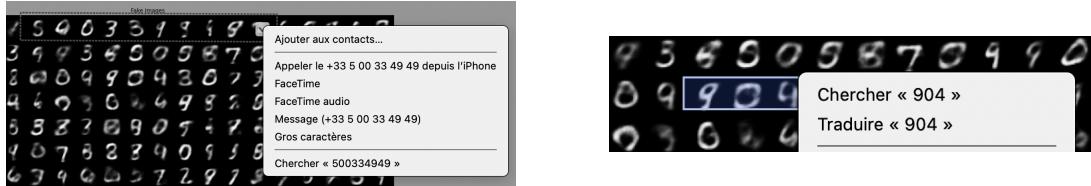


FIGURE 21 – Chiffres reconnus comme un numéro de téléphone par notre ordinateur.

### 3 TP11 : MADDPG

Dans ce TP, on s'intéresse à un problème multi-agent. L'environnement considéré est un environnement à actions continues. De ce fait, nous reprenons l'agent DDPG, implanté lors du TP 7. 3 scénarios existent : `simple_spread` (22a), `simple_adversary` (22b), `simple_tag` (22c).

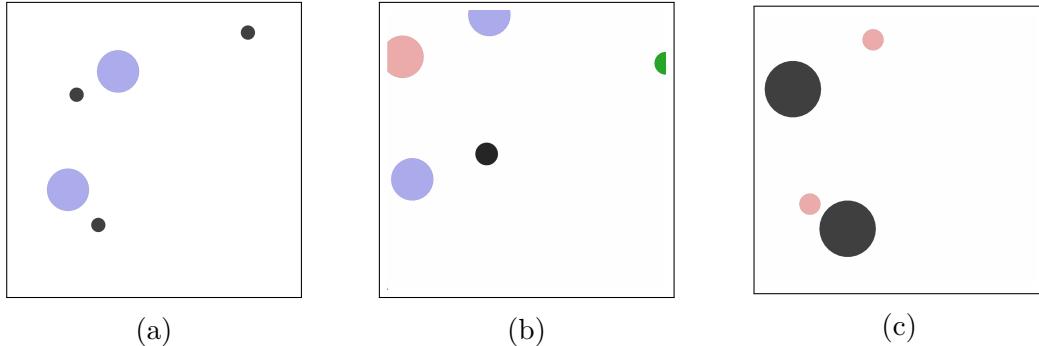


FIGURE 22 – Environnements considérés pour notre tâche d'apprentissage multi agent. De gauche à droite : `simple_spread`, `simple_adversary`, `simple_tag`.  
En bleu, les agents qui coopèrent, en rouge, les adversaires, en noir les bases et en vert les cibles.

Dans le premier environnement (figure 22a), les agents (bleus) doivent apprendre à collaborer afin d'être situés le plus proche possible des bases (noires), sans entrer en collision.

La seconde (22b) consiste en 3 agents : deux bons et un adversaire et 3 bases donc 1 cible. Le but de tous les agents est de s'approcher le plus proche de la cible. Seulement l'adversaire reçoit ses récompenses seul, quand les deux autres partagent les leurs. Ils doivent donc apprendre à collaborer pour empêcher l'adversaire de s'approcher, tout en garder un contrôle sur les bases.

Finalement, le dernier jeu 22c consiste en 3 prédateurs et 1 proie. Les prédateurs doivent apprendre à collaborer pour attraper la proie, qui doit elle, apprendre à les éviter.

L'algorithme utilisé pour résoudre ces tâches est l'algorithme **MADDPG** pour (Multi-Agent DDPG). chaque agent possède son propre algorithme d'apprentissage DDPG. Tous les agents reçoivent les informations de positions des autres afin d'optimiser leurs fonctions valeurs, mais leurs politiques leurs sont propres. On utilise les hyperparamètres suivants sur notre agent **MADDPG** :

- Un paramètre de discount de 0.95.
- Un pas d'apprentissage de 0.01 pour l'optimiseur Adam pour le réseau Q et 0.01 pour le réseau de la politique.
- Une mémoire de taille 10000 dont on extrait un batch de taille 100. Les rewards sont clippée pour rester entre -10 et 10.
- Un réseau pour la fonction Q avec une couches cachée de 128 neurones et une activation *leakyRelu* (pas d'activation finale).

- Un réseau acteur (politique) avec une couche cachée de 128 neurones, une activation *leakyRelu* et une activation finale tanh.
- Deux réseaux cibles pour la politique et la fonction Q, mis à jour toutes les 10 étapes.
- Un bruit Orn-Uhlen de paramètres  $\mu = 0$  et  $\sigma = 0.2$ .

### 3.1 Coopérative navigation

En utilisant les hyperparamètres précédents sur l'environnement `simple_spread`, on obtient les résultats présentés en figures 23a à 23d.

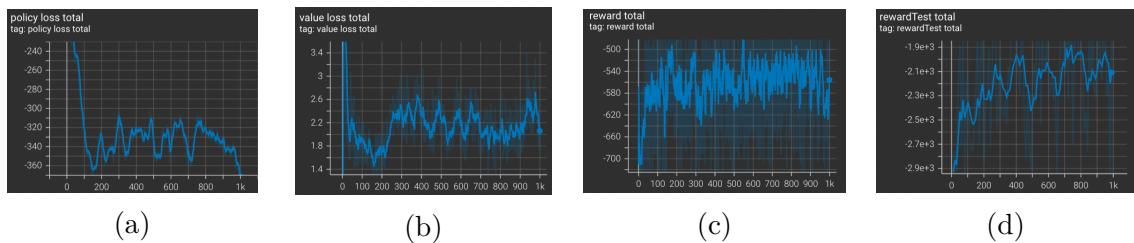


FIGURE 23 – De gauche à droite : Loss pour l'acteur et la critique et récompenses en entraînement et en test obtenues avec notre implémentation de MADDPG sur l'environnement à actions continues `simple_spread`. Les valeurs de reward présentées ici correspondent à la somme des rewards des 3 agents.

On voit clairement au niveau des rewards que l'agent a appris une stratégie lui permettant de maximiser ses récompenses. Les loss associées aux réseaux Q et de l'acteur se stabilisent dans le même temps. En pratique, on observe que les agents se placent à des endroits intéressants sur la carte pour être au plus proche des bases.

### 3.2 Physical deception

Sur le jeu `simple_adversary`, on obtient les résultats présentés en figures 24a à 27d.

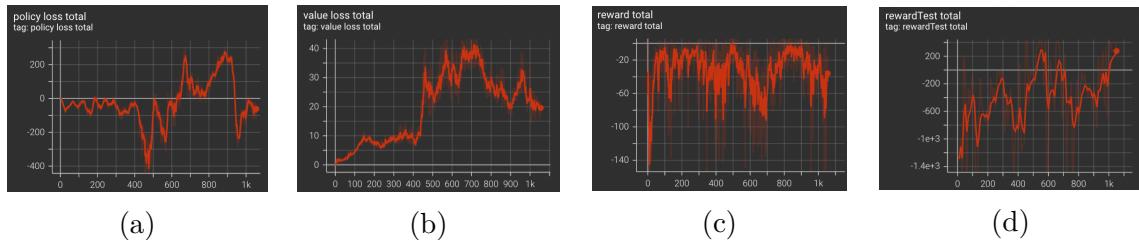


FIGURE 24 – De gauche à droite : Loss pour l'acteur et la critique et récompenses en entraînement et en test obtenues avec notre implémentation de MADDPG sur l'environnement à actions continues `simple_adversary`. Les valeurs de reward présentées ici correspondent à la somme des rewards des 3 agents.

Pour ce jeu, on détaille les indicateurs pour chacun des agents. En effet, dans ce jeu, les récompenses obtenues ne sont pas les mêmes pour tous les agents, car l'adversaire

a un but différent des 2 agents coopérants. Les résultats détaillés sont présentés dans les figures 25a à 27d. Dans ces résultats, l’adversaire correspond à l’agent 0 (figures 25a à 25d) et les deux autres agents sont numérotés 1 et 2 sur les graphiques (figure 26a à 27d). Les indicateurs globaux, nous permettent de constater que l’agent apprend à jouer puisque les récompenses augmentent (24c, 24d).



FIGURE 25 – De gauche à droite : Loss pour l’acteur et la critique et récompenses en entraînement et en test obtenues avec notre implémentation de MADDPG sur l’environnement à actions continues simple\_adversary pour l’agent 0 (adversaire).



FIGURE 26 – De gauche à droite : Loss pour l’acteur et la critique et récompenses en entraînement et en test obtenues avec notre implémentation de MADDPG sur l’environnement à actions continues simple\_adversary pour l’agent 1 (coopérant)..

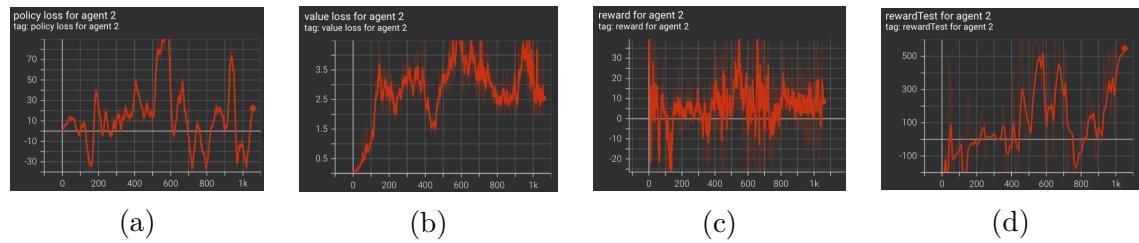


FIGURE 27 – De gauche à droite : Loss pour l’acteur et la critique et récompenses en entraînement et en test obtenues avec notre implémentation de MADDPG sur l’environnement à actions continues simple\_adversary pour l’agent 2 (coopérant).

De par la confrontation qui existe entre les deux agents, on note que l’apprentissage est plus instable (voir loss comme par exemple la figure 25a). A mesure que les itérations avancent, on note les différentes stratégies apprises par les agents, notamment visible sur les récompenses en entraînement (25c, 26c, 27c). En effet dès le début, c’est l’adversaire qui apprend à augmenter significativement ses récompenses, et celles des agents coopérants diminuent. En réponse à cela, leurs politiques s’adaptent et ils apprennent à coopérer pour déstabiliser l’adversaire, ce qui entraîne une baisse des

récompenses de ce dernier (autour des epochs 700). Il réagit donc en trouvant une autre stratégie lui permettant d'éviter les deux agents (autour de 850 epochs). Si l'entraînement s'était poursuivi plus longtemps, cette alternance se serait probablement poursuivie. Ces éléments sont également visibles sur les loss des agents (figures a et b pour chacun des agents), avec par exemple, autour de l'epoch 600, le changement de comportement de l'adversaire qui est visible.

### 3.3 Predator-prey

Sur le jeu `simple_tag`, on obtient les résultats présentés en figures 28a à 31d. Pour rappel, 4 agents sont impliqués : 3 prédateurs et 1 proie. Les résultats présentés par la suite sont obtenus en utilisant le même jeu d'hyperparamètres que dans les précédents jeux.



FIGURE 28 – De gauche à droite : Loss pour l'acteur et la critique et récompenses en entraînement et en test obtenues avec notre implémentation de MADDPG sur l'environnement à actions continues `simple_tag`. Les valeurs de reward présentées ici correspondent à la somme des rewards des 4 agents.

Pour ce jeu également, on détaille les indicateurs pour chacun des agents. En effet, dans ce jeu, les récompenses obtenues ne sont pas les mêmes pour tous les agents, car les prédateurs ont un but différent de la proie. Les résultats détaillés sont présentés dans les figures 29a à 32d. Dans ces résultats, les prédateurs correspondent aux agents 0 à 2 (figures 29a à 31d) et la proie est numérotée 3 sur les graphiques (figure 32a à 32d).

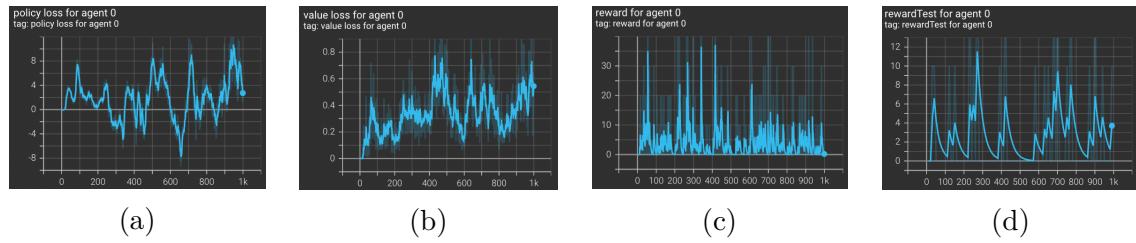


FIGURE 29 – De gauche à droite : Loss pour l'acteur et la critique et récompenses en entraînement et en test obtenues avec notre implémentation de MADDPG sur l'environnement à actions continues `simple_tag` pour l'agent 0 (prédateur).

Pour ce jeu, on remarque que c'est surtout la proie qui a appris à éviter ses prédateurs, et ce, dès le début de l'apprentissage (les récompenses sont intéressantes dès l'epoch 100). Cela est probablement dû au fait que les récompenses de la proie sont bien inférieures à celles des prédateurs. En conséquence sa politique est plus rapidement apprise.



FIGURE 30 – De gauche à droite : Loss pour l’acteur et la critique et récompenses en entraînement et en test obtenues avec notre implémentation de MADDPG sur l’environnement à actions continues simple\_tag pour l’agent 1 (prédateur).

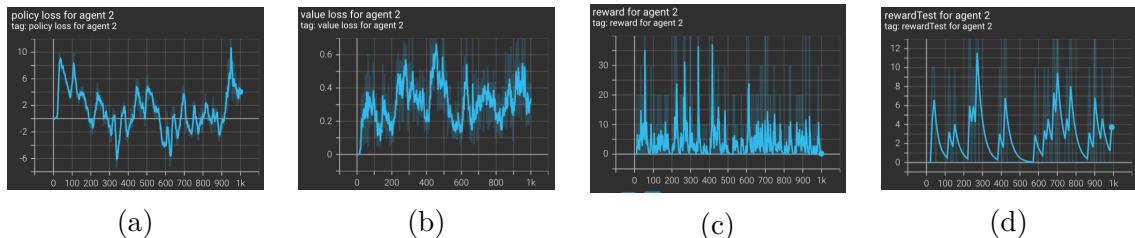


FIGURE 31 – De gauche à droite : Loss pour l’acteur et la critique et récompenses en entraînement et en test obtenues avec notre implémentation de MADDPG sur l’environnement à actions continues simple\_tag pour l’agent 2 (prédateur).

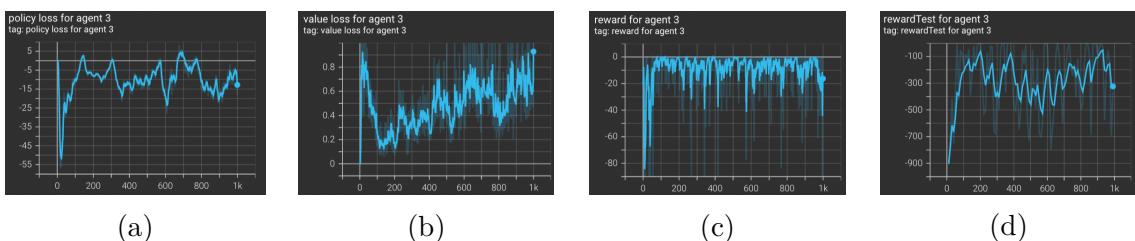


FIGURE 32 – De gauche à droite : Loss pour l’acteur et la critique et récompenses en entraînement et en test obtenues avec notre implémentation de MADDPG sur l’environnement à actions continues simple\_tag pour l’agent 3 (proie).

## 4 TP12 : Imitation Learning

Dans ce TP, on va considérer un problème d'apprentissage guidé (Imitation Learning). C'est à dire qu'on a accès à des données expertes permettant de guider l'apprentissage de l'agent vers une politique intéressante si ce n'est optimale. Plusieurs approches de cette branche de l'apprentissage par renforcement existent : Behavioral cloning, Interactive policy learning, apprenticeship learning... Après une première approche de **Behavioral cloning** très simple, nous nous intéresserons à l'algorithme **GAIL**, pour Generative Adversarial Imitation Learning, qui propose de générer des trajectoires indiscernables de celles de l'expert par un discriminateur appris. Nous reprenons l'environnement lunar lander (discret), déjà étudié lors de précédents TP et pour lequel on dispose de trajectoires expertes ayant un reward élevé.

### 4.1 Behavioral Clonning

Cette première approche consiste en un algorithme très simple composé d'un réseau de politique qui apprend à imiter les trajectoires de l'expert. Pour cela, nous utilisons les paramètres suivants :

- Un réseau modélisant la politique  $\pi_\theta$  à 2 couches cachées de 30 neurones chacune, avec une activation intermédiaire `tanh` et une activation finale `sigmoid`.
- un optimiseur Adam avec un pas d'apprentissage initial de 0.01
- Une loss  $\ell_1$  `smooth` calculée entre la sortie de la politique et la trajectoire issue de l'expert

Ces paramètres nous donnent les résultats présentés sur les figures 33a à 33c.

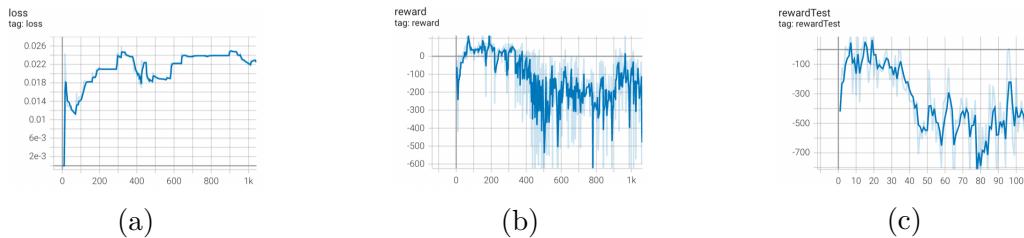


FIGURE 33 – De gauche à droite : Loss pour l'acteur et récompenses en entraînement et en test obtenues avec notre implémentation de Behavrioral Clonning sur l'environnement Lunar Lander sur 1000 epochs.

Comme attendu, on observe une très forte augmentation des reward, suivi d'une diminution de ces dernières. Notre agent sur-apprend les trajectoires de l'expert et ne réussit pas à généraliser correctement face à de nouvelles situations.

### 4.2 GAIL

On reprend notre implémentation de l'algorithme `PP0_clipped` comme base pour notre agent.

- Un paramètre de discount de 0.999.
- Un pas d'apprentissage variant de 0.0001 à 0.01 pour l'optimiseur Adam pour les 3 réseaux impliqués

- Une mémoire de taille 10000 dont on extrait un batch de taille 1000.
- Les rewards sont les sorties du discriminateurs, le log de ces sorties ou  $-\log(1-x)$  de ces sorties.
- Un réseau pour la fonction Q avec deux couches cachées de 30 à 130 neurones et une activation tanh (pas d'activation finale).
- Un réseau acteur (politique) avec deux couches cachées de 30 à 130 neurones, une activation *tanh* et une activation finale *Softmax*.
- Un réseau discriminateur avec deux couches cachées de 30 à 130 neurones, une activation *tanh* et une activation finale *Sigmoid*.
- Un bruit Gaussien de paramètres  $\mu = 0$  et  $\sigma = 0.01$  pour bruiter les observations en entrée du discriminateur
- une loss `11-smooth` pour la critique et `BCE` ou `BCEwithLogit` pour le discriminateur
- un agent `PPO_clipped` ou `PPO_KL` ou `A2C` pour mettre à jour la politique

Cet algorithme nous aura posé quelques difficultés. En effet, notre agent ne réussit pas à trouver de politique optimal intéressante lui permettant d'atteindre des récompenses positives, et ce, malgré les différents jeux d'hyper-paramètres testés. Des résultats d'entraînements sont présentés sur les figures 34a à 34g.

Sur les graphiques de la figure 34, on voit que l'agent a bien convergé, notamment au niveau des loss qui sont très stable pour la plupart des entraînements. C'est moins évident pour les récompenses, mais de façon globale, elles convergent toutes vers une politique avec des récompenses moyennes autour de -150. Cette valeur est déjà satisfaisante car elle témoigne qu'une certaine politique a été apprise, mais pas suffisante au vu des données expertes incorporées. On s'attendait en effet à atteindre au moins des récompense positives comme l'agent PPO réussissait à atteindre, bien que bien moins stable lors de son apprentissage. De plus, le discriminateur est trop discriminant sur les scores qu'il produit. Les probabilités pour la classification sont très proches de 1 ou 0, quand on s'attendrait à une classification plus nuancée afin de guider l'agent (0.8-0.2).

En comparaison, voici les résultats obtenus avec l'agent PPO (KL), implémenté lors du TP6 (figure 35a à 35b). Ces résultats ont été obtenus en prenant les mêmes paramètres que pour GAIL, excepté :

- Les réseaux critiques et acteurs n'ont que 30 neurones sur leurs 2 couches
- les pas d'apprentissages associés sont de 0.01

On note que PPO met du temps avant de trouver une politique intéressante. Cependant, les récompenses associées sont positives, ce qui témoigne d'un bon apprentissage. De plus, l'apprentissage est très instable : on observe de brusques changement de politiques, et notre algorithme se trouve être très sensible aux coups aléatoires portés sur les premières itérations et dans la suite de l'algorithme (comme précisé dans le rapport précédent dans le TP6).

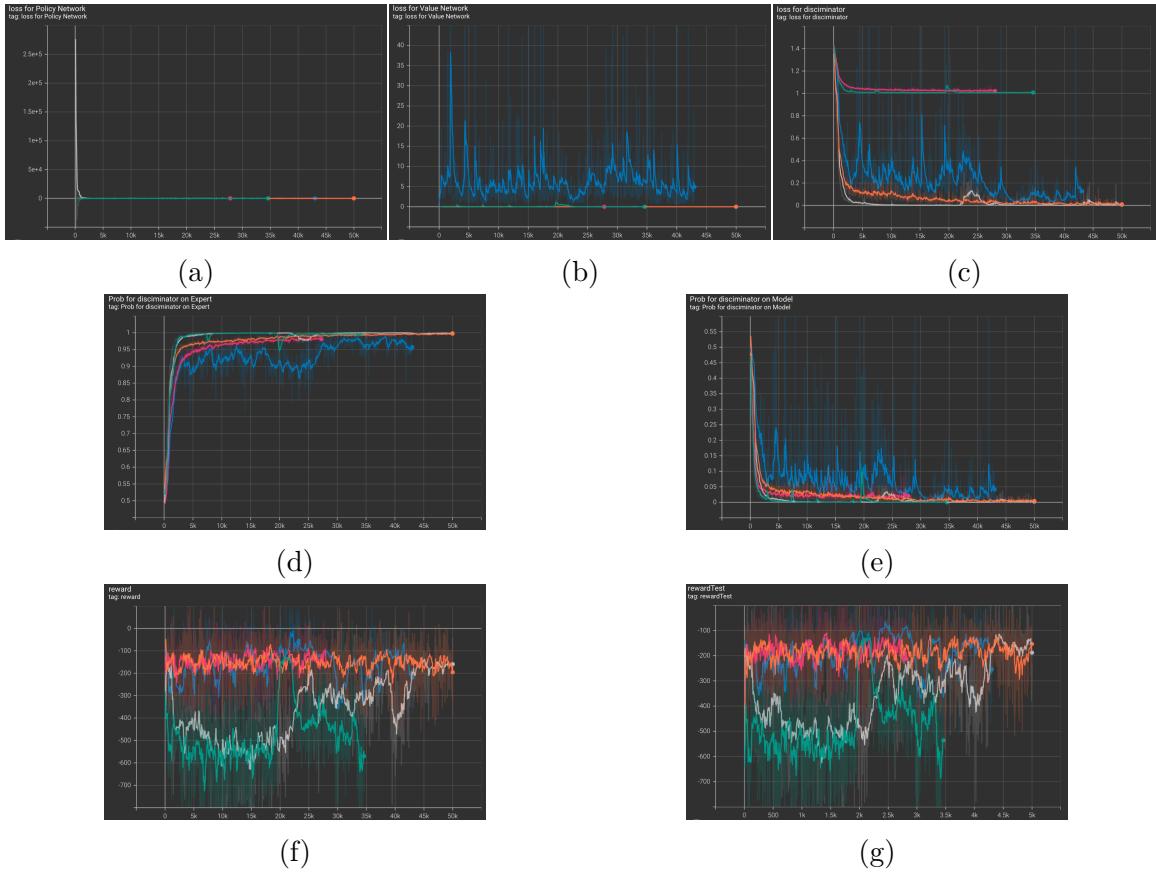


FIGURE 34 – Plusieurs résultats de notre agent GAIL sur l'environnement lunaire. De gauche à droite :

Première ligne : loss de l'acteur, loss de la critique, loss du discriminateur

Deuxième ligne : scores du discriminateur sur les trajectoires expertes et générées par l'agent

Troisième ligne : récompenses en entraînement et en test de l'agent.

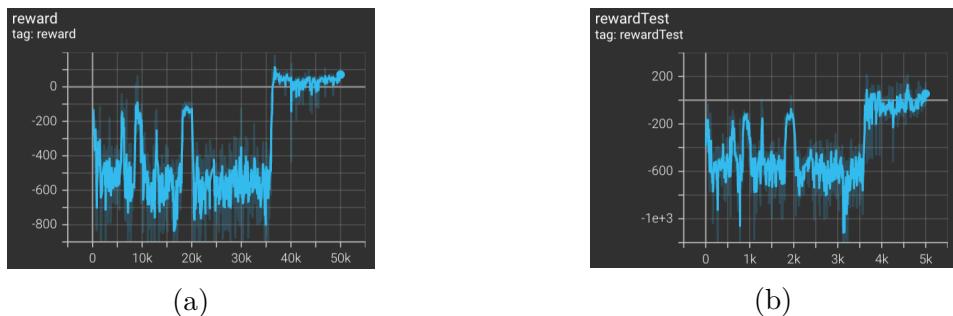


FIGURE 35 – De gauche à droite : Récompenses en entraînement et en test obtenues avec notre implémentation de PPO du TP6 sur l'environnement Lunar Lander sur 50000 epochs.

## 5 TP13 : Implicit Curriculum RL

Dans ce TP, on s'intéressera à différentes méthodes de Curriculum Learning. 3 d'entre elles seront étudiées : GoalDQN, HER et Goal Sampling. Elles seront appliquées au jeu gridworld, sur les cartes suivantes [36a](#) et [36b](#).

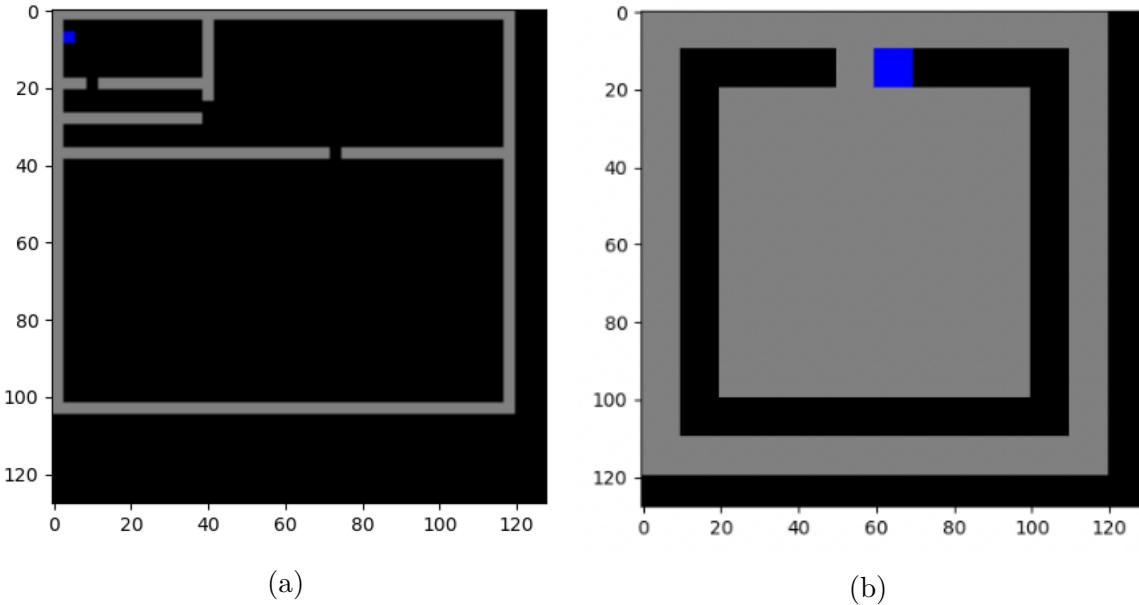


FIGURE 36 – Carte 2 et 3 de l'environnement Gridworld considéré

### 5.1 Goal-base DQN

Dans cette section, nous présentons les résultats obtenus sur l'environnement Gridworld avec l'algorithme goal-DQN. Pour ce faire, l'implémentation présentée dans le TP4 est modifiée afin d'intégrer l'information sur les buts à l'agent. En utilisant les hyperparamètres suivants, on obtient les courbes présentées sur les figures [37a](#) à [37c](#).

- Une stratégie d'exploration  $\epsilon$ -greedy avec un seuil d'exploration de 0.2, qui décroît avec un facteur 0.99 à chaque itération.
- Un paramètre de discount de 0.99.
- Un pas d'apprentissage de 0.001 pour l'optimiseur Adam.
- L'utilisation d'un réseau cible que l'on optimise toutes les 100 étapes .
- Une mémoire de taille 10000 dont on extrait des mini-batchs de taille 100.
- Un réseau Q avec deux couches cachées de 200 neurones

L'environnement utilisé est une carte très complexe (voir figure [36a](#)), sur laquelle des buts intermédiaires ont été renseignés, afin de guider l'agent dans sa résolution du problème.

En utilisant les mêmes paramètres avec un DQN simple, on obtient les résultats figures [38a](#) à [38c](#). On voit clairement que DQN diverge et est perdu sur une carte aussi grande.

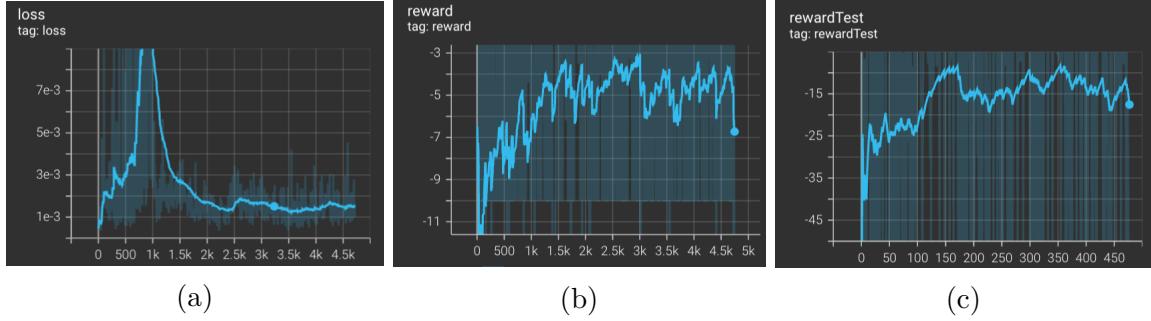


FIGURE 37 – De gauche à droite : Loss, reward en entraînement et en test obtenus après 5000 itérations de goal-DQN.

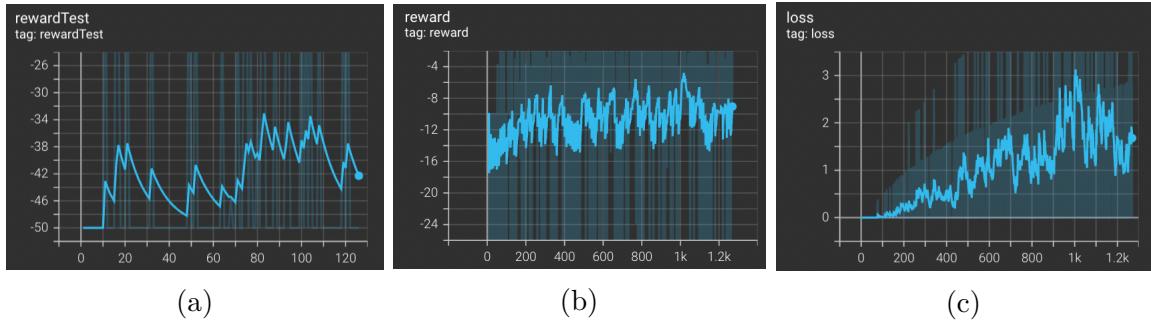


FIGURE 38 – De gauche à droite : Loss, reward en entraînement et en test obtenus après 1200 itérations de DQN.

## 5.2 HER

Lorsque la carte est plus compliquée, l'algorithme Goal DQN a des difficultés à apprendre une quelconque politique. On reprend la carte précédente (36a) et on enlève les récompenses intermédiaires c'est-à-dire qu'aucune récompense guidant l'agent n'est obtenue (voir figures 39a à 39d). Dans la suite, nous utiliserons également comme indicateur, les positions finales atteintes par l'agent, c'est-à-dire lorsqu'un signal `done` est reçu par l'algorithme. Cela peut arriver lorsqu'un but est atteint ou lorsque la limite de temps est passée. Cet indicateur nous intéressera plus que celui des reward. En effet, les récompenses peuvent être élevées parce que l'agent atteint facilement des buts intermédiaires, ce qui n'est pas la tâche finale intéressante dans ce cadre.

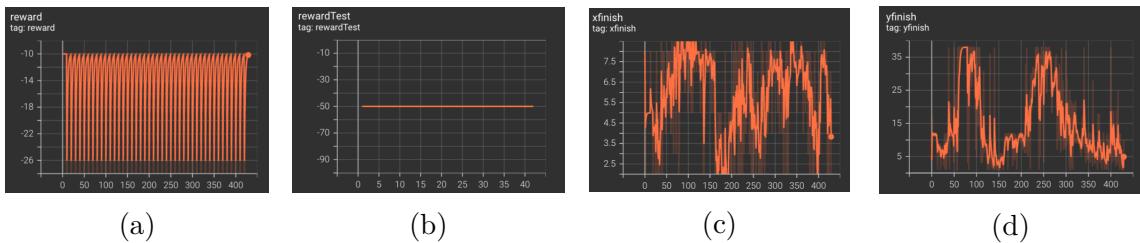


FIGURE 39 – De gauche à droite : reward en entraînement et en test obtenus et positions finales atteintes par l’agent après 500 itérations de goalDQN sur la carte 2 (36a) sans guidage.

Les positions finales atteintes par l'agent sont très aléatoire de part sa difficulté à

apprendre une quelconque politique pertinente sur cette carte. Au niveau des récompenses, elles sont très mauvaises, notamment en test, ou elles sont au plus bas, sans aucune trace d'apprentissage.

HER permet de surmonter ce problème en ajoutant les état déjà atteints en tant que but. Cela permet à l'algorithme de retrouver dans ces zones plus rapidement afin de continuer à explorer et trouver le véritable but final. La carte utilisée pour cet algorithme est la carte présentée à la figure 36a, mais sans buts intermédiaires. Les résultats ont été obtenus avec les hyper paramètres suivants et sont présentés dans les figures 40a à 40c.

- Une stratégie d'exploration  $\epsilon$ -greedy avec un seuil d'exploration de 0.2, qui décroît avec un facteur 0.99 à chaque itération.
- Un paramètre de discount de 0.99.
- Un pas d'apprentissage de 0.001 pour l'optimiseur Adam.
- L'utilisation d'un réseau cible que l'on optimise toutes les 100 étapes .
- Une mémoire de taille 10000 dont on extrait des mini-batchs de taille 100.
- Un réseau Q avec deux couches cachées de 200 neurones

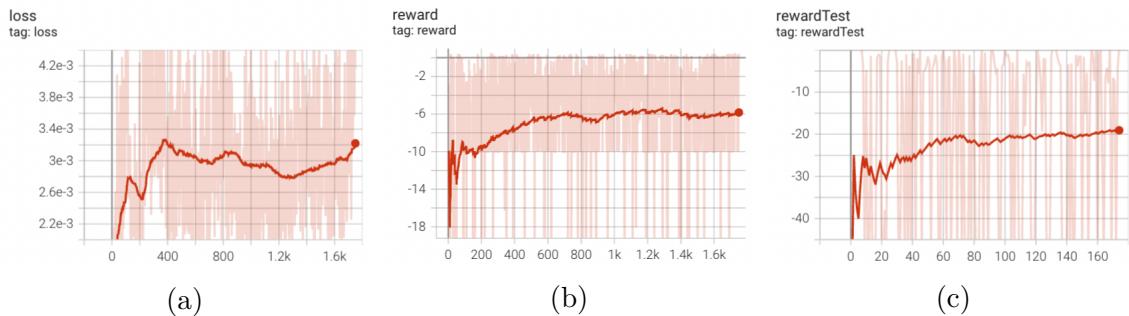


FIGURE 40 – De gauche à droite : Loss, reward en entraînement et en test obtenus après 1200 itérations de HER.

En comparant avec les résultats obtenus sur les figures 38a à 38c, on note que les reward sont bien meilleurs avec le mécanisme HER qu'avec un GoalDQN classique. On obtient alors des récompenses de  $-6$  en entraînement (contre  $-12$  avec GoalDQN) et  $-20$  en test ( $-46$ ). Une seconde exécution de l'algorithme sur 1000 epochs ajoute l'affichage des état finaux afin de vérifier que l'agent atteint bien l'état souhaité. On remarque dans les figure 41a à 41c, que l'agent réussit à atteindre l'état visé (coordonnées plus élevées).

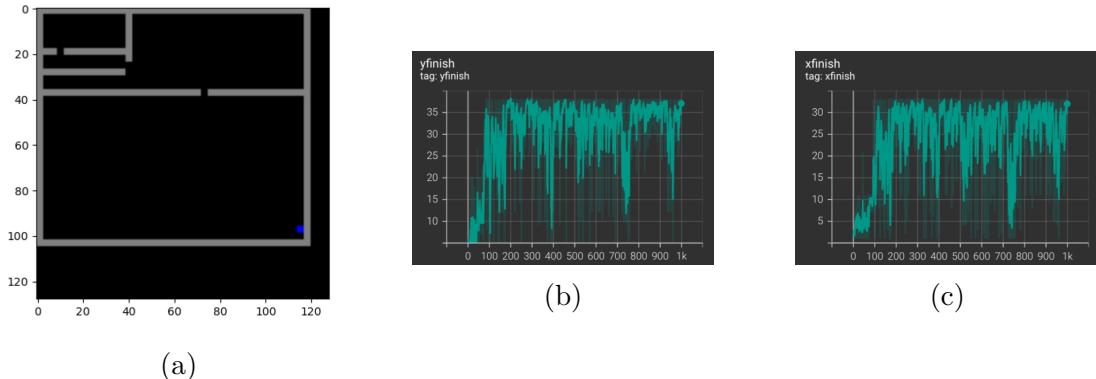


FIGURE 41 – De gauche à droite : état final atteint, position finale de l’agent obtenus après 1000 itérations de HER.

Sur la troisième carte (figure 36b), HER produit également de très bon résultats (figures 42a à 42f).

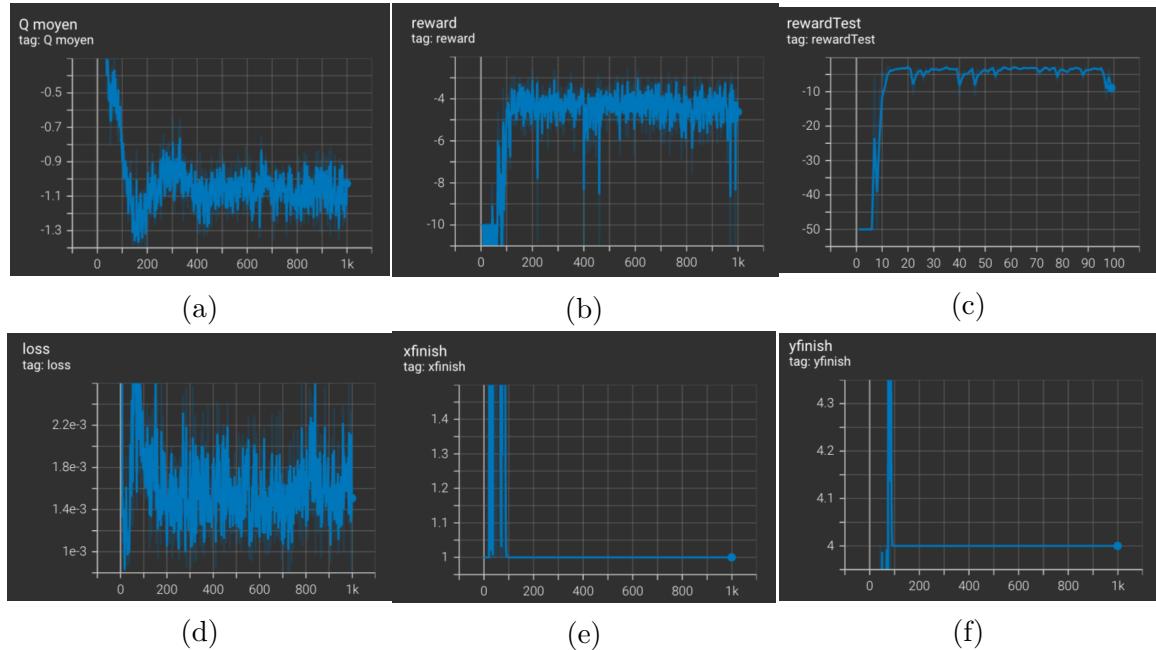


FIGURE 42 – De gauche à droite et de haut en bas : valeur Q moyennes, reward en entraînement et en test, loss, position finale en x et y obtenus après 1000 itérations de HER sur la carte 36b

### 5.3 Goal Sampling

La dernière méthode étudiée dans ce TP est l’échantillonage de buts. Dans cet algorithme, on ajoute des buts fictifs que l’agent doit tenter d’atteindre. Ces derniers sont échantillonés selon une probabilité définie selon le nombre de fois où le but a été choisi et atteint. Le vrai but de l’environnement est également échantilloné selon une probabilité  $1 - \beta$ . En utilisant les paramètres suivant :

- Une stratégie d'exploration  $\epsilon$ -greedy avec un seuil d'exploration de 0.2, qui ne décroît pas (decay de 1) à chaque itération.
- Un paramètre de discount de 0.99.
- Un pas d'apprentissage de 0.001 pour l'optimiseur Adam.
- L'utilisation d'un réseau cible que l'on optimise toutes les 100 étapes .
- Une mémoire de taille 10000 dont on extrait des mini-batchs de taille 100.
- Une mémoire d'épisode de taille 1000
- Une liste de buts FIFO contenant 10 faux buts et alimentée toutes les 10 étapes, avec une probabilité d'échantillonner un faux but de  $\beta = 0.9$ , et un paramètre  $\alpha = 3$  (pour le choix du faux but lorsque l'algorithme a choisi d'en considérer un).
- Un réseau Q avec deux couches cachées de 200 neurones

On obtient les résultats présentés figures 43a à 43e.

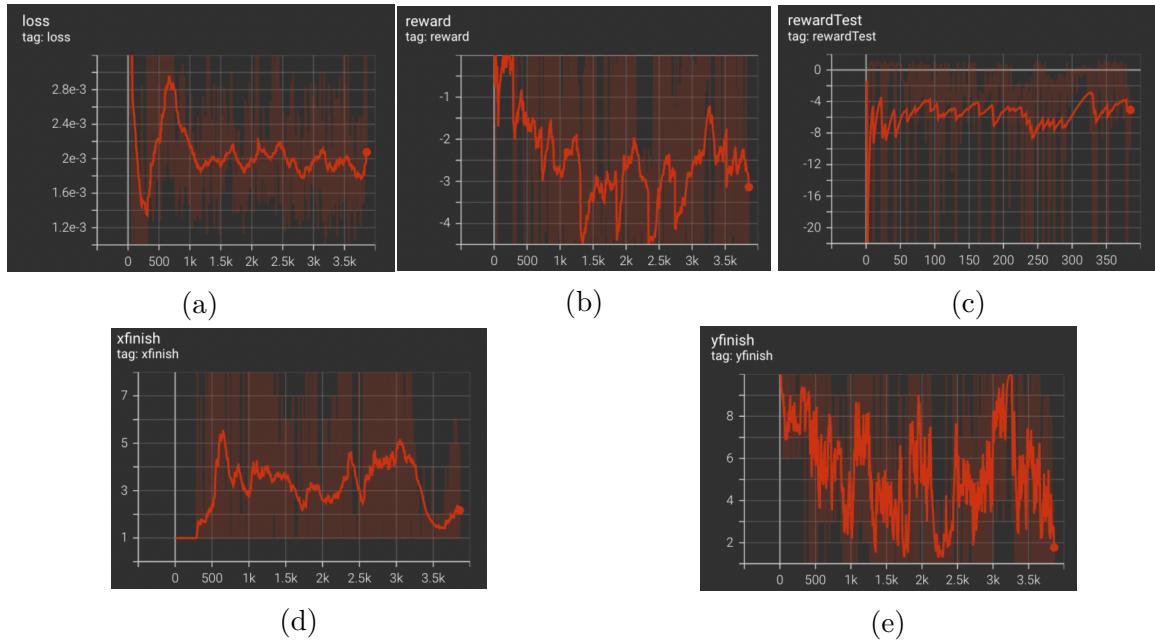


FIGURE 43 – De gauche à droite et de haut en bas : Loss, reward en entraînement et en test obtenus, postions finales de l'agent après 3500 itérations de goal sampling

Bien que l'algorithme réussisse à trouver la solution optimale afin d'atteindre l'état finale, on note une certaine instabilité, dûe au fait qu'on échantillonne toujours avec la même probabilité de faux buts. De ce fait, nous avons réalisé une expérience supplémentaire en ajoutant un hyper paramètre de decay sur la probabilité de tirer un faux but. Ainsi, quand l'algorithme a suffisamment exploré la carte grâce à son échantillonnage de buts, on tire plus fréquemment le vrai but de la carte. En positionnant ce paramètre à 0.99, on obtient le résultat 44a à 44f.

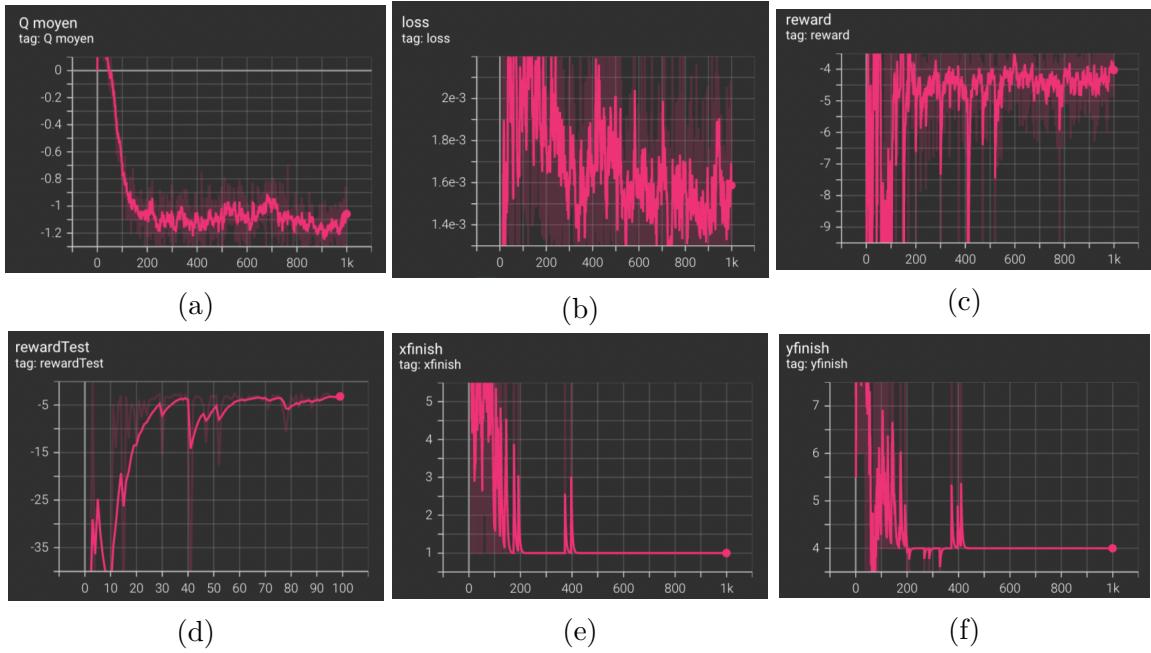


FIGURE 44 – De gauche à droite et de haut en bas : valeur Q moyennes, reward en entraînement et en test, loss, position finale en x et y obtenus après 1000 itérations de Goal Sampling avec un decay sur  $\beta$  de 0.99.

Les résultats obtenus sont bien plus stables, puisque le but échantilloné est désormais plus souvent le but voulu. On note la convergence vers l'état (1, 4), très proche de l'état de départ (1, 6), mais qui nécessite tout de même de parcourir toute la carte.

## 6 TP14 : Flow

Ce dernier TP a pour but d'étudier les modèles de flots. Il s'agit de la troisième méthode de modèles génératifs étudiée. Dans ce modèle, on cherche à apprendre une transformation entre une distribution latente et celle de nos données. Seulement, plusieurs hypothèses sont faites sur cette transformation, et en particulier, on suppose qu'elle n'est constituée que de fonctions inversibles. Cela permet d'obtenir la transformation inverse en utilisant la formule de changement de variables. Cette méthode permet d'optimiser directement la vraisemblance, contrairement aux GAN (voir section 1) et aux VAE (voir section 2).

La vraisemblance du modèle s'écrit alors :

$$L(\Theta) = \sum_{i=1}^N \log(p_{data}(x^i, \theta)) \quad (1)$$

$$= \sum_{i=1}^N p_z(f^{-1}(x^i, \theta), \phi) + \log \left| \det\left(\frac{\partial f^{-1}(x^i, \theta)}{\partial x}\right) \right| \quad (2)$$

$$= \sum_{i=1}^N p_z(f^{-1}(x^i, \theta), \phi) - \log \left| \det\left(\frac{\partial f(x^i, \theta)}{\partial x}\right) \right| \quad (3)$$

où  $\phi$  représente les paramètres de  $p_z$ , le loi de l'espace latent,  $\theta$  les paramètres de  $f$ , les  $x^i$  sont nos données et suivent la loi  $p_{data}$ . Dans la suite, on note  $\det\left(\frac{\partial f(x^i, \theta)}{\partial x}\right) = J_f$ .

### 6.1 Transformation affine

Dans ce premier exercice, on cherche à apprendre une transformation affine entre 2 distribution normales i.e. :  $\mathcal{N}(0, 1) \xrightarrow{f} \mathcal{N}(\mu, \sigma^2)$ .

#### 6.1.1 Calculs préliminaire et code

La transformation à apprendre est la suivante :  $f = x \odot \exp s + t$ . On cherche alors à estimer les paramètres  $s$  et  $t$  du modèle. On calcule analytiquement  $f$  et son inverse  $f^{-1}$  ainsi que leurs log-déterminants, utiles au calcul de la vraisemblance. On utilise des log-déterminants et donc  $\exp s$  dans  $f$ , dans un soucis de stabilité. En effet, on a :

$$\begin{aligned} f(x) &= x \odot \exp s + t \\ f^{-1}(y) &= (y - t) \odot \exp -s \\ \log |\det J_f| &= \log \left| \prod_{i=1}^d \exp s_i \right| = \sum_{i=1}^d s_i \\ \log |\det J_f^{-1}| &= \log \left| \prod_{i=1}^d \exp -s_i \right| = - \sum_{i=1}^d s_i \end{aligned}$$

On définit alors une classe **Affine** contenant le modèle (45).

```

class Affine(FlowModule):
    def __init__(self, d) → None:
        super().__init__()
        self.s = nn.Parameter(torch.ones(d,
            requires_grad=True))
        self.t = nn.Parameter(torch.zeros(d,
            requires_grad=True))

    def f(self, x):
        fx = torch.exp(self.s) * x + self.t
        detJf = (self.s).sum()
        return fx, torch.abs(detJf)

    def invf(self, y):
        finvx = (y - self.t) * torch.exp(- self.s)
        detinvJf = -(self.s).sum()
        return finvx, detinvJf

```

FIGURE 45 – Classe affine d'un modèle de flots

Ainsi, pour apprendre notre transformation à partir de données  $x_i$ , il suffit d'appliquer la transformation inverse  $f^{-1}$  afin de revenir dans l'espace latent, d'estimer la log-vraisemblance, ainsi que les log-probailités de la distribution latente. Cela nous mène à la boucle d'apprentissage en figure (46).

```

for epoch in range(epochs):
    # sample data from start dist
    xs = target_dist.sample(torch.Size([batch_size]))
    # compute transformation f(x) and prob p(f(x))
    # (ie from target dist in model) and log/det df(x)
    /dx/
    log_prob, _, Jf = model.invf(xs)
    optim.zero_grad()
    # compute loss and optimize
    negliklh = - (log_prob + Jf).mean()
    negliklh.backward()
    optim.step()

```

FIGURE 46 – Boucle d'apprentissage d'un modèle de flots

### 6.1.2 Résultats expérimentaux

Pour cet exemple jouet, et les paramètres suivants :

- une loi en dimension 3 de paramètres  $\mu = \begin{pmatrix} 0 \\ 10 \\ 3 \end{pmatrix}$ ,  $\sigma^2 = \begin{pmatrix} 1 \\ 5 \\ 10 \end{pmatrix}$
- 10000 epochs

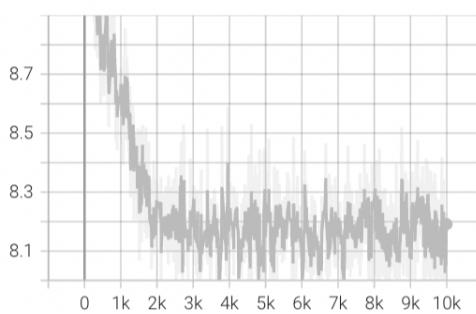
- une loi dans l'espace latent  $\mathcal{N}(0, I_3)$  en dimension 3
- un taille de batch de 100
- une optimisation à chaque epoch avec l'optimiseur Adam et un lr de 0.01

On obtient les résultats présentées sur les figures 47a et 48a.

```
estimated sigma tensor([0.9976, 5.1017, 9.7056], grad_fn=<ExpBackward>) true value tensor([ 1.,  5., 10.])
estimated mu Parameter containing:
tensor([-0.0177, 9.8946, 2.9182], requires_grad=True) true value tensor([ 0., 10.,  3.])
```

(a)

FIGURE 47 – Paramètres obtenus après 10000 itérations.



(a)

FIGURE 48 – loss obtenue après 10000 itérations.

La vraisemblance négative se stabilise autour d'une valeur de 8.2. De plus, avec les 10000 epochs, on note que les valeurs visées sont correctement retrouvées pour une précision de l'ordre de  $10^{-2}$ .

## 6.2 Modèle GLOW

Le modèle GLOW, est composé de 3 blocks alternés : une transformation affine (normalisé à la première étape), une convolution  $1 \times 1$  et une coupling layer.

### 6.2.1 Calculs préliminaires

Une coupling layer consiste en : l'identité pour la première moitié du vecteur de sortie et une transformation affine appliquée sur la première moitié du vecteur d'entrée, pour la seconde moitié du vecteur de sortie, c'est-à-dire que, pour  $x = (x_1 \dots x_d, x_{d+1} \dots x_{2d})$ , on a :

$$\begin{aligned}
f(x)_{1:d} &= x_{1:d} \\
f(x)_{d+1:2d} &= x_{d+1:2d} \odot \exp s_\theta(x_{1:d}) + t_\phi(x_{1:d}) \\
f(y)_{1:d} &= y_{1:d} \\
f(y)_{d+1:2d} &= (y_{d+1:2d} - t_\phi(x_{1:d})) \odot \exp -s_\theta(x_{1:d}) \\
\log |\det J_f| &= \log \left| \prod_{i=1}^d \exp s_i \right| = \sum_{i=1}^d s_i \\
\log |\det J_f^{-1}| &= \log \left| \prod_{i=1}^d \exp -s_i \right| = - \sum_{i=1}^d s_i
\end{aligned}$$

Le module de convolution  $1 \times 1$  suit les équations suivantes, en utilisant la décomposition  $LU$  de la matrice  $W$  :

$$\begin{aligned}
f(x) &= Wx \\
f(y)_{1:d} &= W^{-1}y \\
W &= P(L' + I_d)(U' + \text{diag}(S)) \\
\log |\det J_f| &= \sum_{i=1}^d \log |s_i|
\end{aligned}$$

### 6.2.2 Expérimentations

En utilisant une structure de code similaire à celle utilisée dans le modèle affine, un modèle GLOW est implémenté. Nous cherchons à l'optimiser afin de trouver la fonction entre une loi Normale et les données présentées en figures 49a et 49b.

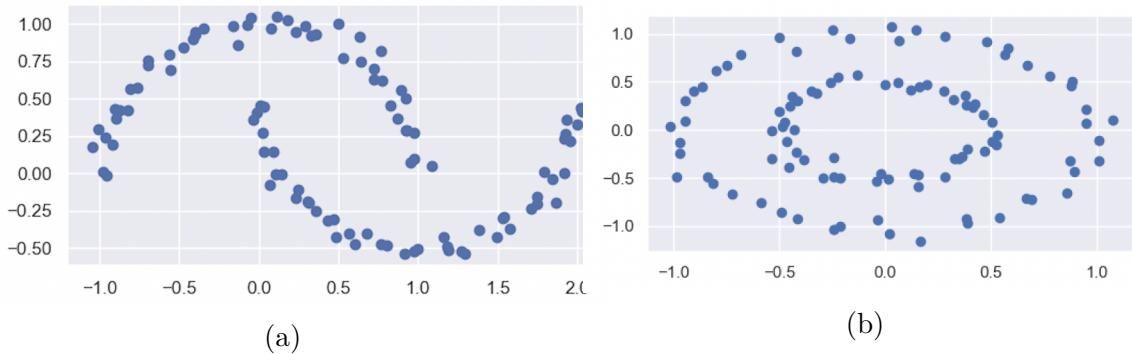
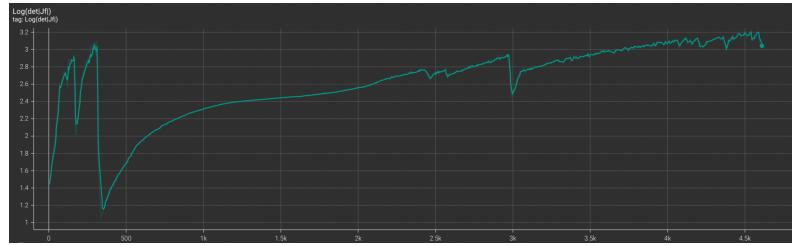


FIGURE 49 – Dataset utilisés pour optimiser le modèles GLOW

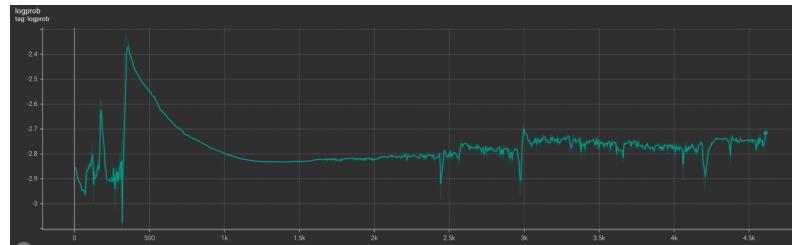
Sur le jeu de données `moons`, on obtient les résultats présentés en figure 50a à 51c. Pour cela, on a utilisé les paramètres suivants :

- 100 données sont générées à chaque epochs. Environ 4600 epochs ont été exécutées.

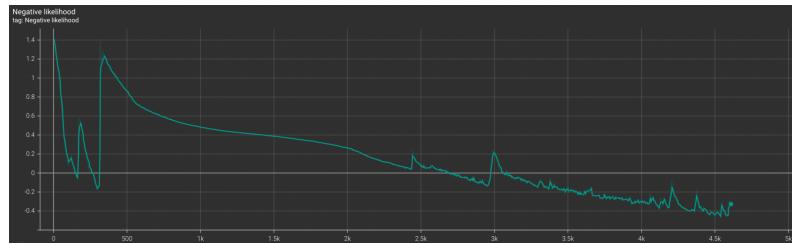
- On optimise le modèle à l'aide de l'algorithme Adam, et un learning rate de 0.0001
- 10 modules GLOW sont alternés, avec une dimensions de 2 (celles des données) pour les couches `actnorm`, `conv1x1` et un espace de dimension latente de taille 100 pour la `coupling layer` (entrée et sortie de taille 2)



(a)



(b)



(c)

FIGURE 50 – Log-déterminant, log-probabilités et loss pour notre entraînement de GLOW sur le dataset moons (de haut en bas).

En utilisant des paramètres similaires sur le jeu de données `circles`, on obtient les résultats 52a à 53c.

On note que la reconstruction est difficile dans les deux cas, bien qu'on observe que la structure des données tend à avoir la forme souhaitée. Avec plus d'entraînement, le modèle réussirait sûrement à capturer la forme manquante du jeu de données. Notons que lors de nos différentes expériences, nous avons rencontré de nombreux problèmes d'instabilités sur nos modèles (de part la présence de log et d'une matrice à inverser), ce qui avait pour effet d'introduire des Nans dans les valeurs.

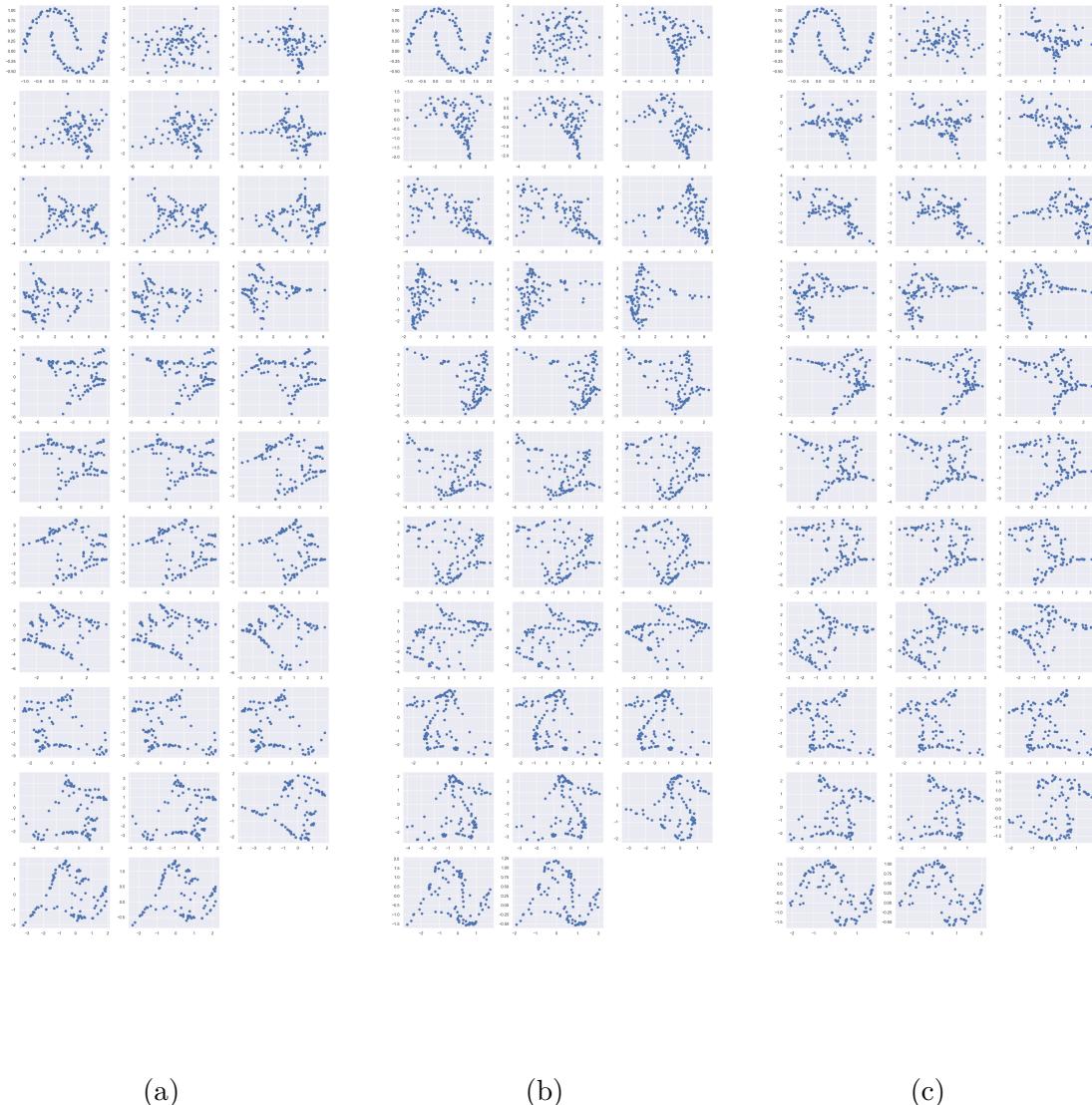


FIGURE 51 – Résultat obtenus, après chacune des couches de GLOW, après 300, 1200 et 4600 epochs (de gauche à droite).

Sur chaque image : En haut à gauche : dataset d'origine puis de gauche à droite et de haut en bas : résultats après les couches successives du modèles.

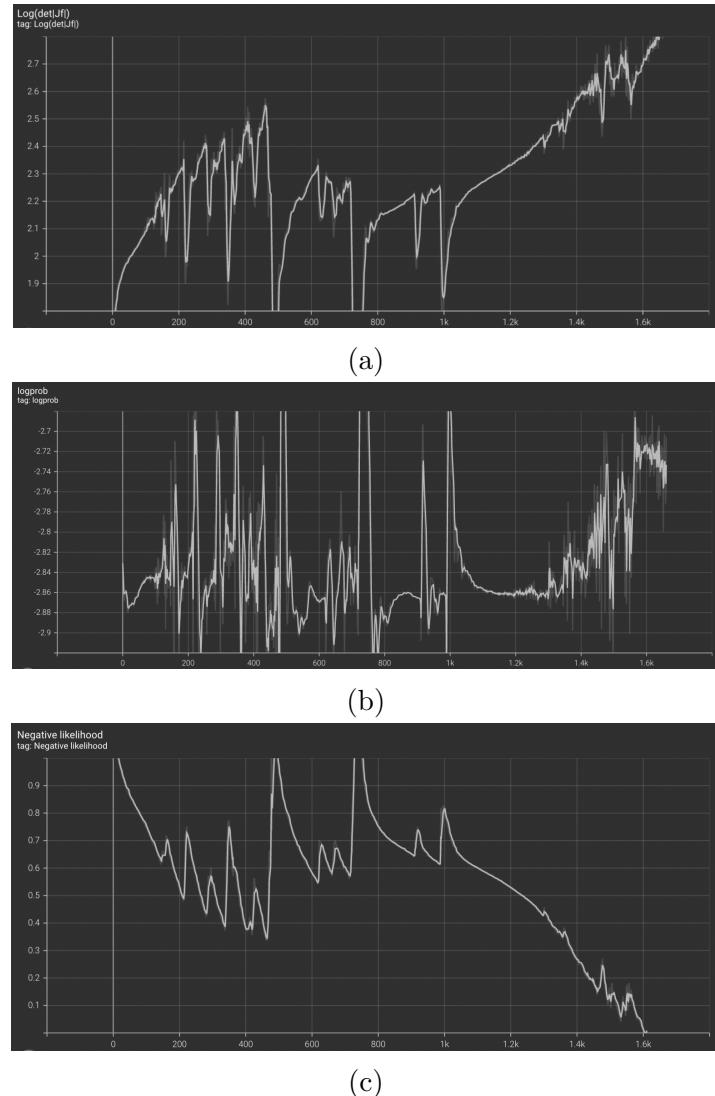


FIGURE 52 – Log-déterminant, log-probabilités et loss pour notre entraînement de GLOW sur le dataset circles (de haut en bas).

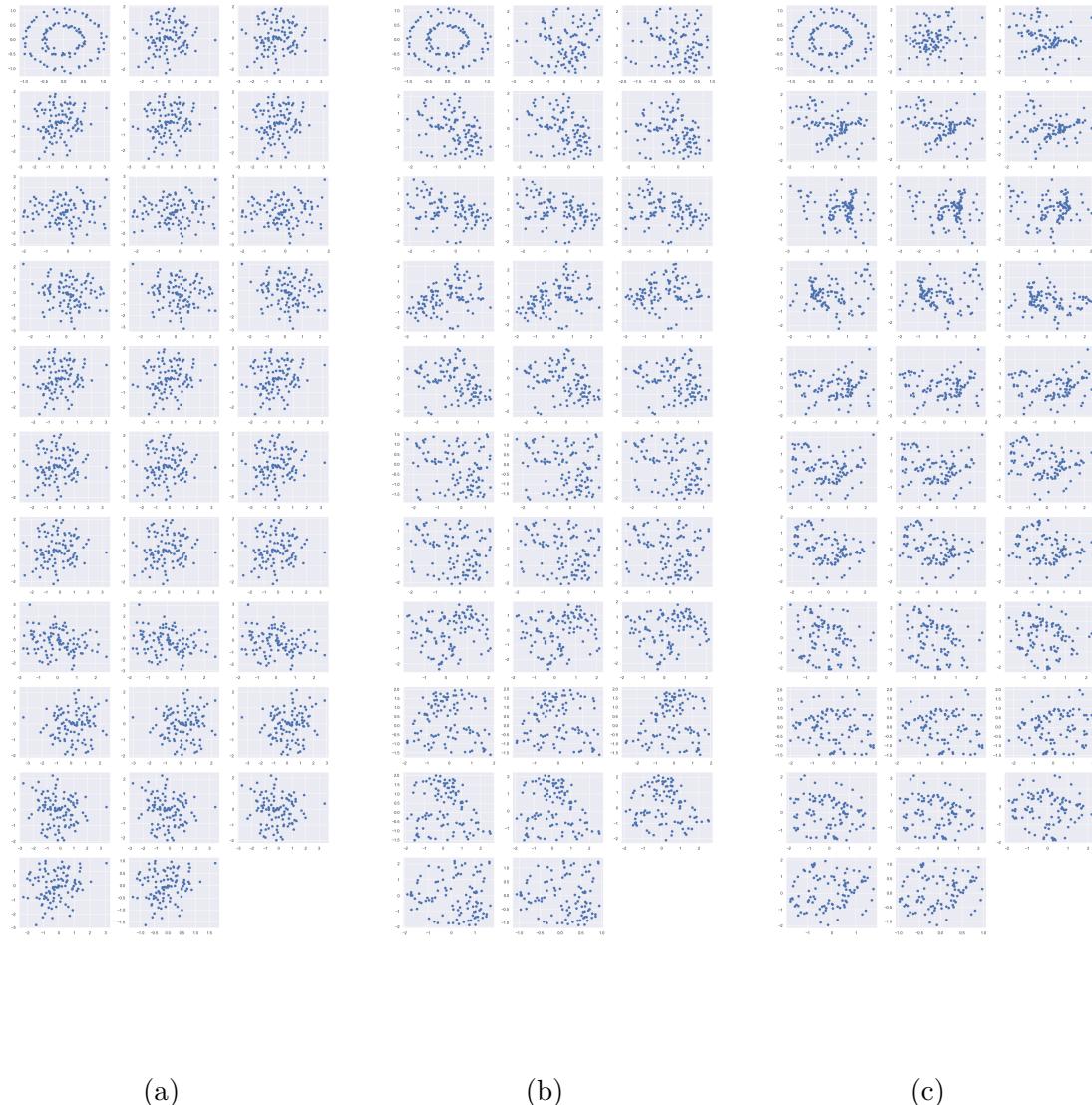


FIGURE 53 – Résultat obtenus, après chacune des couches de GLOW, après 0, 500 et 1500 epochs (de gauche à droite).

Sur chaque image : En haut à gauche : dataset d'origine puis de gauche à droite et de haut en bas : résultats après les couches successives du modèles.

## Conclusion

Ce second rapport présente différents résultats expérimentaux d'algorithmes d'apprentissage par renforcement ou d'apprentissage profond. Il se divise en 2 catégories d'algorithmes : les modèles génératifs (GAN, VAE et modèles de flots) et des méthodes d'apprentissage par renforcements de l'état de l'art (Multi-agents, apprentissage par imitation et curriculum learning).

La principale difficulté rencontrée dans ces TP est la recherche des bons hyper-paramètres permettant de faire apprendre à l'agent une politique de jeu intéressante est ardue et longue car elle nécessite de tester de nombreuses combinaisons d'hyper-paramètres et d'effectuer des entraînements, parfois très lents. En effet, lorsqu'un agent ne réussit pas à apprendre à résoudre sa tâche, il est difficile de savoir s'il n'apprend pas car l'implémentation n'est pas correcte ou alors si ce sont les hyper-paramètres qui ne sont pas bons. Ce fut principalement le cas pour l'algorithme GAIL (4) qui apprend une politique encore éloignée de celle de l'expert qui le guide. Une seconde difficulté fut celle du temps d'apprentissage et des ressources informatiques, notamment pour entraîner les GAN (1). En effet, le jeu de donnée étant très conséquent, l'entraînement est très long et nécessite une puissance de calcul non négligeable.