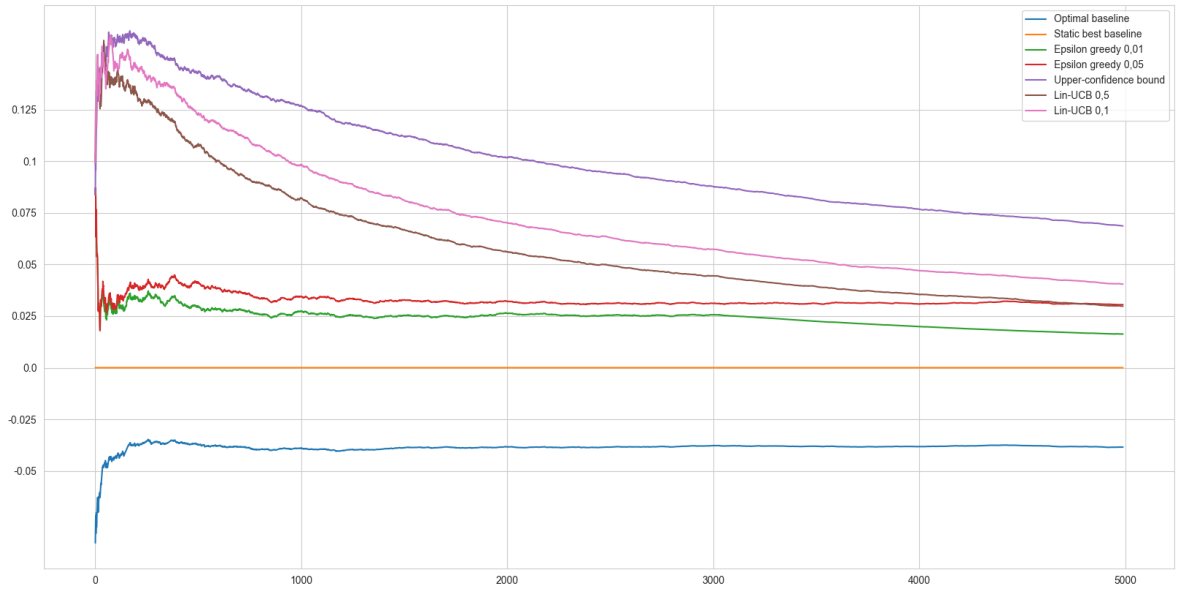
TME 1 — Problèmes de bandits

Victor Duthoit, Pierre Wan-Fat

Dans ce TME, on expérimente plusieurs stratégies afin de résoudre un problème de bandits à plusieurs bras. Voici le regret moyen cumulé :



On constate que l'algorithme UCB a des moins bonnes performances que l'algorithme Lin-UCB, preuve que le contexte est utile dans ce problème. En revanche, la stratégie *epsilon greedy* reste plus efficace que Lin-UCB.

TME 2 - MPD- Victor Duthoit

Imports

```
import gym
import gridworld
import randomAgent
import matplotlib
import matplotlib.pyplot as plt
# matplotlib.use("Qt5agg")
%matplotlib inline
import time
import numpy as np
import operator
import progressbar
import pandas as pd
```

Utilities

```
def mse(a,b):
    #return the norm 2 (mean square error) of to functions
    return np.sqrt(sum( (a[x]-b[x])**2 for x in a ))

def dic_argmax(d):
    #return the argmax for dictionnaries
    return max(d.items(), key=operator.itemgetter(1))[0]

pI, vI, rD = 'policy_iteration','value_iteration','random'
```

Policies algorithms

1. Policy iteration

```
def policyIteration(states, P, eps=0.01, gamma=0.5):
    #return the policy following the policy iteration algorithm
    final_states = set(states)-set(P)
    pi = {s : np.random.randint(4) for s in P } #initialize policy
    #while the policy is unstable
    while True :
        v = {s : 0 for s in states } #initialize the value
        while True : #while value is not stable
            #update of v by the Bellman equation
            v_new = {s : sum( p*(rew + gamma*v[s_p]) for p,s_p,rew,done in P[s]
[pi[s]]) for s in P }
            for f in final_states:
                v_new[f] = 0
            if mse(v,v_new)<eps:
```

```
                    v = v_new
                    break
                v = v_new
            pi_new = {}
            for s in P:
                #choose the action with the highest value
                pi_action = {a : sum( p*(rew+gamma*v[s_p]) for p,s_p,rew,done in P[s]
[a] ) for a in P[s]}
                pi_new[s] = dic_argmax(pi_action)
            if mse(pi,pi_new)==0:
                break
            pi = pi_new
        return pi
```

2. Value iteration

```
def valueIteration(states, P, eps=0.01, gamma=0.5):
    #return the policy following the value iteration algorithm
    final_states = set(states)-set(P)
    v = {s : 0 for s in states } #initialize the value
    while True : #while value is not stable
        v_new = {s: max([sum( p*(rew + gamma*v[s_p]) for p,s_p,rew,done in P[s]
[a]) for a in P[s]]) for s in P }
        for f in final_states:
            v_new[f] = 0
        if mse(v,v_new)<eps:
            v = v_new
            break
        v = v_new
    pi = {s: dic_argmax(
        {a : sum( p*(rew + gamma*v[s_p]) for p,s_p,rew,done in P[s][a]) for a in
P[s] }
    ) for s in P }
    return pi
```

Agent

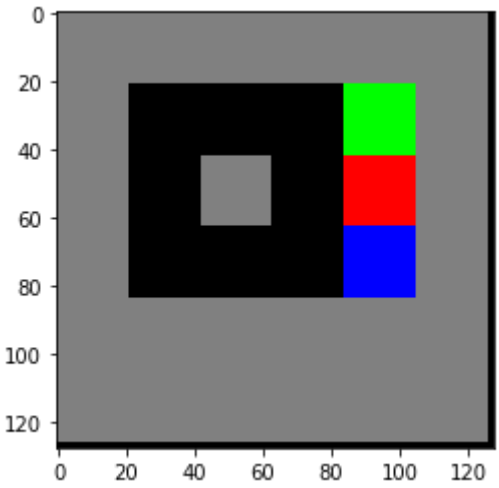
Nous définissions des agents qui ont la particularité d'avoir des politiques déterministes à définir.

```
class Agent(randomAgent.RandomAgent):
    #an agent pocesses a policy pi

    def __init__(self, action_space):
        super().__init__(action_space)
        self.pi = dict()

    def act(self, observation, reward, done):
        return self.pi[observation]
```

Environnement - plan0

```
def createEnv(plan='0', visual=False):
```

```
    # matplotlib.use("Qt5agg")
    env = gym.make("gridworld-v0")
    env.setPlan(f"gridworldPlans/plan{plan}.txt", {0: -0.001, 3: 1, 4: 1, 5: -1, 6: -1})
    states, P = env.getMDP()
    env.seed(0)  # Initialise le seed du pseudo-random
    # print(env.action_space)  # Quelles sont les actions possibles
    # print(env.step(1))  # faire action 1 et retourne l'observation, le reward, et un done un booleen (jeu fini ou pas)
    env.render()  # permet de visualiser la grille du jeu
#     env.render(mode="human") #visualisation sur la console
    statedic, mdp = env.getMDP()  # recupere le mdp : statedic
#     print("Nombre d'etats : ",len(statedic))  # nombre d'etats ,statedic : etat-> numero de l'etat
    state, transitions = list(mdp.items())[0]
    return env
    # print(state)  # un etat du mdp
    # print(transitions)  # dictionnaire des transitions pour l'etat :  {action-> [proba,etat,reward,done]}

createEnv()
```




```
<gridworld.gridworld_env.GridworldEnv at 0x1a30578828>
```

Test agent

On définit deux fonctions de test. L'une pour tester un agent dans un environnement connu, l'autre pour tester tous les paramètres en entrée.

```
def testAgent(agent, env, episode_count=1, visual=False, freq=100, FPS=0.01):
    #test an agent for a given environment
    if visual:
        matplotlib.use('Qt5Agg')
    reward = 0
    done = False
    rsum = 0
    R = 0
```

```
    actions = 0
    for i in range(episode_count):
        obs = env.reset()
        if visual:
            print(obs)
        env.verbose = (i % freq == 0 and i > 0)  # afficher 1 episode sur 100
        if env.verbose and visual:
            env.render(FPS)
        j = 0
        rsum = 0
        while True:
            action = agent.act(gridworld.GridworldEnv.state2str(obs), reward, done)
            obs, reward, done, _ = env.step(action)
            rsum += reward
            j += 1
            if env.verbose and visual:
                env.render(FPS)
            if done:
                if visual:
                    print("Episode : " + str(i) + " rsum=" + str(rsum) + ", " + str(j) + " actions")
                R+= rsum
                actions+= j
                break
    avg_rsum = R/episode_count
    avg_action = actions/episode_count
    env.close()
    if visual :
        %matplotlib inline
    return [avg_rsum, avg_action]
```

```
def testSeries(plan = ['0'],
               agents = [pI,vI,rD],
               epsilon = [0.1],
               gamma = [0.1],
               episode_count = [100]):
    #tests all the agents for given parameters
    res = []
    N = len(plan)*len(agents)*len(epsilon)*len(gamma)*len(episode_count)
    i=0

    bar = progressbar.ProgressBar(maxval=N, \
    widgets=[progressbar.Bar('=', '[', ']'), ' ', progressbar.Percentage()])
    bar.start()
    for p in plan:
        env = gym.make("gridworld-v0")
        env.setPlan(f"gridworldPlans/plan{p}.txt", {0: -0.001, 3: 1, 4: 1, 5: -1, 6: -1})
        states, P = env.getMDP()
        agent = Agent(env.action_space)
        for eps in epsilon:
            for g in gamma:
                for ec in episode_count:
                    for ag in agents:
                        if ag==rD:
                            agent = randomAgent.RandomAgent(env.action_space)
```

```
                            t_pi = 0.
                        else:
                            t1_pi = time.time()
                            if ag==pI:
                                policy = policyIteration(states, P, eps=eps,
gamma=g)
                            if ag==vI:
                                policy = valueIteration(states, P, eps=eps,
gamma=g)
                            t_pi = time.time()-t1_pi
                            agent.pi = policy
                        t1_test = time.time()
                        test = testAgent(agent,env, episode_count = ec)
                        t_test = time.time()-t1_test
                        res.append([p,eps,g,ec,ag]+test+[t_pi,t_test])
                        i+=1
                        bar.update(i)
    bar.finish()
    columns =
['plan','eps','gamma','episode_count','agent','rsum','action','t_pi','t_test']
    return pd.DataFrame(res,columns=columns)
```

```
def plotSeries(df, xaxis='gamma',yaxis='rsum',legend='agent',legendValue=
[pI,vI],bar=False):
    plot = ()
    for legendV in legendValue:
        subdf = df[df[legend]==legendV]
        x = subdf[xaxis].to_numpy()
        y = subdf[yaxis].to_numpy()
        plot+=(x,y)
    if bar:
        plt.bar(*plot)
    else:
        plt.plot(*plot)
    plt.ylabel(yaxis)
    plt.xlabel(xaxis)
    plt.legend(legendValue)
```

```
# Execution avec un Agent

# agent = randomAgent.RandomAgent(env.action_space)
eps = 0.1
gamma = 0.1
policy = policyIteration(states, P, eps=eps, gamma=gamma)
# policy = valueIteration(states, P,eps=eps, gamma=gamma)
agent = Agent(env.action_space)
agent.pi = policy
testAgent(agent, env)
```

```
[0.979, 22.0]
```

Premier tests

Test for plan0, seed(0), 10000 episode, esp = 0.0001, gamma = 0.9 :

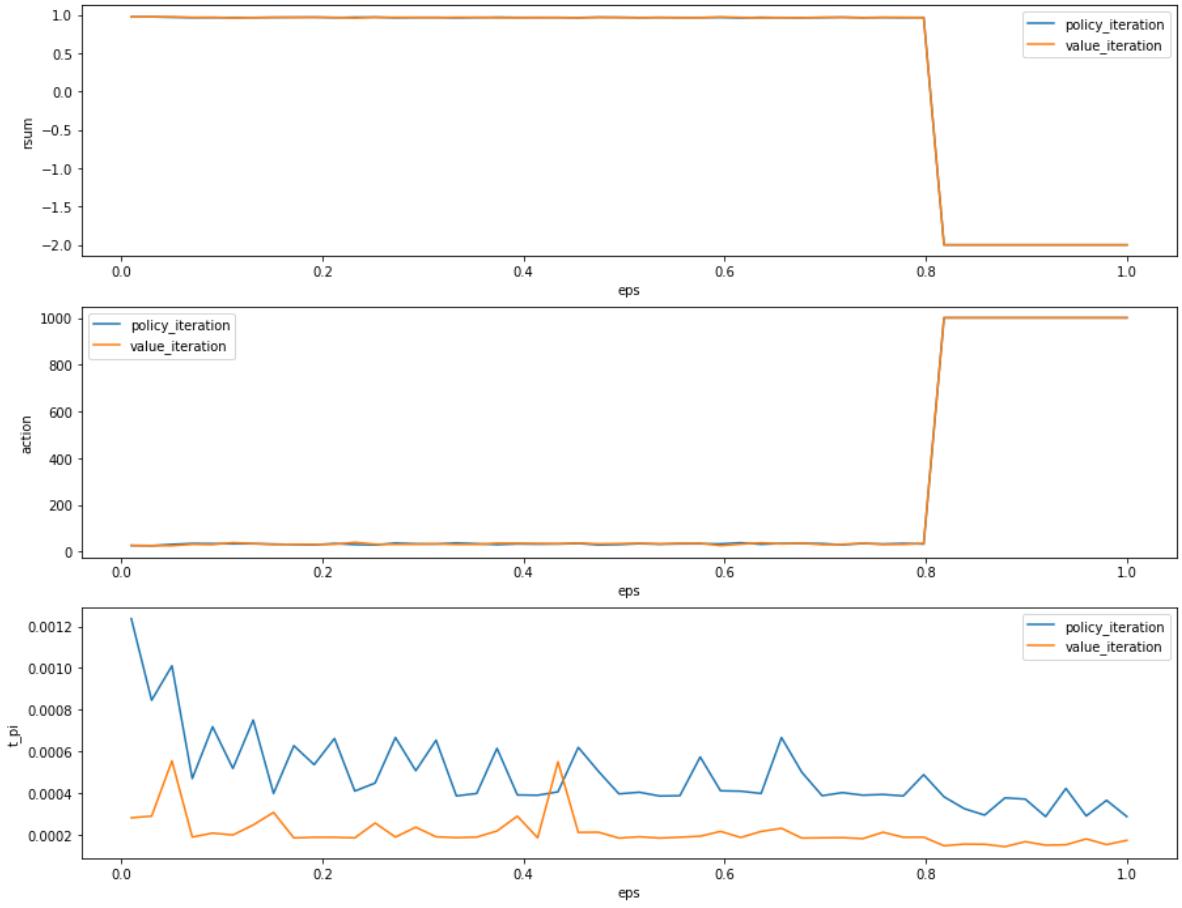
- **policy iteration agent:** average rsum : 0.56, average action : 4.79
- **value iteration:** average rsum : 0.55, average number of action : 4.80
- **random agent:** average rsum : -0.81, average action : 11.33

Les agents "politisés" sont bien meilleurs.

Choix d'epsilon

```
epsilon = np.linspace(0.01,1,num=50)
df_eps = testSeries(epsilon=epsilon,agents=[pI,vI]
```

```
#plots
plt.figure(figsize=(15,12))
plt.subplot(311)
plotSeries(df_eps, xaxis='eps', yaxis='rsum')
plt.subplot(312)
plotSeries(df_eps, xaxis='eps', yaxis='action')
plt.subplot(313)
plotSeries(df_eps, xaxis='eps', yaxis='t_pi') #t_pi is the time to determine the
policy
```



On remarque qu'un epsilon relativement élevé est suffisant pour une bonne détermination de la politique. On choisira ainsi un eps = 0.1 pour la suite des calculs. On remarque de plus que l'itération de la politique est plus longue à déterminer.

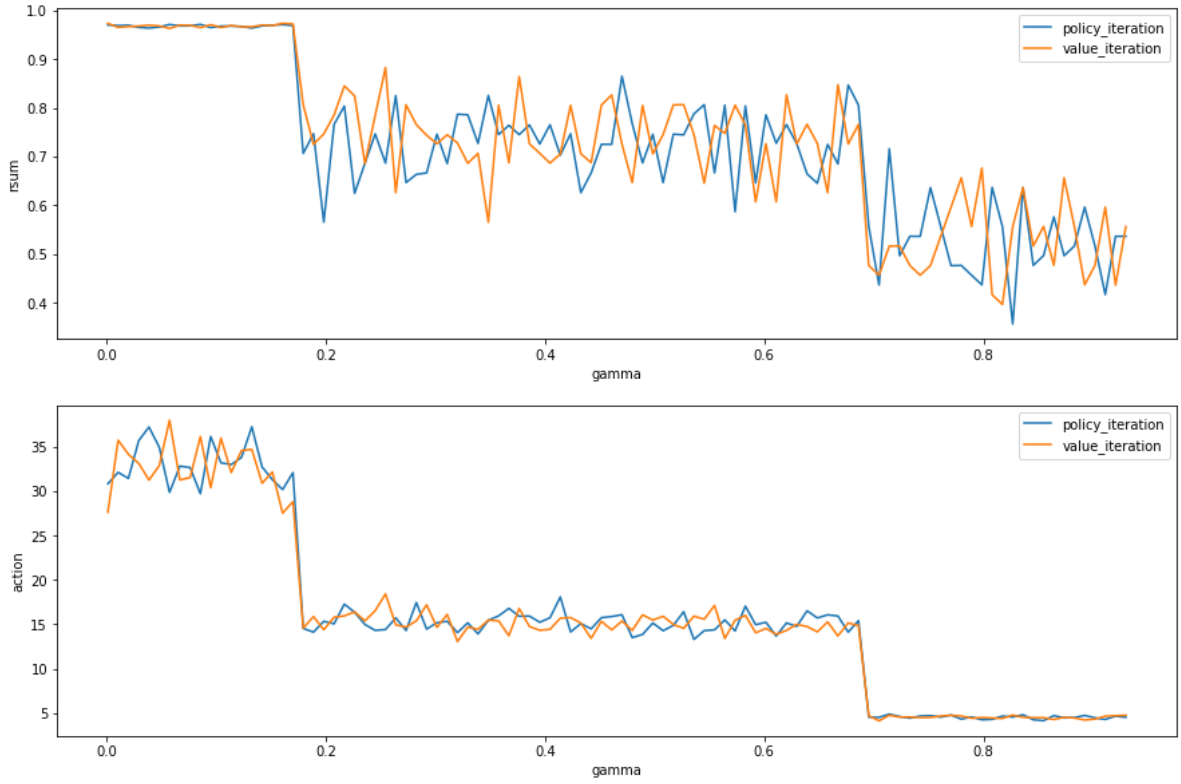
Rôle de gamma

```
gamma = np.linspace(0.001,0.93,num=100)
df_gamma = testSeries(gamma=gamma,agents=[pI,vI])
```

```
[=                                                                    ]   2%

0
```

```
[======================================================================] 100%
```

```
#plots
plt.figure(figsize=(15,10))
plt.subplot(211)
plotSeries(df_gamma, xaxis='gamma', yaxis='rsum')
plt.subplot(212)
plotSeries(df_gamma, xaxis='gamma', yaxis='action')
```

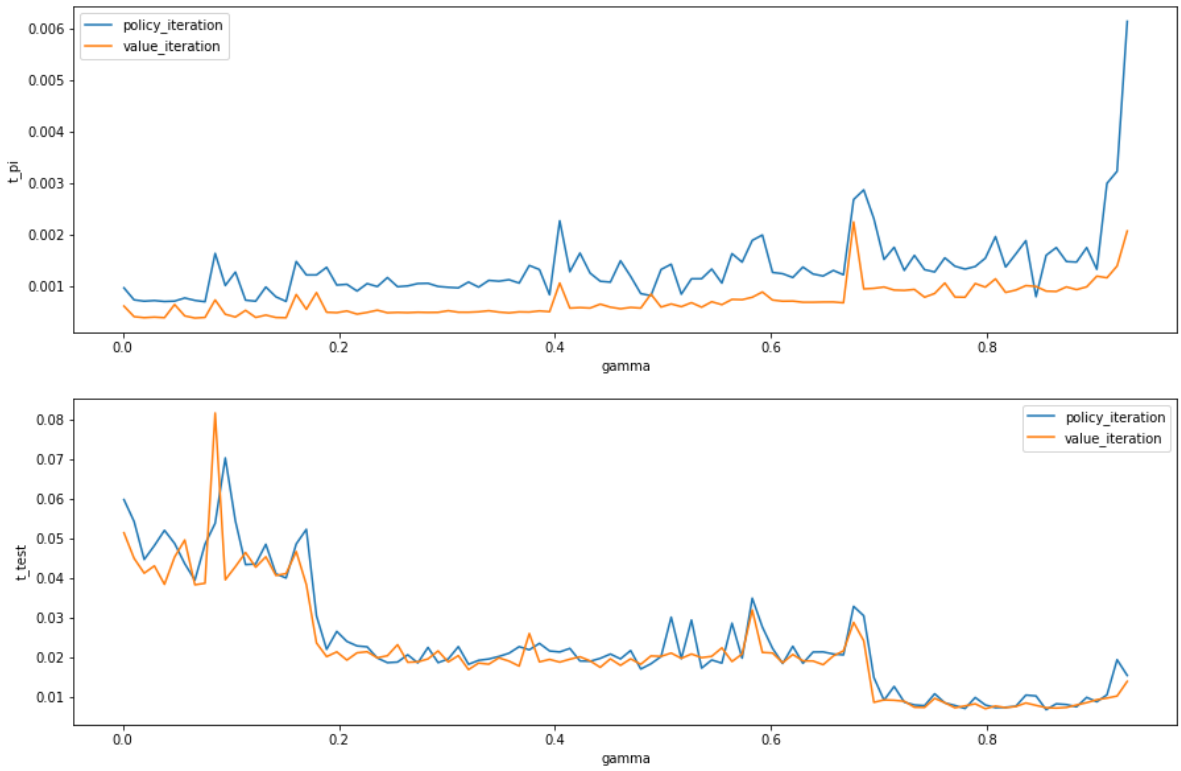


On peut noter deux points importants :

1. Les deux algorithmes semblent procurer des résultats similaires. On pourra ainsi se contenter de l'algorithme d'itération de la valeur.
2. Gamma joue un rôle essentiel dans le choix des stratégies de jeux. Apparemment, un gamma élevé privilégiera des gains accessibles rapidement, la partie s'écourtera ainsi plus rapidement. En revanche un gamma moindre a tendance à ne pas trop dévaloriser les gains accessibles en plus de coûts. Les paliers incitent à croire qu'un gain est possible en effectuant plus d'actions. Ainsi, cela donne à penser que le gamma dépend de la topologie du terrain.
3. L'optimal de reward est atteint pour des valeurs proches de 0 ou de 1. On retiendra la valeur 0.1 pour gamma (proche de 0) car le palier proche de 1 semble limité.

Gestion du temps

```
#plots
plt.figure(figsize=(15,10))
plt.subplot(211)
plotSeries(df_gamma, xaxis='gamma', yaxis='t_pi')
plt.subplot(212)
plotSeries(df_gamma, xaxis='gamma', yaxis='t_test')
```

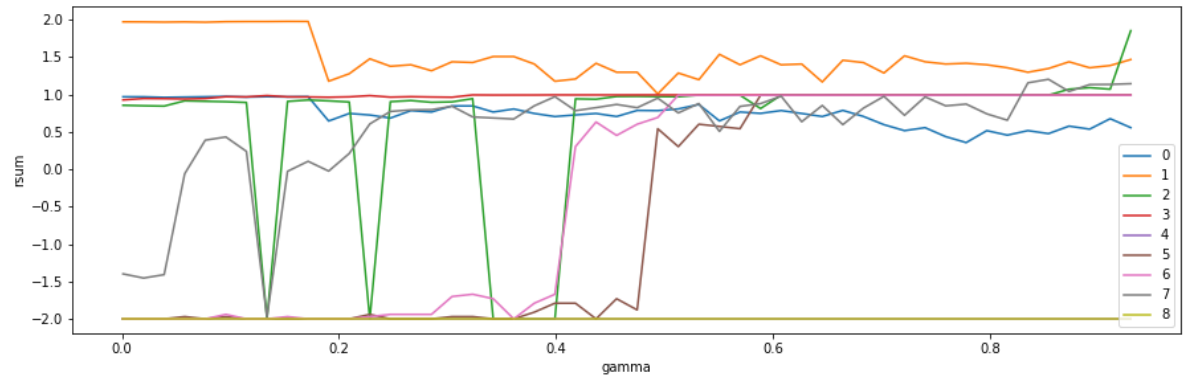


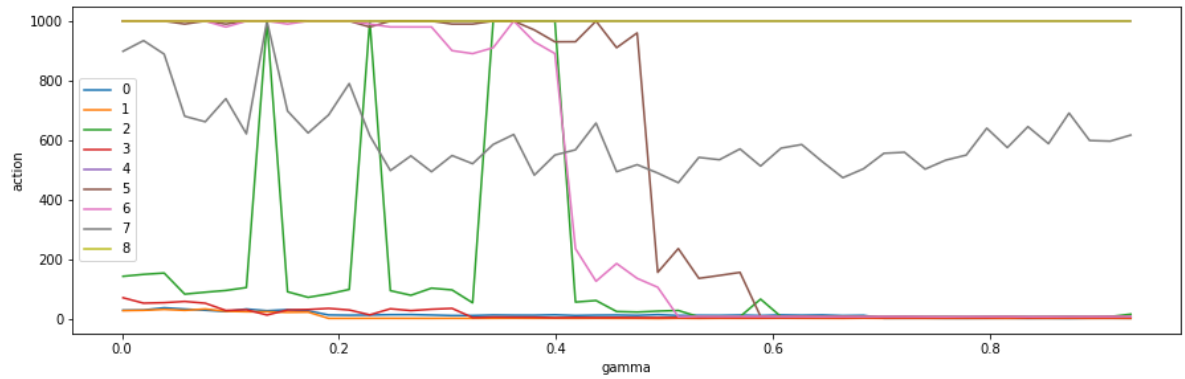
1. La première observation que l'on peut faire est que l'ordre de grandeur de calcul de la politique est très faible devant les temps de tests.
2. De plus, on remarque que l'algorithme d'itération de la valeur est plus rapide à terminer que son homologue d'itération de la politique.
3. Les temps de tests sont similaires pour les deux algos et ressemblent grossièrement à la distribution du nombre d'actions.

Sur de nouveaux terrains

```
gamma = np.linspace(0.001,0.93,num=50)
plan = list(range(9))
df_plan = testSeries(plan=plan, gamma=gamma,agents=[vI])
```

```
#plots
plt.figure(figsize=(15,10))
plt.subplot(211)
plotSeries(df_plan, xaxis='gamma', yaxis='rsum', legend='plan', legendValue=plan)
plt.subplot(212)
plotSeries(df_plan, xaxis='gamma', yaxis='action', legend='plan',
legendValue=plan)
```



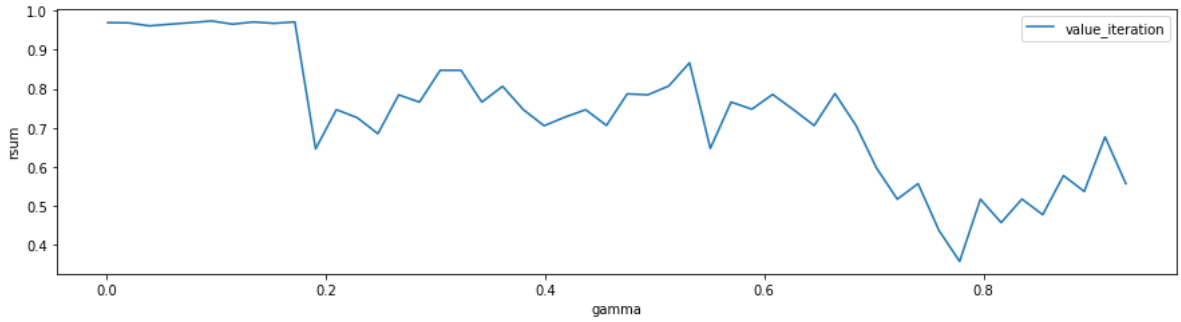


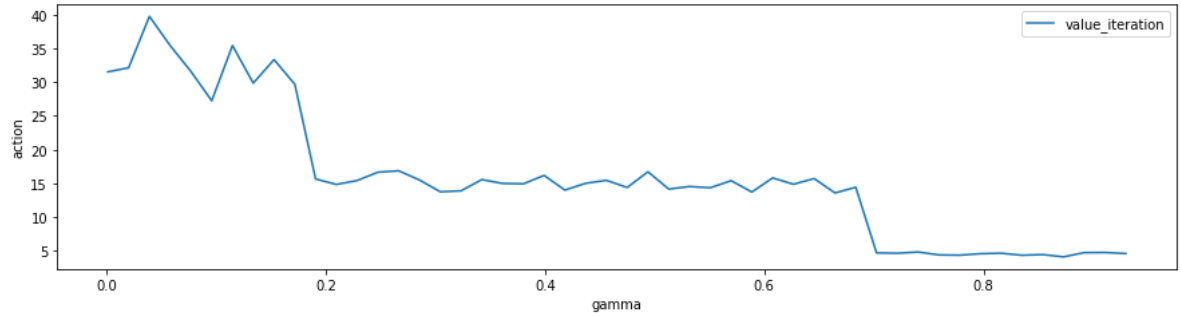
On remarque que les plans n'ont pas les mêmes ordres de grandeur concernant le nombre d'actions et les gains. Néanmoins, nous pouvons voir que le dépendance en fonction de gamma changent avec les plans. Voyons de plus près.

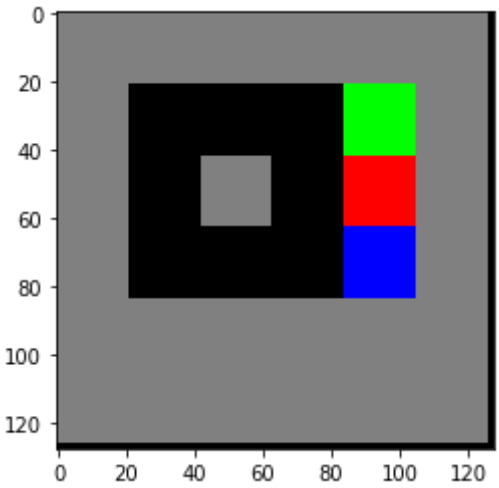
```
def plotPlan(df,p):
    sub_df = df[df['plan'] == p]
    plt.figure(figsize=(15,8))
    plt.subplot(211)
    plotSeries(sub_df, xaxis='gamma', yaxis='rsum', legendValue=[vI])
    plt.subplot(212)
    plotSeries(sub_df, xaxis='gamma', yaxis='action', legendValue=[vI])
```

Plan 0

```
plotPlan(df_plan,0)
createEnv(0)
```

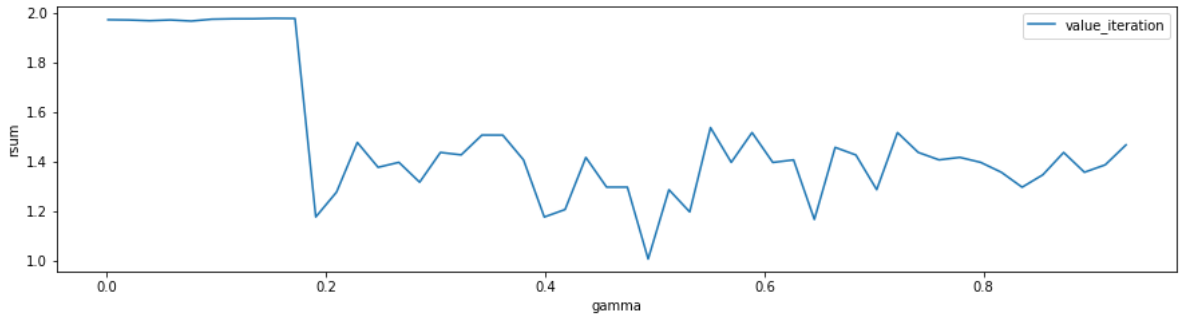


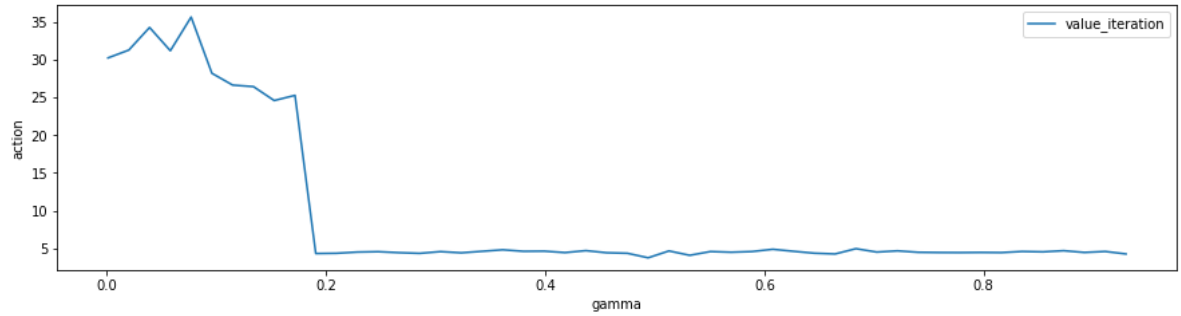


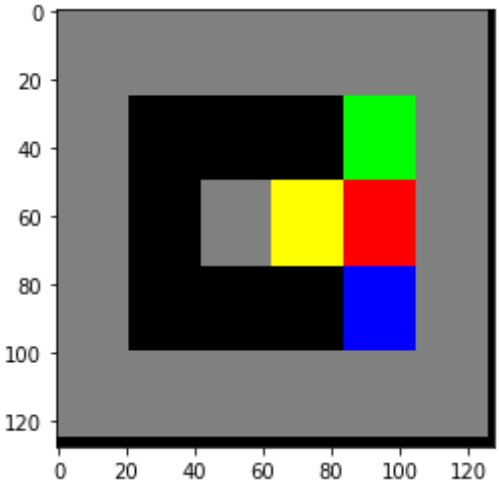


Plan 1

```
plotPlan(df_plan,1)
createEnv(1)
```

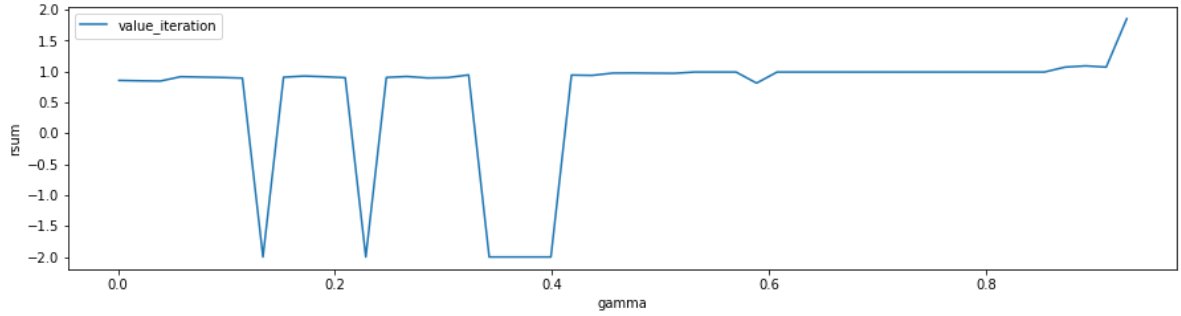


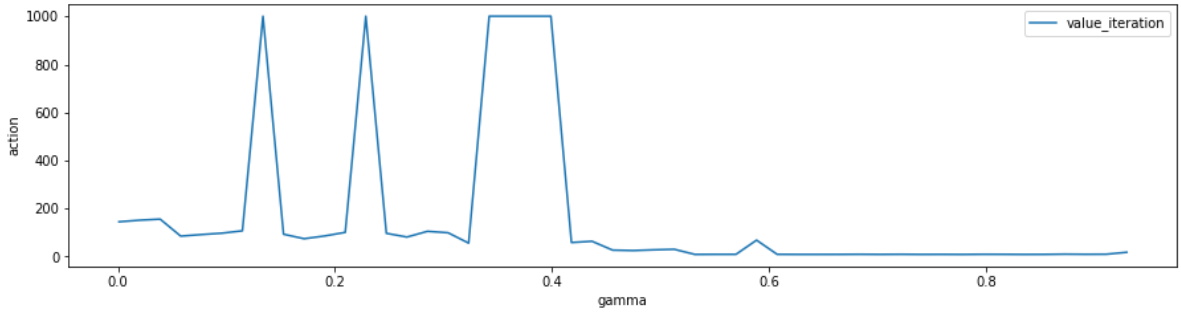


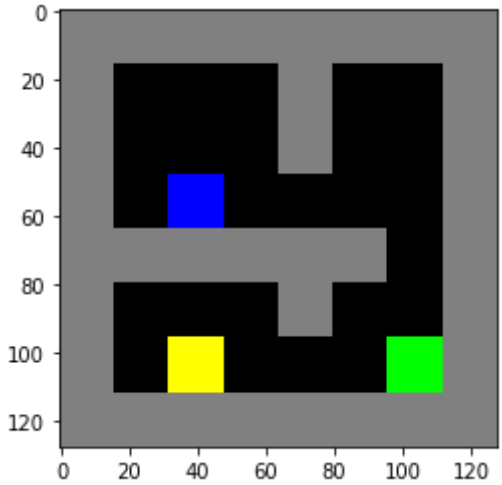


Plan 2

```
plotPlan(df_plan,2)
createEnv(2)
```

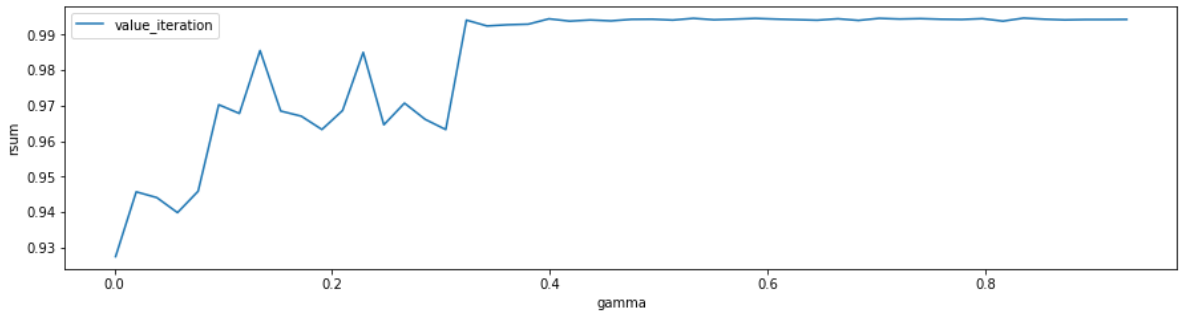


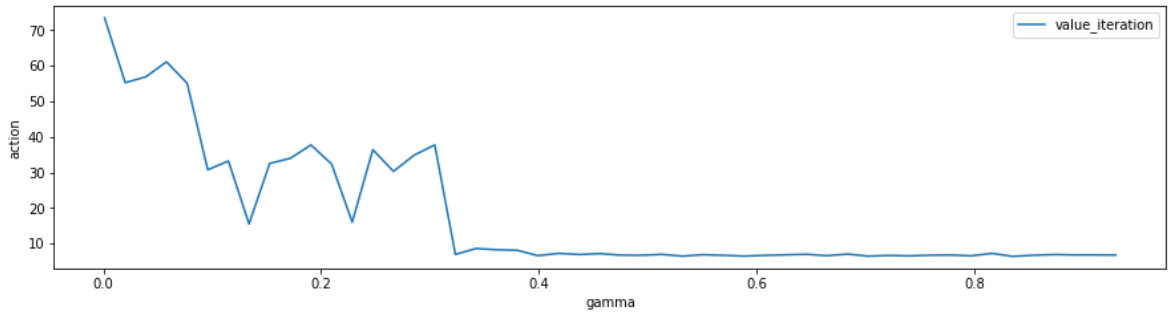


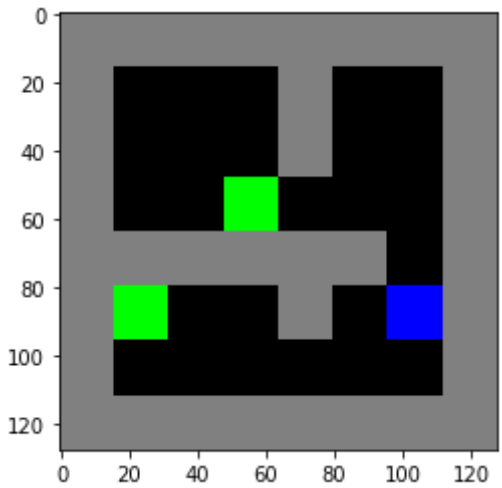


Plan 3

```
plotPlan(df_plan,3)
createEnv(3)
```

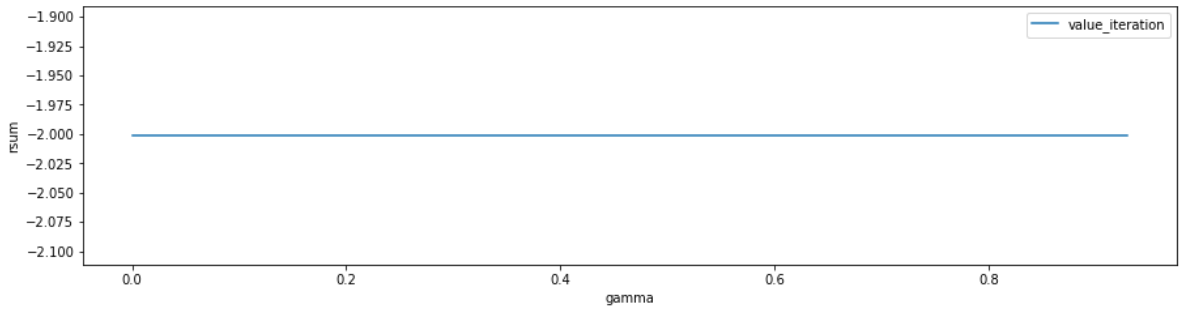


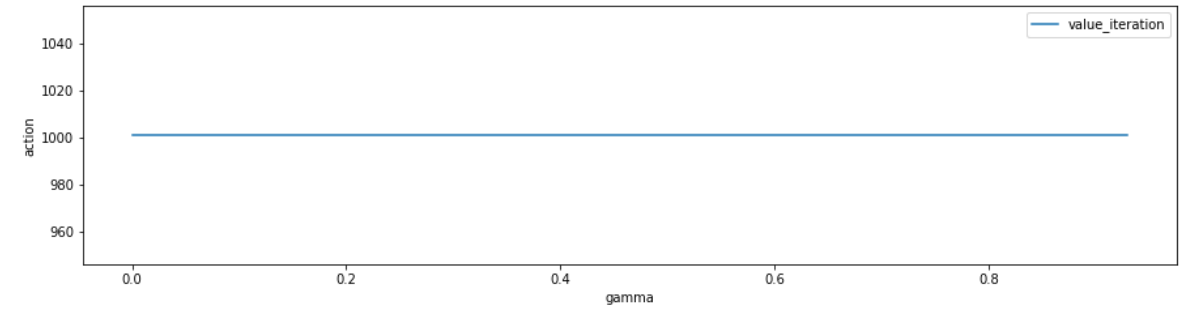


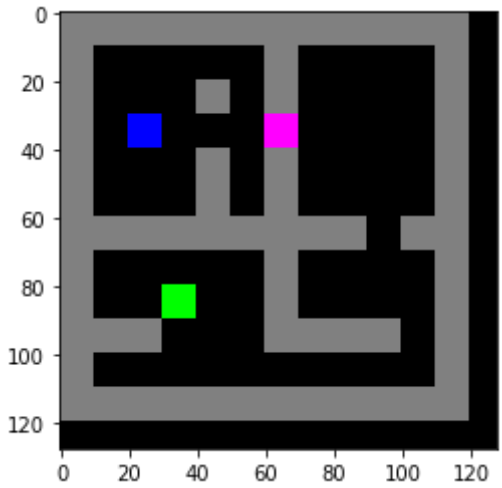


Plan 4

```
plotPlan(df_plan,4)
createEnv(4)
```

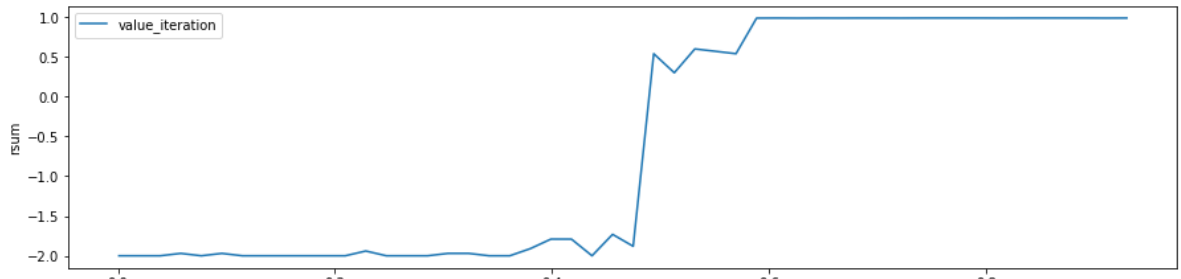


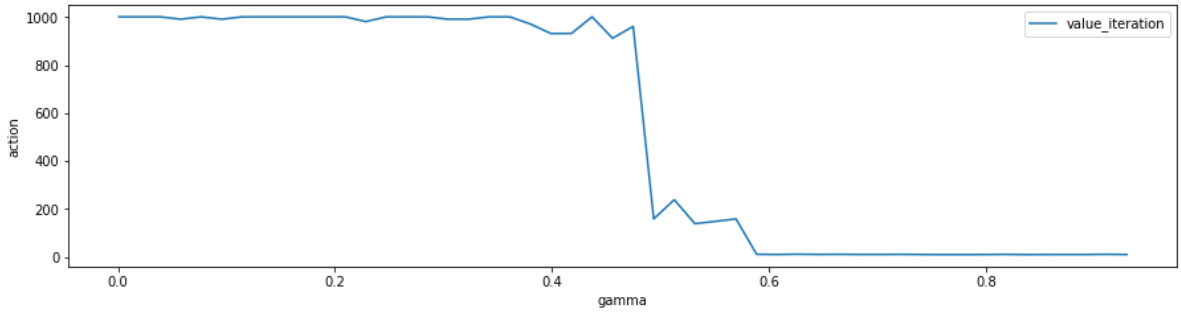


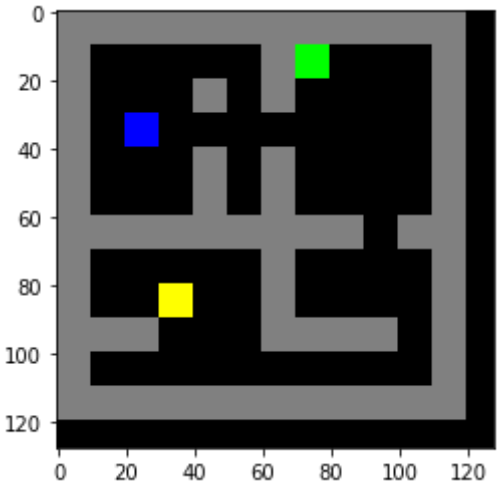


Plan 5

```
plotPlan(df_plan,5)
createEnv(5)
```

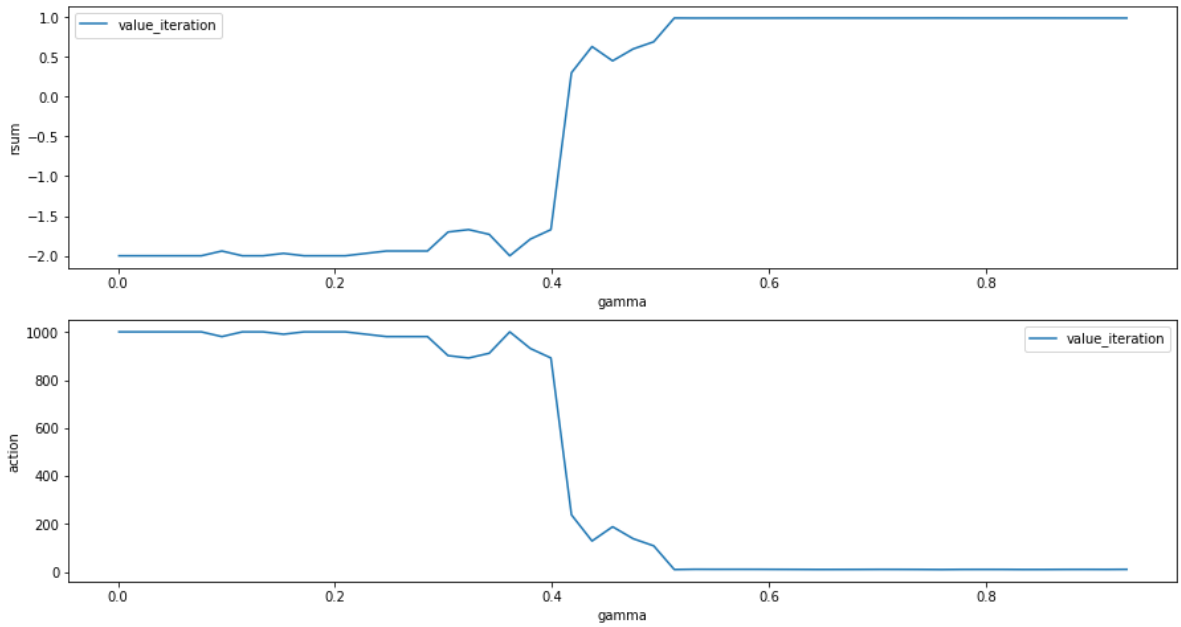


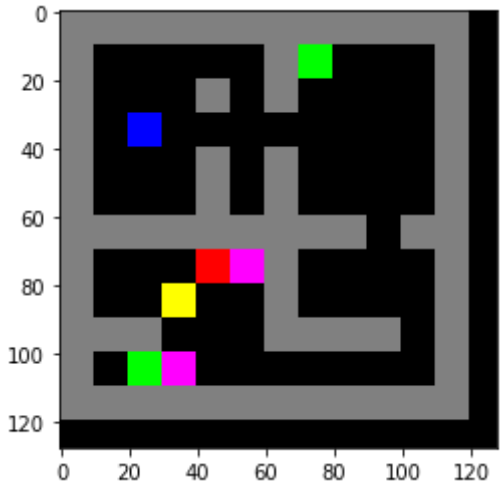




Plan 6

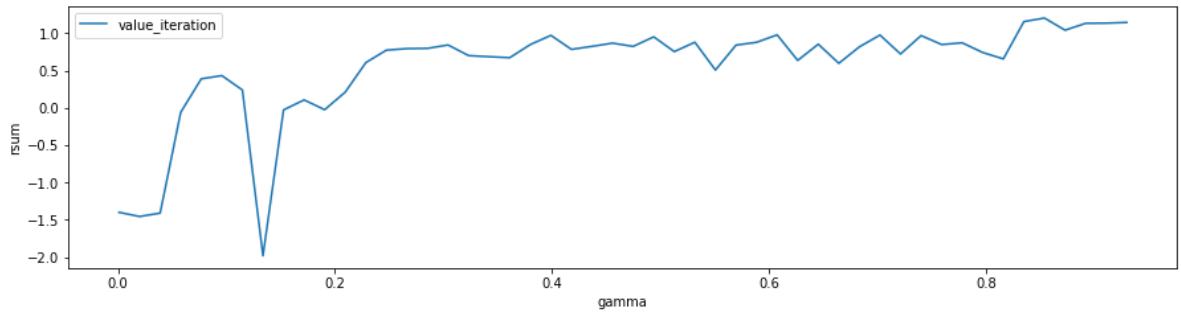
```
plotPlan(df_plan,6)
createEnv(6)
```

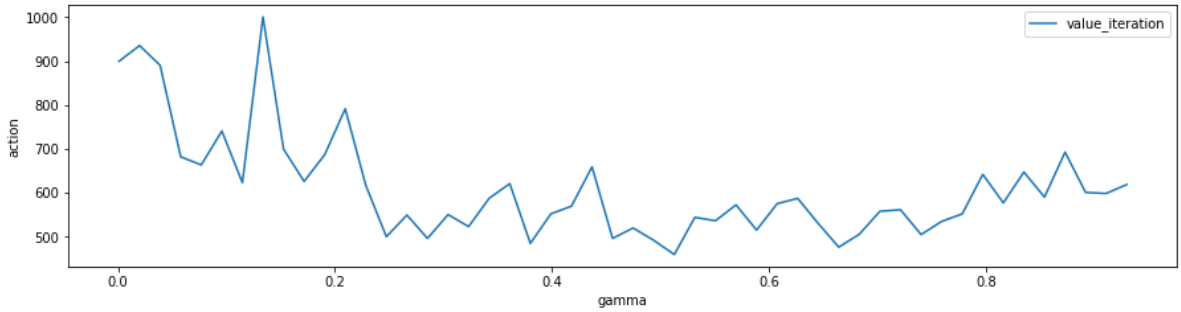


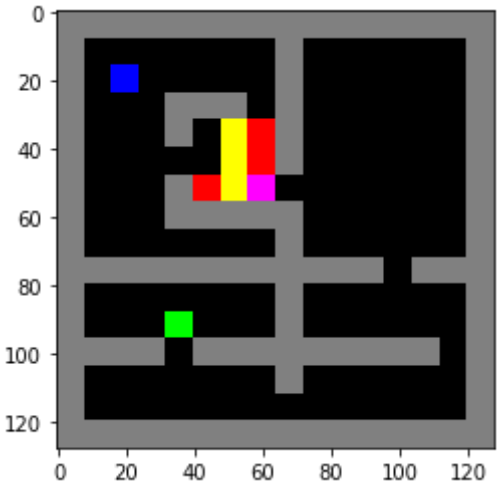


Plan 7

```
plotPlan(df_plan,7)
createEnv(7)
```

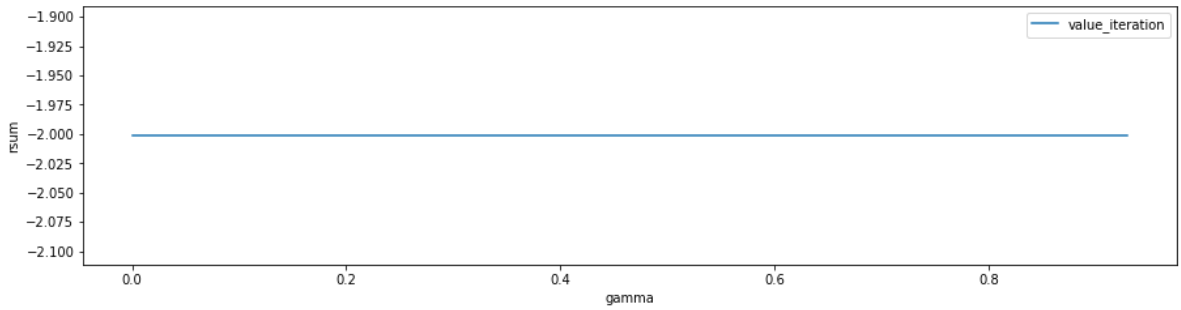


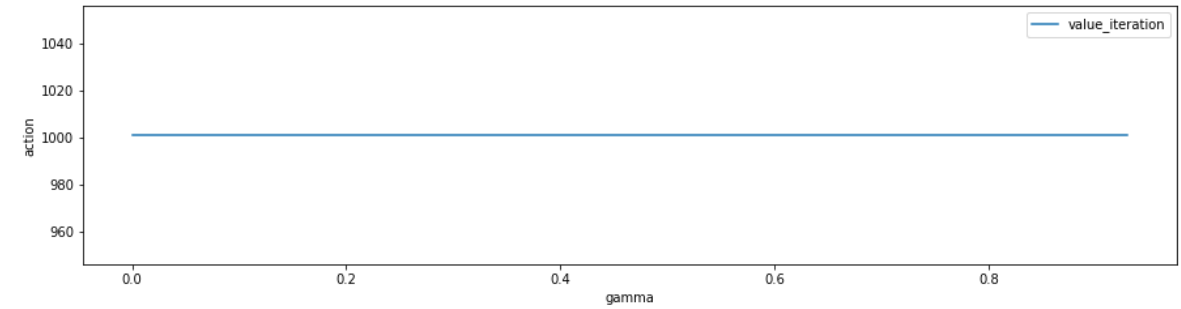


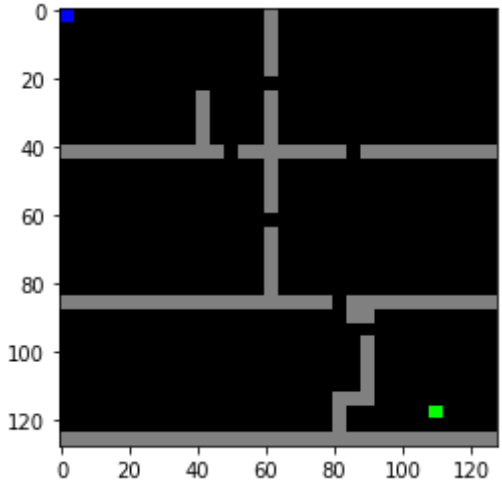


Plan 8

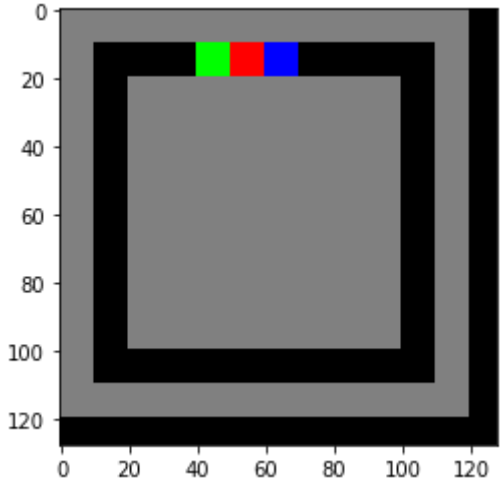
```
plotPlan(df_plan,8)
createEnv(8)
```








```
createEnv(10)
```



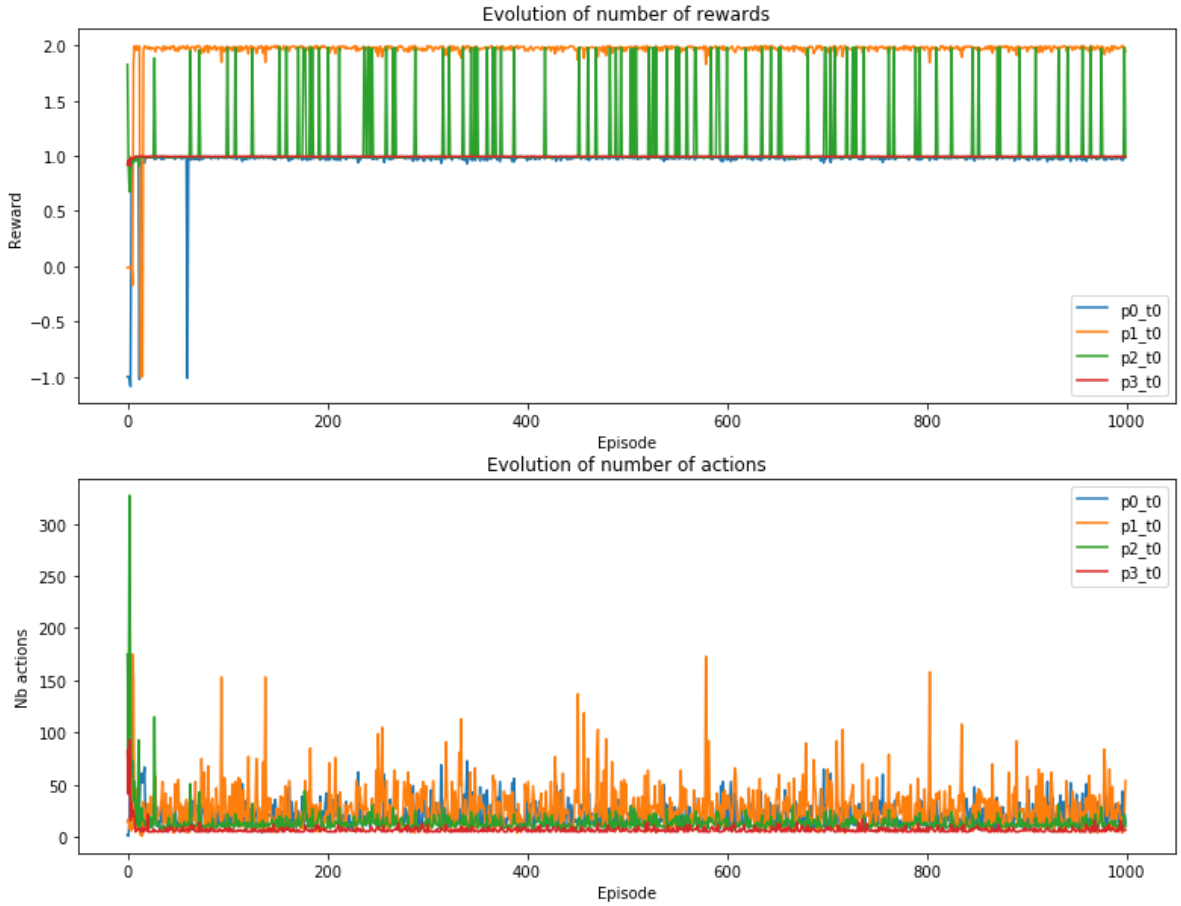
TME 3 — Q-Learning

Victor Duthoit, Pierre Wan-Fat

Dans ce TME, on utilise l'environnement GridWorld.

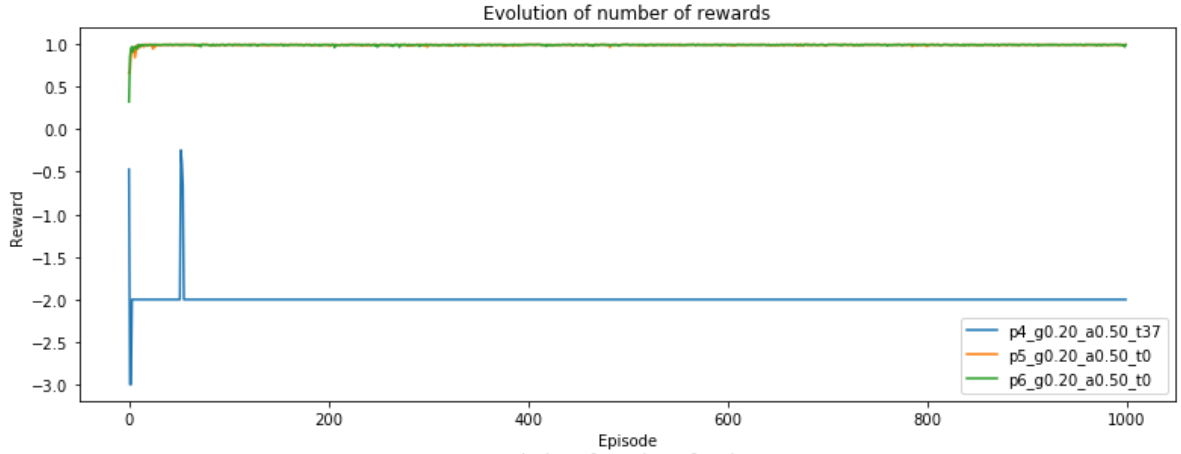
Q-Learning

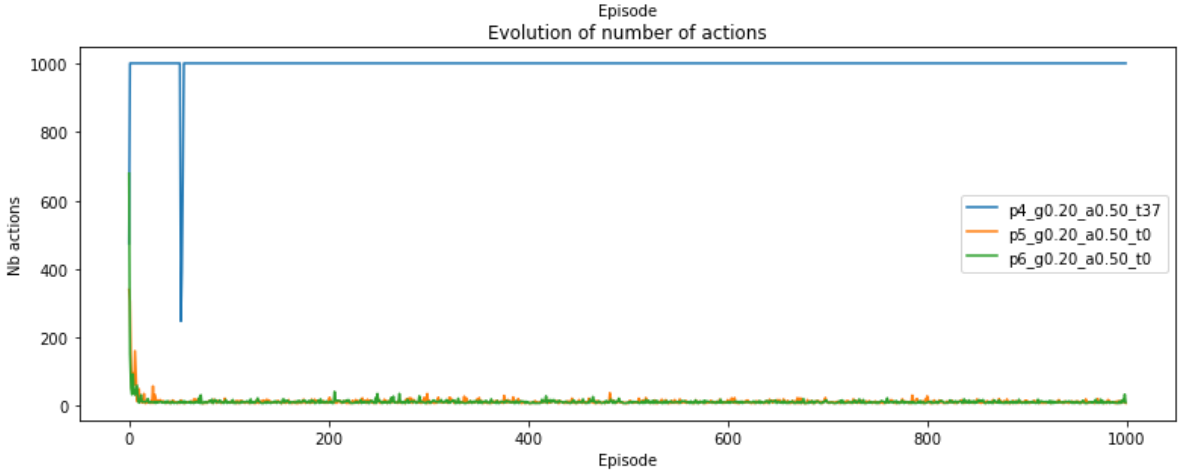
Plans 0 à 3



0. Comme prévu, l'agent apprend très vite sur ce plan.
1. L'agent apprend tout aussi.
2. L'agent se trouve bloqué dans un minimum local pour le plan 2, même s'il est passé au moins une fois dans une trajectoire qui lui a donné la récompense additionnelle.
3. L'agent apprend correctement sur un terrain avec deux états terminaux.

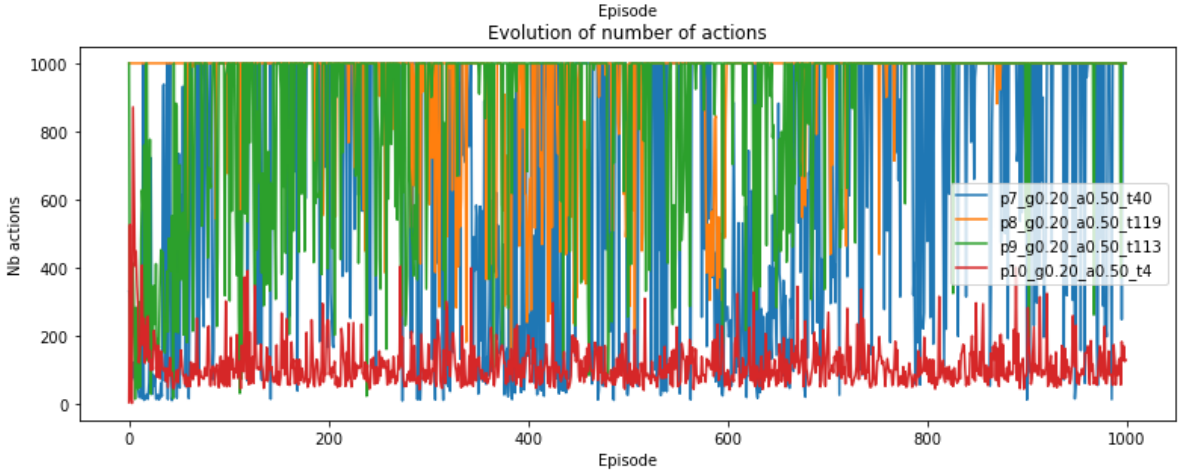
Plans 4 à 6





4. Accéder à l'état terminal nécessite de prendre un malus. L'agent ne semble pas réussir à apprendre cela. Comme pour le plan 2, on note que ce n'est pas parce que la trajectoire optimale a été suivie une fois qu'elle sera suivie par la suite : un apprentissage de plus de 1000 épisodes semble nécessaire.
5. L'agent est bloqué dans un minimum local. Il ne considère pas la récompense qui se trouve loin d'être aléatoirement atteignable.
6. La conclusion tirée pour le plan 5 est aussi valable pour le plan 6.

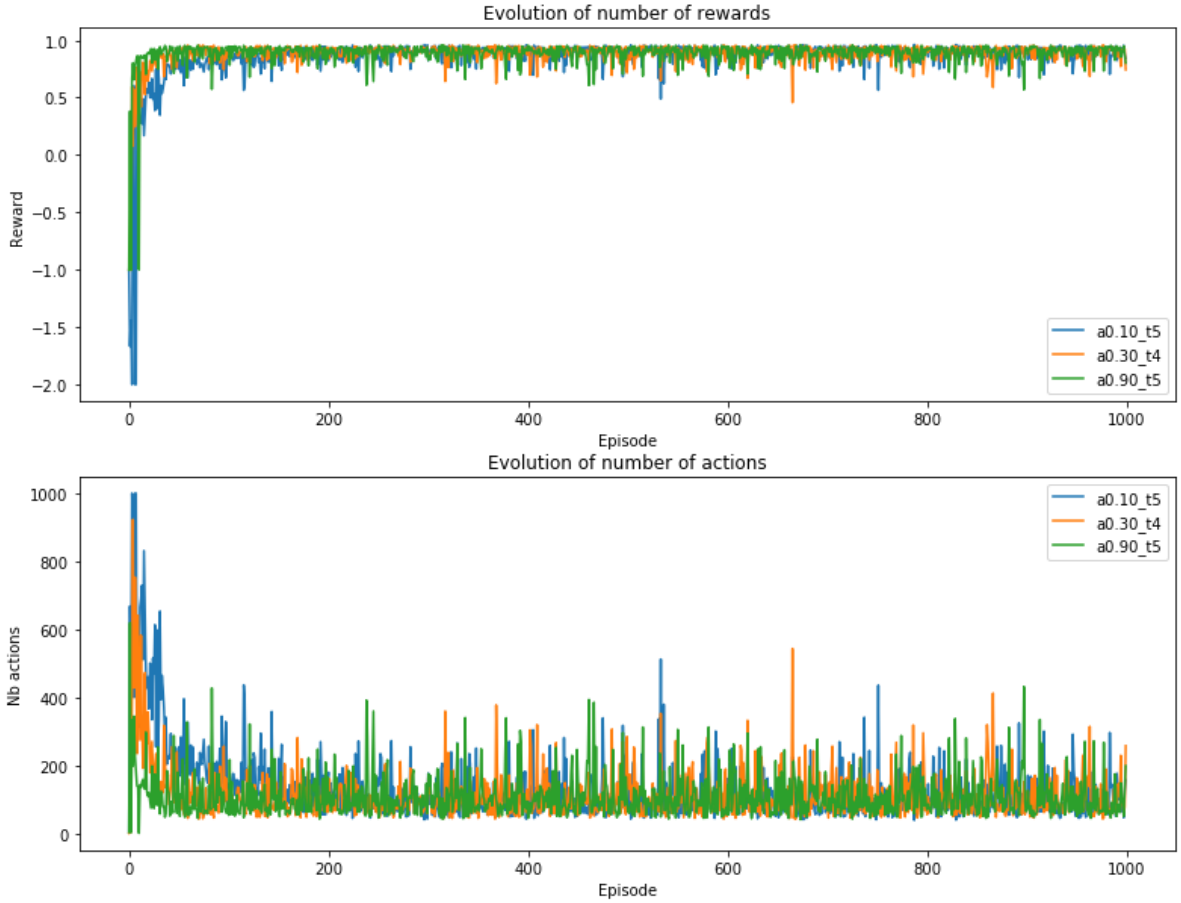
Plans 7 à 10



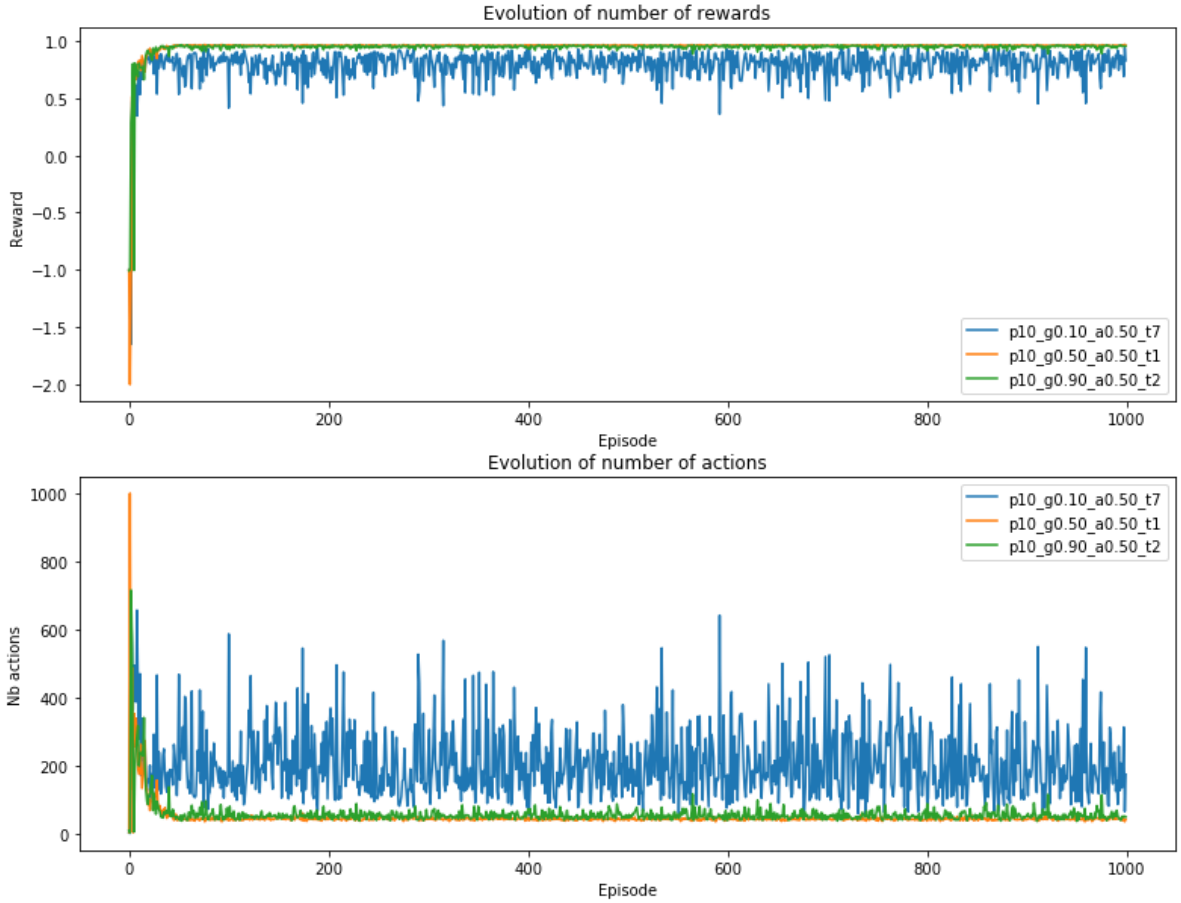
7. Le plan 7 est une version plus difficile du plan 4. L'agent arrive seulement à récupérer les récompenses proches, sans pouvoir atteindre l'état final.
8. Le plan 8 est relativement grand, l'agent prend beaucoup de temps à explorer souvent sans atteindre l'état final. Néanmoins, il atteint plusieurs fois l'état final.
9. Dans ce plan, l'agent trouve une solution sous-optimale : il prend les récompenses mais reste coincé dans la zone de départ. On peut émettre l'hypothèse que les états finaux à récompenses négatives "découragent" l'agent de s'approcher de la porte.
10. Le plan 10 est relativement facile et son apprentissage est relativement stable.

Rôle de γ et α

On effectue les tests sur le plan 1.



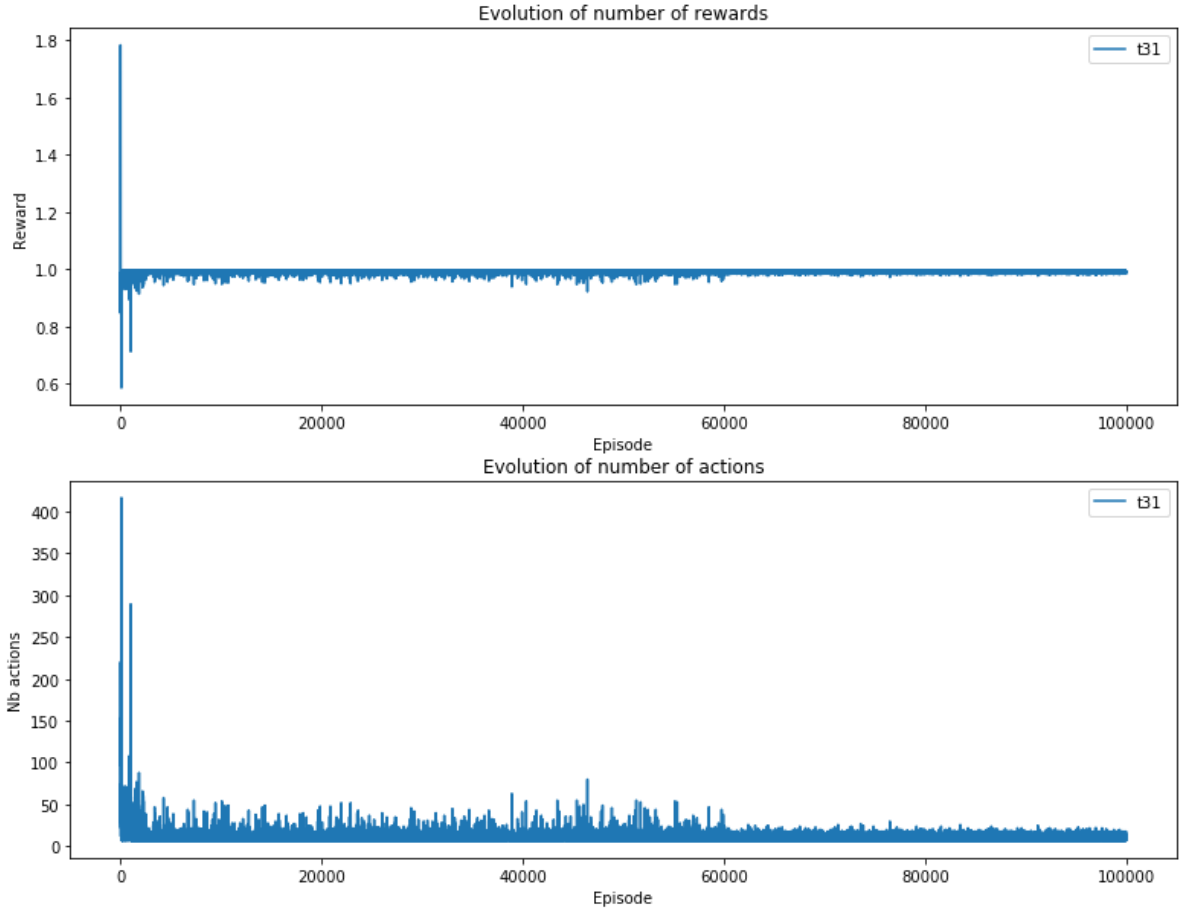
L'apprentissage est plus rapide pour des α élevés ; plus précisément, un α trop faible rend l'apprentissage plus lent et moins stable.



Les conclusions similaires pour l'effet de γ : une valeur trop faible donne un apprentissage instable (l'agent "doute").

Durée de l’apprentissage

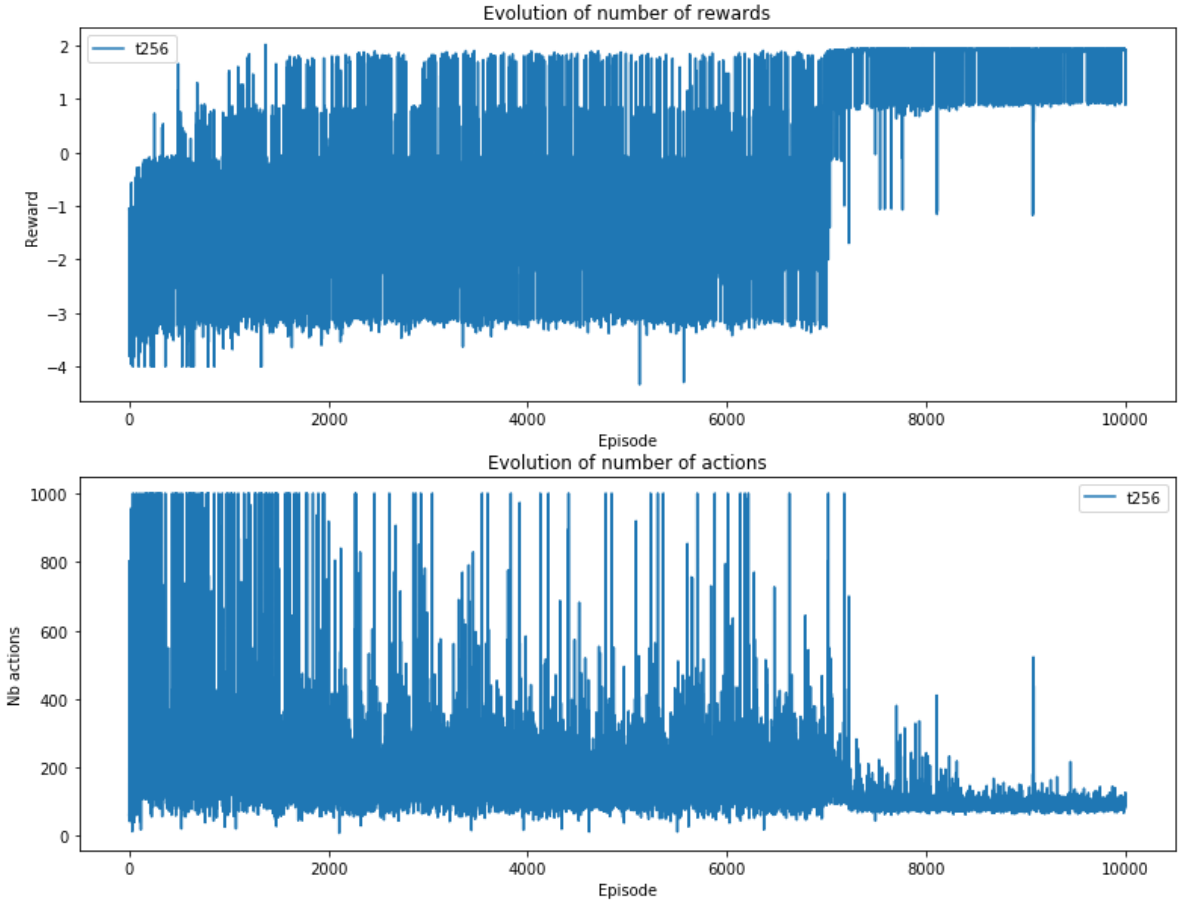
On tente d’augmenter le temps d’apprentissage sur les environnements difficiles.



Le nombre d’épisodes permet avant tout de limiter le nombre d’actions, la trajectoire est plus directe mais pas optimale.

SARSA

On couple l’algorithme SARSA avec une stratégie *epsilon-greedy* qui devient 0 (exploitation pure) après 500 épisodes. Sur le plan 2 :



Il semble qu'une exploration de type "full random" puis "full determinist" ne donne pas de résultats très bons. L'agent explore bien le plan avant l'épisode 7000. Il semblerait qu'il ait appris puisque qu'il réempruntera de nombreuses fois la trajectoire optimale. Néanmoins, il semblerait qu'il change sa stratégie autour de l'épisode 8500 pour prendre uniquement la trajectoire la plus directe. On peut émettre l'hypothèse que la récompense jaune n'est pas forcément évidente à atteindre même en connaissant la trajectoire car l'agent passe par deux fois à côté de l'état final. Ainsi, l'effet non déterministe des mouvements (MDP) va faire arrêter l'agent à plusieurs reprises sur l'état final. Un potentiel phénomène d'apprentissage va augmenter la qualité de la trajectoire menant directement à l'état final. En effet, quand l'agent est placé sur certaines cases, il faut :

- dans le cas où la récompense jaune a déjà été atteinte, se tourner vers la sortie
- dans le cas où la récompense jaune n'a pas encore été ramassée, se tourner vers la case jaune. Une temporalité/directionnalité est ainsi à mettre en place pour permettre de surpasser cet obstacle.

TME 4 — DQN

Victor Duthoit, Pierre Wan-Fat

Nous avons implémenté trois algorithmes, DQN, Prioritized DQN et Dueling DQN. Pour tous ces algorithmes, les techniques d'Experience Replay et de Target Network ont été utilisées. Comme conseillé, afin d'entraîner le réseau Q, nous avons utilisé la fonction de coût Huber.

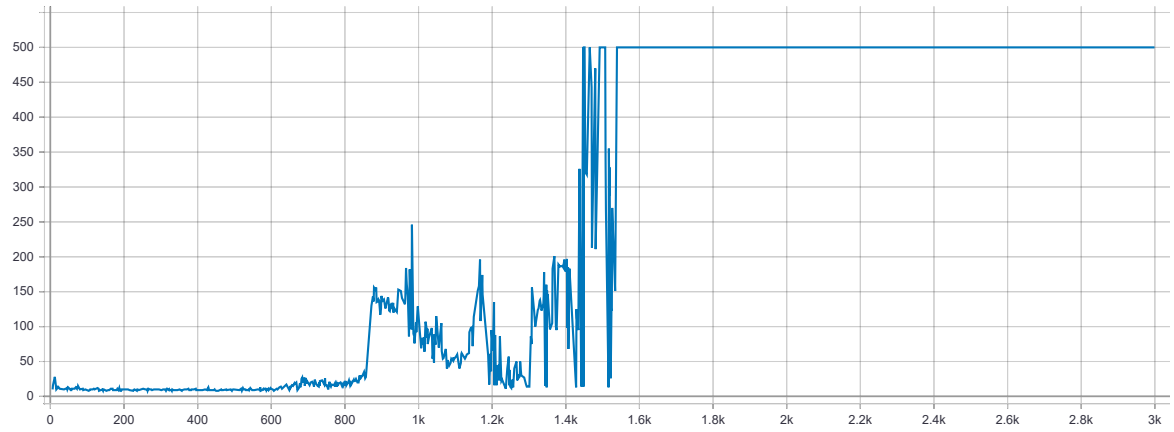
Cartpole

DQN

```
    "memory_size": 3000,  # Size of the Experience Replay buffer.
    "batch_size": 32,
    "epsilon_0": 0.005,  # epsilon = 1 / (epsilon_0 * t).
    "gamma": 0.99,
    "lr": 0.05,
    "q_layers": [24, 24],
    "sync_frequency": 500,  # For Target Network: swap the network every
    `sync_frequency` steps.
```

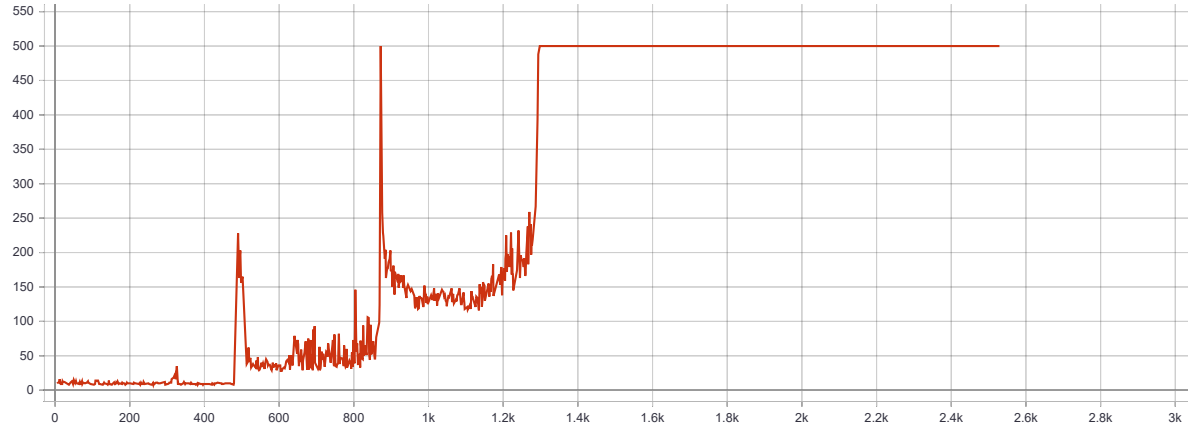
Prioritized DQN

```
    "memory_size": 3000,
    "batch_size": 32,
    "epsilon_0": 0.005,
    "gamma": 0.99,
    "lr": 0.05,
    "q_layers": [24, 24],
    "memory_alpha": 0.5,
    "memory_beta": 0.5,
    "sync_frequency": 500,
```



Dueling DQN

```
"memory_size": 3000,
"batch_size": 32,
"epsilon_0": 0.005,
"gamma": 0.99,
"lr": 0.05,
"advantage_layers": [24],
"value_layers": [24],
"memory_alpha": 0.5,
"sync_frequency": 500,
```



Globalement, les trois algorithmes parviennent à apprendre l'environnement et obtenir le score maximal au bout de quelques centaines d'épisodes. L'algorithme DQN est le plus rapide, bien que l'entraînement soit un peu chaotique au début. Dueling DQN a une courbe un peu plus lisse, mais il apprend un peu moins vite. Enfin, Prioritized DQN a les moins bonnes performances.

LunarLander

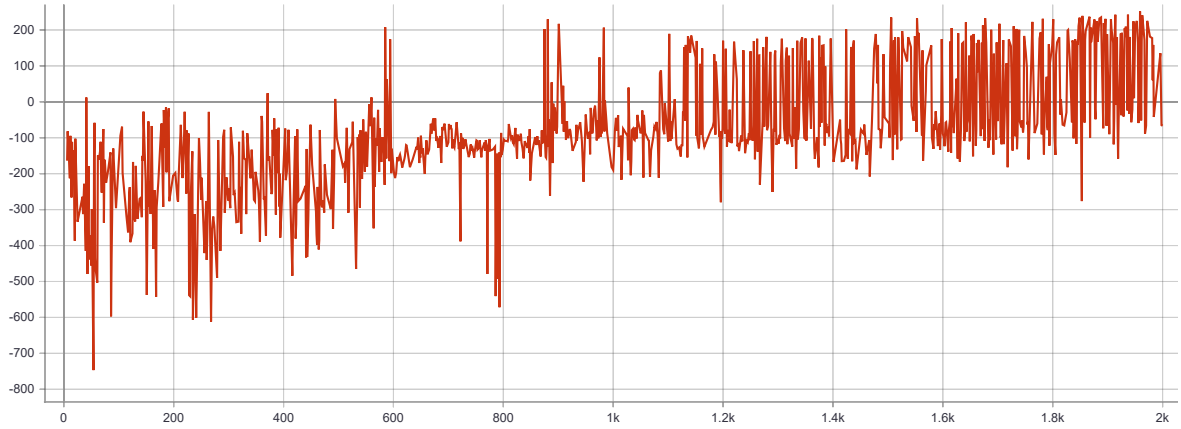
DQN

Afin de trouver de bons hyperparamètres, on procède par recherche par grille.

```
memory_size in (3000, 10000)
epsilon_0 in (0.0001, 0.001, 0.01, 0.1, 1)
gamma in (0.98, 0.99, 0.999)
lr in (1e-7, 1e-6, 1e-5, 1e-3)
sync_frequency in (1000, 5000)
```

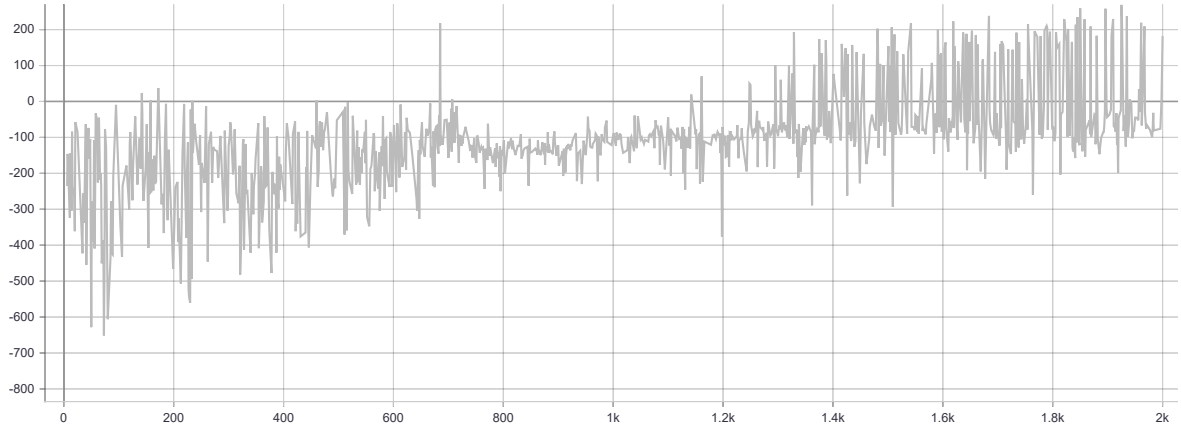
Presque aucun modèle n'a de performances satisfaisantes. On obtient cependant des performances acceptables pour le jeu d'hyper-paramètres suivant :

```
"memory_size": 3000
"batch_size": 64
"epsilon_0": 0.0001
"gamma": 0.99
"lr": 0.001
"sync_frequency": 1000
```



Ou bien avec :

```
"memory_size": 3000
"batch_size": 64
"epsilon_0": 0.0001
"gamma": 0.99
"lr": 0.001
"sync_frequency": 5000
```



On constate tout de même une très forte variance entre les essais, même à la fin de l'entraînement, lorsqu'il n'y a plus d'exploration.

TME 5 — Policy Gradients

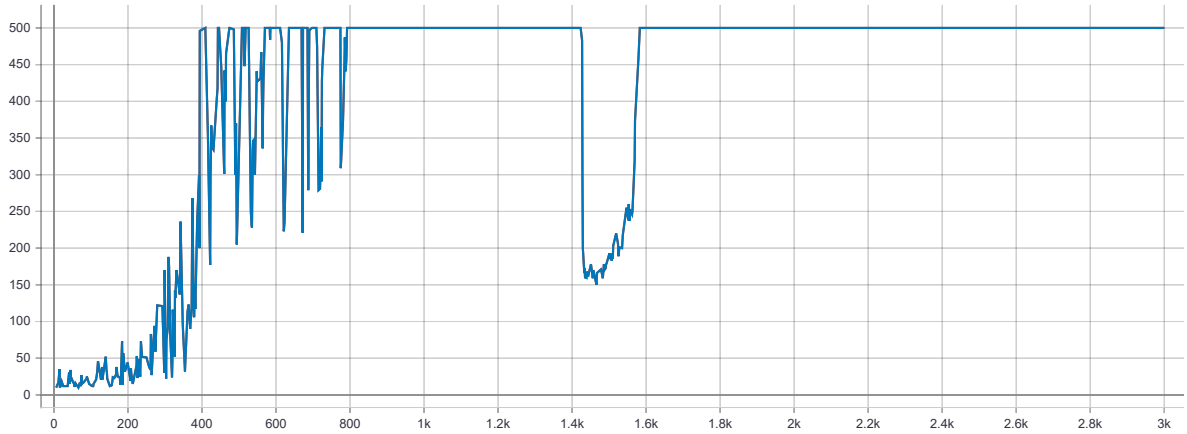
Victor Duthoit, Pierre Wan-Fat

On a implémenté l'algorithme Actor-Critic en version TD(0).

CartPole

Les réseaux de politique et de value partagent la même première couche, à laquelle ils ajoutent chacun une couche cachée (256 neurones dans chaque couche). On utilise par ailleurs un optimiseur Adam.

```
"learning_rate": 0.0002,
"gamma": 0.98,
```



On voit que le réseau Actor-Critic a de très bonnes performances sur CartPole.

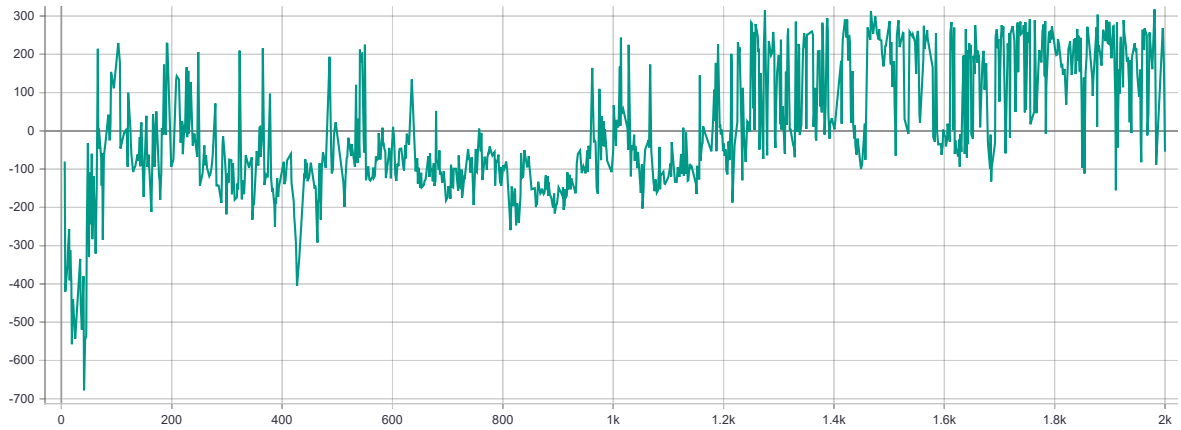
LunarLander

Afin de trouver de bons hyperparamètres, on procède par recherche par grille.

```
learning_rate in (0.0001, 0.001, 0.01, 0.1, 1)
gamma in (0.98, 0.99, 0.999)
```

Presque aucun modèle n'a de performances satisfaisantes. On obtient cependant des performances acceptables pour le jeu d'hyper-paramètres suivant :

```
"learning_rate": 0.001
"gamma": 0.98
```



On constate tout de même une très forte variance entre les essais, même à la fin de l'entraînement, lorsqu'il n'y a plus d'exploration.

TME 6 — Advanced Policy Gradients

Victor Duthoit, Pierre Wan-Fat

On a implémenté les deux versions de PPO (Adaptive KL et Clipped Objective).

Les réseaux de politique et de value partagent la même première couche, à laquelle ils ajoutent chacun une couche cachée (256 neurones dans chaque couche). On utilise par ailleurs un optimiseur Adam.

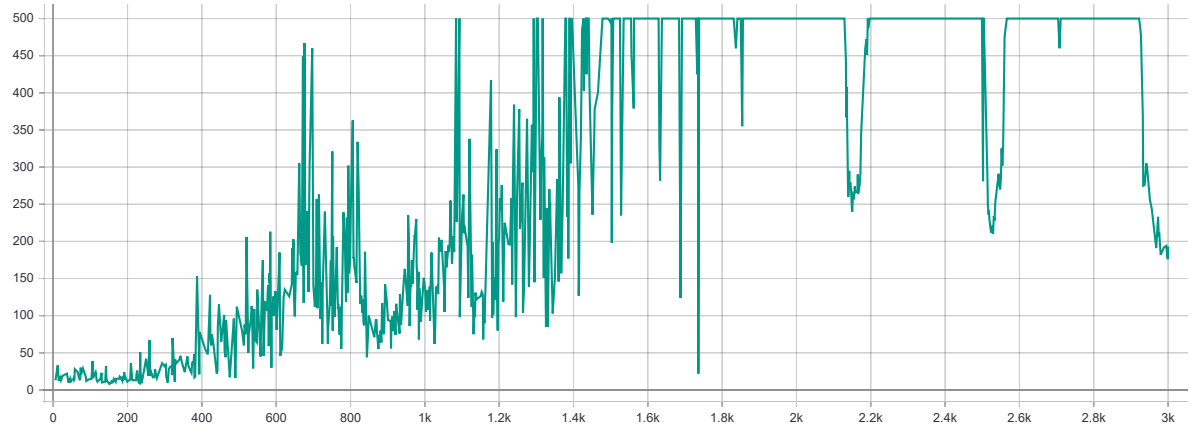
CartPole

PPO Adaptive KL

Afin de trouver de bons hyperparamètres, on procède par recherche par grille.

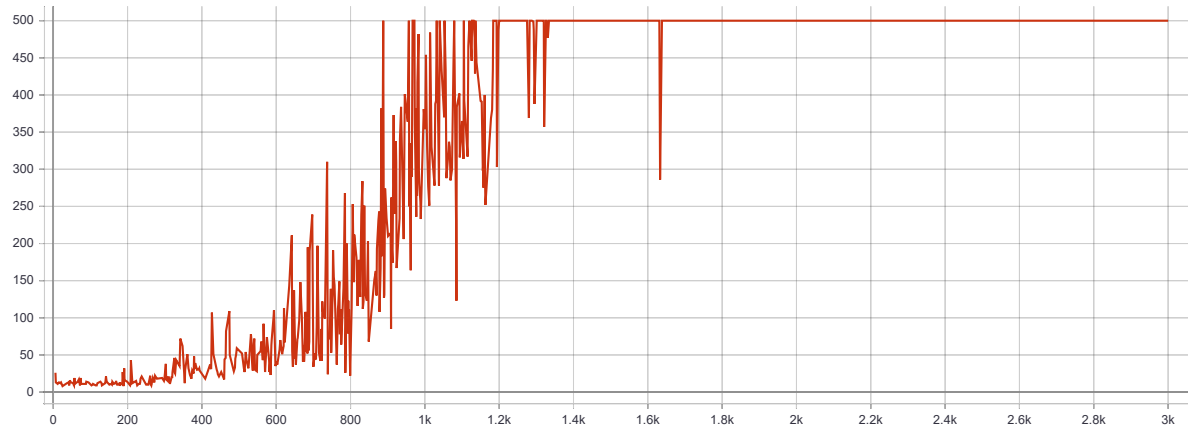
```
learning_rate in (0.0001, 0.001, 0.01)
gamma in (0.98, 0.99, 0.999)
delta in (1e-3, 1e-2, 5e-2)
k in (2, 3, 4)
```

Sur un environnement aussi simple, l'algorithme arrive souvent à atteindre des épisodes à 500 itérations, mais très souvent, il décroche et oublie ce qu'il a appris, comme par exemple :



On trouve néanmoins des entraînements plus stables, comme celui-ci :

```
"learning_rate": 0.0001,
"gamma": 0.98,
"delta": 0.001,
"k": 3,
```

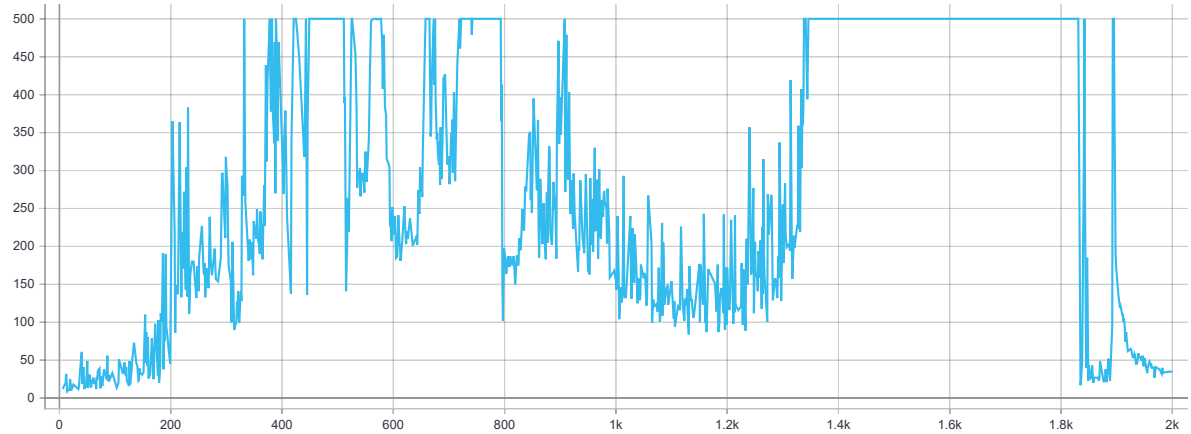


PPO Clipped

Afin de trouver de bons hyperparamètres, on procède par recherche par grille.

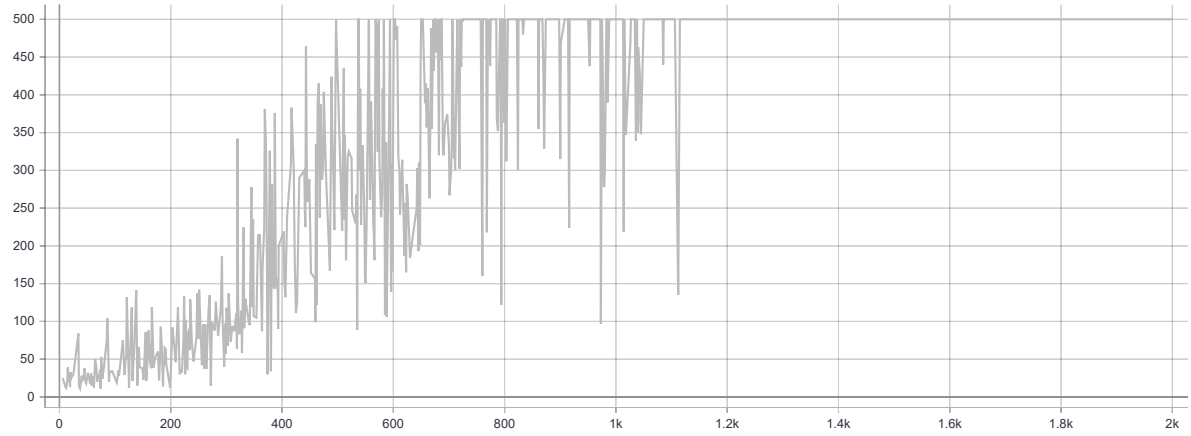
```
learning_rate in (0.0001, 0.001, 0.01)
gamma in (0.98, 0.99, 0.999)
epsilon in (1e-3, 1e-2)
k in (2, 3, 4)
```

Globalement, cette version de PPO arrive encore plus que la précédente à atteindre les 500 itérations. Cependant, il est plus rare que l'agent parvienne à maintenir cette performance, et de nombreux agents connaissent du *catastrophic forgetting* :



On trouve néanmoins des entraînements plus stables, comme celui-ci :

```
"learning_rate": 0.0001,
"gamma": 0.98,
"epsilon": 0.01,
"k": 4,
```

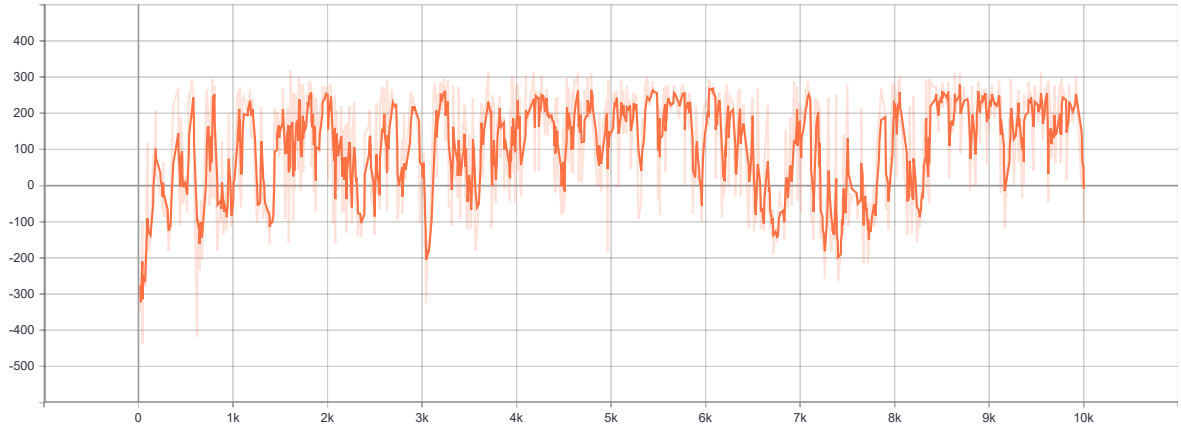


LunarLander

PPO Adaptive KL

Après recherche par grille, on trouve les hyperparamètres suivants :

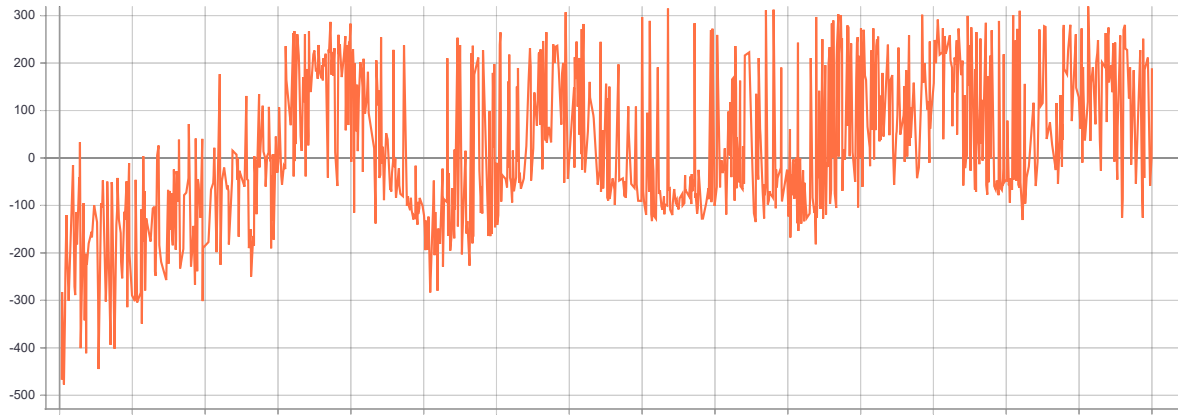
```
"learning_rate": 0.001
"gamma": 0.98
"k": 2
"delta": 0.01
```



PPO Clipped

Après recherche par grille, on trouve les hyperparamètres suivants :

```
"learning_rate": 0.0001
"gamma": 0.98
"k": 4
"epsilon": 0.01
```



On constate que l'algorithme a des récompenses globalement positives, ce qui indique qu'il a réussi la tâche, même s'il y a une fort variance dans les résultats, avec des passages où les récompenses deviennent complètement négatives.

TME 7 — Continuous Actions

Victor Duthoit, Pierre Wan-Fat

On a implémenté l'algorithme DDPG.

Pendulum

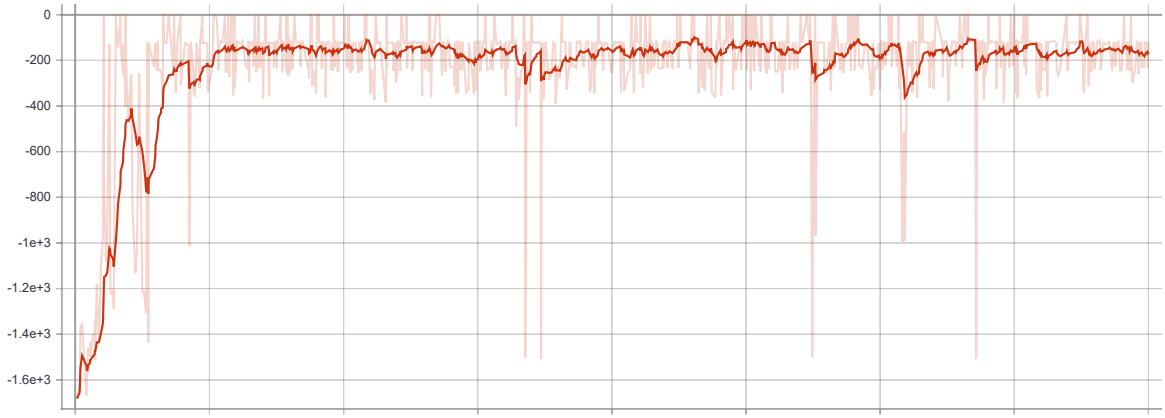
Afin de trouver de bons hyperparamètres, on procède par recherche par grille.

```
number_of_updates in (3, 5)
policy_learning_rate in (1e-4, 1e-3, 1e-2)
q_learning_rate in (1e-4, 1e-3, 1e-2)
noise_sigma in (0.05, 0.1)
gamma in (0.98, 0.99)
rho in (0.99, 0.995)
```

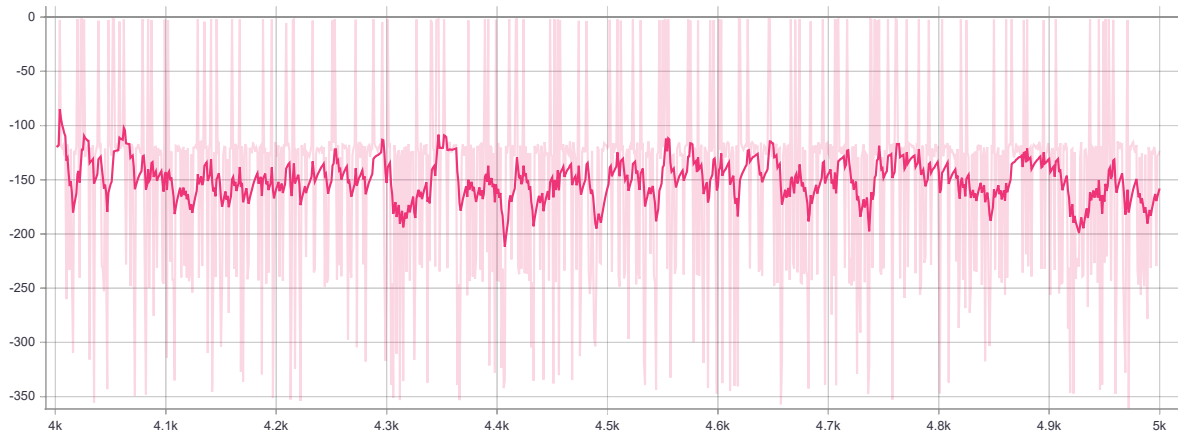
On trouve les hyperparamètres suivants :

```
"number_of_updates": 5,
"policy_learning_rate": 0.001,
"q_learning_rate": 0.01,
"noise_sigma": 0.1,
"memory_max_size": 10000,
"batch_size": 1024,
"gamma": 0.99,
"rho": 0.99,
```

Les 4 000 premiers épisodes sont des épisodes d'entraînement :



Puis, on évalue l'agent sur 1 000 épisodes supplémentaires, où on désactive l'exploration :



On voit que l'agent parvient assez souvent à récupérer une récompense maximale (0 ou proche de 0) ; les récompenses oscillent globalement entre -350 et 0.

LunarLanderContinuous

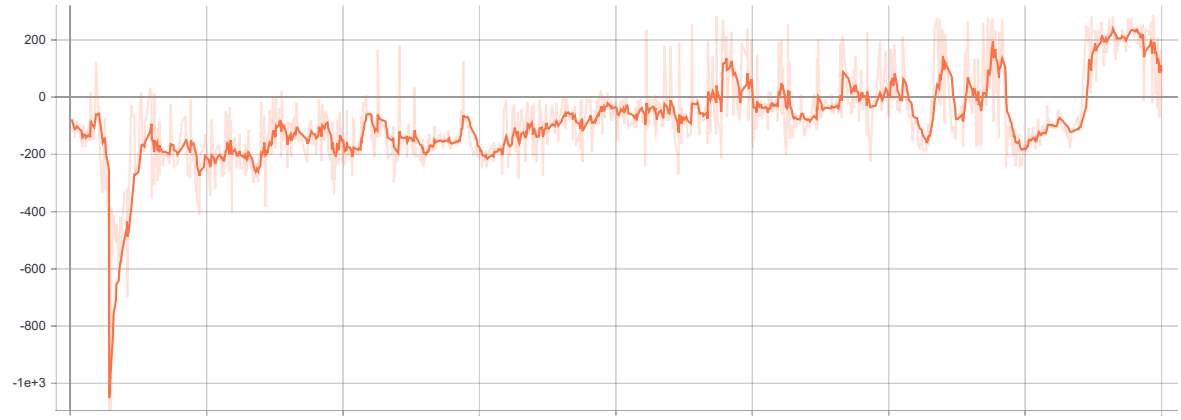
Afin de trouver de bons hyperparamètres, on procède par recherche par grille.

```
number_of_updates in (3, 5)
policy_learning_rate in (1e-4, 1e-3, 1e-2)
q_learning_rate in (1e-4, 1e-3, 1e-2)
noise_sigma in (0.05, 0.1)
gamma in (0.98, 0.99)
rho in (0.99, 0.995)
```

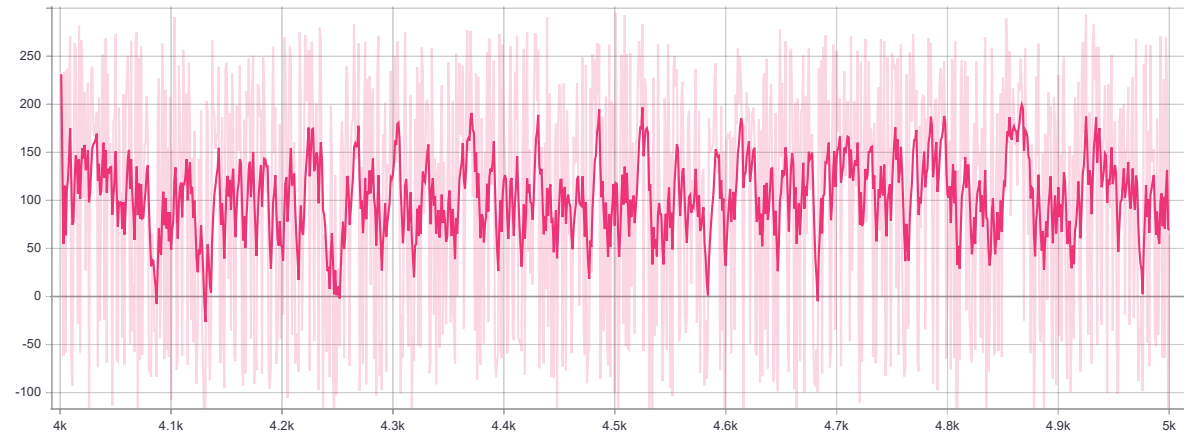
On trouve les hyperparamètres suivants :

```
"number_of_updates": 3,
"policy_learning_rate": 0.0001,
"q_learning_rate": 0.001,
"noise_sigma": 0.1,
"memory_max_size": 10000,
"batch_size": 1024,
"gamma": 0.99,
"rho": 0.995,
```

Les 4 000 premiers épisodes sont des épisodes d'entraînement :



Puis, on évalue l'agent sur 1 000 épisodes supplémentaires, où on désactive l'exploration :



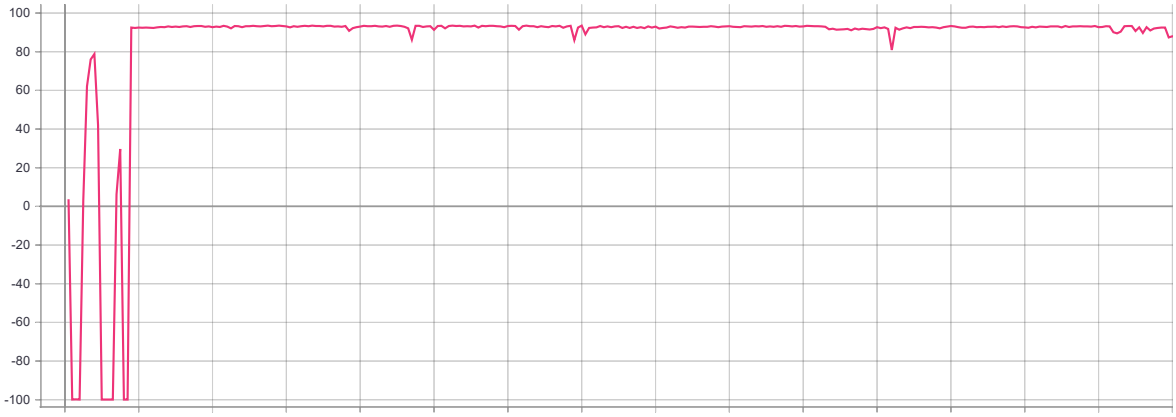
On voit que l'agent parvient assez souvent à récupérer des récompenses positives, même s'il continue, une fois sur cinq, à avoir des récompenses négatives.

MountainCarContinuous

Pour ce problème, une bonne exploration est cruciale. En effet, l'agent perd des points à chaque fois qu'il bouge ; s'il ne comprend pas très vite qu'une récompense l'attend à l'extrême droite du terrain, il va se laisser enfermer dans un minimum local fait de très petites oscillations autour de sa position de départ. Ainsi, dans cet exemple, même si l'agent atteint la récompense ponctuellement, il n'explore plus du tout au bout de quelques itérations :

Nous avons donc changé la stratégie d'exploration par rapport aux problèmes précédents, en prenant un `noise_sigma` de 0,5 (5 fois plus que précédemment) et en introduisant un *decay* sur le bruit.

```
  "number_of_updates": 3,
  "policy_learning_rate": 0.01,
  "q_learning_rate": 0.001,
  "noise_sigma": 0.5,
  "memory_max_size": 10000,
  "batch_size": 256,
  "gamma": 0.99,
  "rho": 0.999,
```



L’agent parvient donc à apprendre le mouvement à réaliser et à s’y maintenir.

TME 8 — Generative Adversarial Networks

Victor Duthoit, Pierre Wan-Fat

Le but de ce TP est de générer des visages de célébrités grâce à des réseaux adverses (GAN). Pour cela, on utilise une architecture DCGAN.

Voici les résultats obtenus après une centaine d'époques :

Comme on le voit, les visages ressemblent effectivement à des visages humains, bien que certains détails ne soient pas parfaits. De plus, les visages sont relativement variés : on ne voit pas deux fois le même visage, et il y a un certaine variation de l'arrière-plan, des couleurs de cheveux, des poses…

VAE : Variational Auto-Encoder

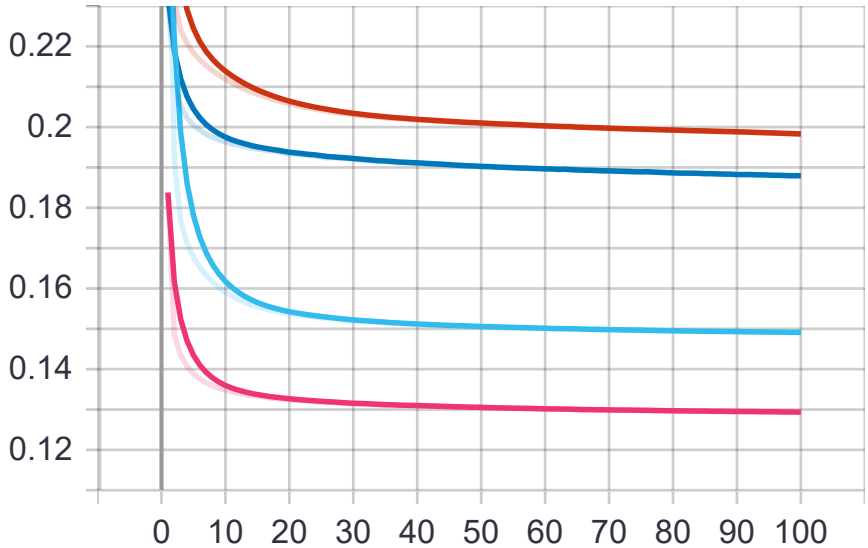
Victor Duthoit, Pierre Wan-Fat

On implémente dans ce TP un modèle génératif par auto-encodage variationnel. On met en place un modèle linéaire et un convolutif pour étudier les différences produites.

Entraînement

Les modèles produits apprennent correctement. On peut voir dans la figure ci-dessous la descente de gradient pour :

- un modèle linéaire de dimension latente 2 (rouge)
- un modèle linéaire de dimension latente 10 (cyan)
- un modèle convolutif de dimension latente 2 (bleu)
- un modèle convolutif de dimension latente 10 (rose)

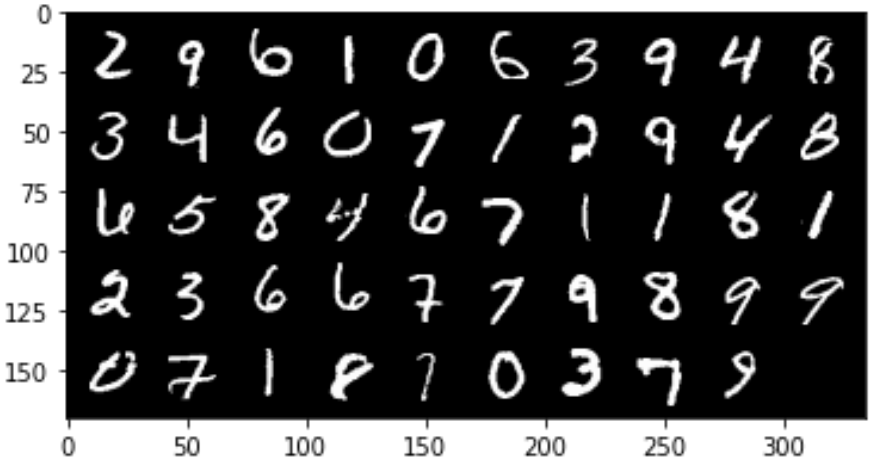


Train loss

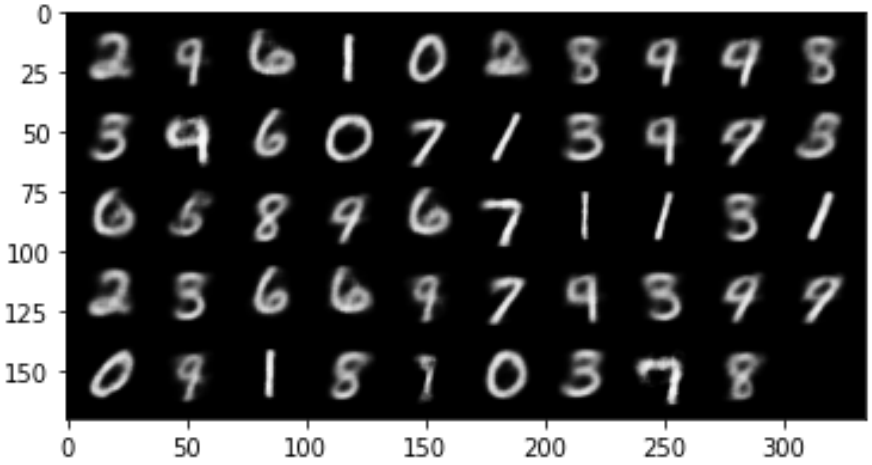
Une dimension plus grande permet un meilleur apprentissage du modèle, laissant plus de flexibilité et de détail dans la représentation latente. Par ailleurs, les réseaux convolutifs permettent d'améliorer la proximité avec la distribution latente normale et la reconstruction de l'image de départ.

On se concentrera ainsi sur les réseaux convolutifs offrant un meilleur apprentissage (une fonction de *loss* plus faible en test).

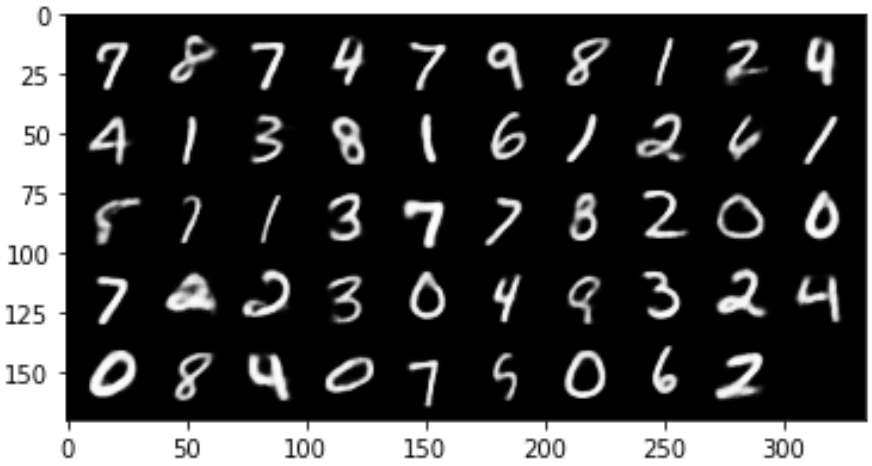
Évaluation



Images originales



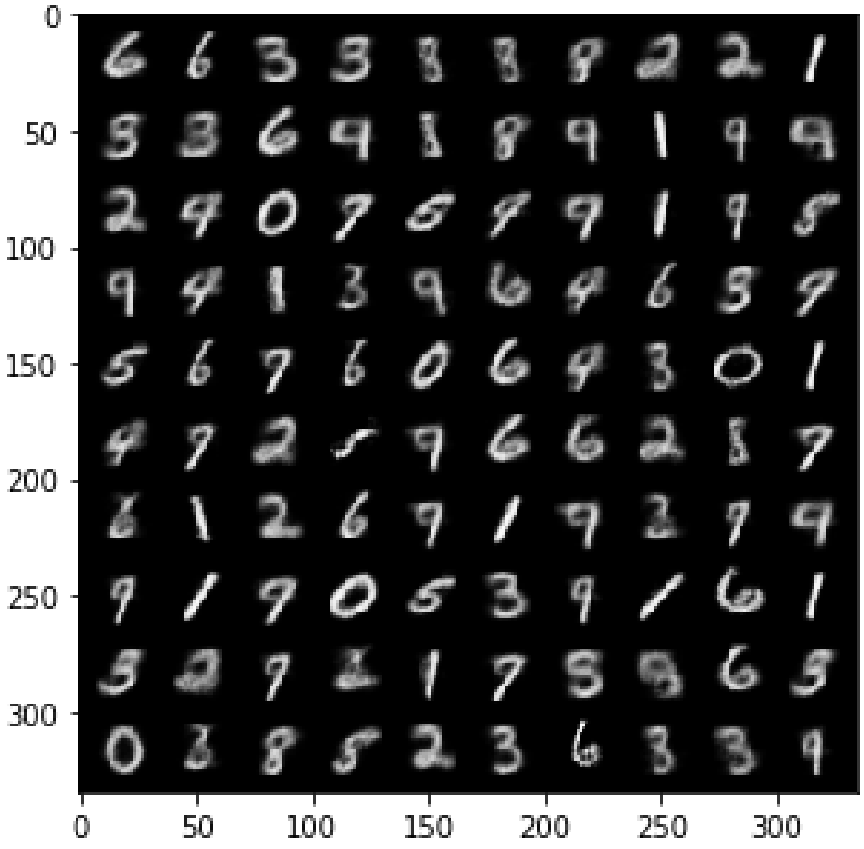
Reconstitution (convolution, dim 2)



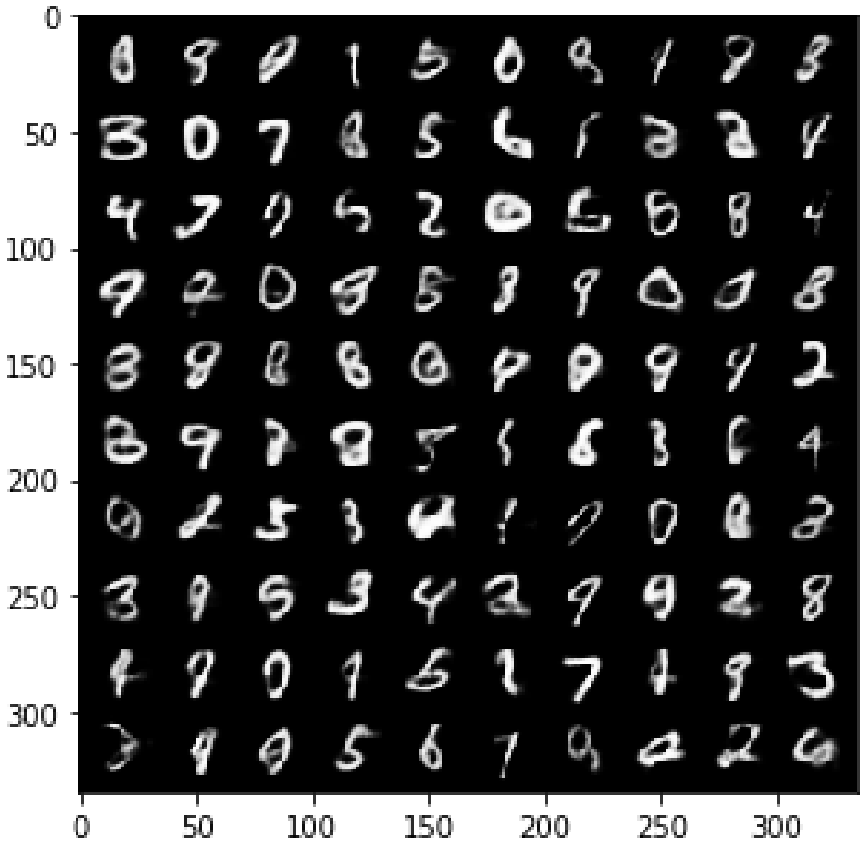
Reconstitution (convolution, dim 10)

On peut apprécier sur la figure ci-dessus que les réseaux reconstituent correctement les images de la base de données. Par ailleurs, le modèle utilisant une dimension de l'espace latent de 10 est plus précis, plus net.

Cela se remarque dans les images générées aléatoirement à partir d'une distribution normale sur l'espace latent :



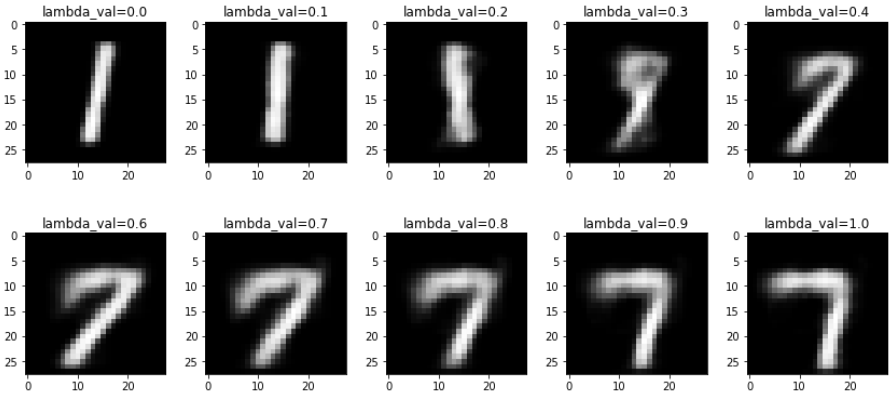
Génération (convolution, dim 2)



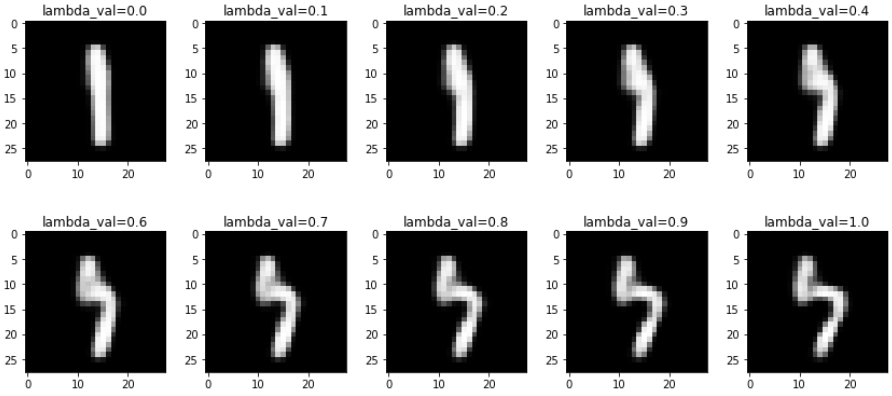
Génération (convolution, dim 10)

Les images sont plus nettes avec un espace plus grand. Néanmoins, elles ne paraissent pas forcément plus vraisemblables. En effet, un plus gros contraste sur le modèle de dimension 10 accentue certains défauts.

Nous avons tenté de mettre en place une interpolation entre deux points de l'espace latent. Ce test permet de vérifier si le *prior* gaussien a bien permis un régularisation de l'espace latent.



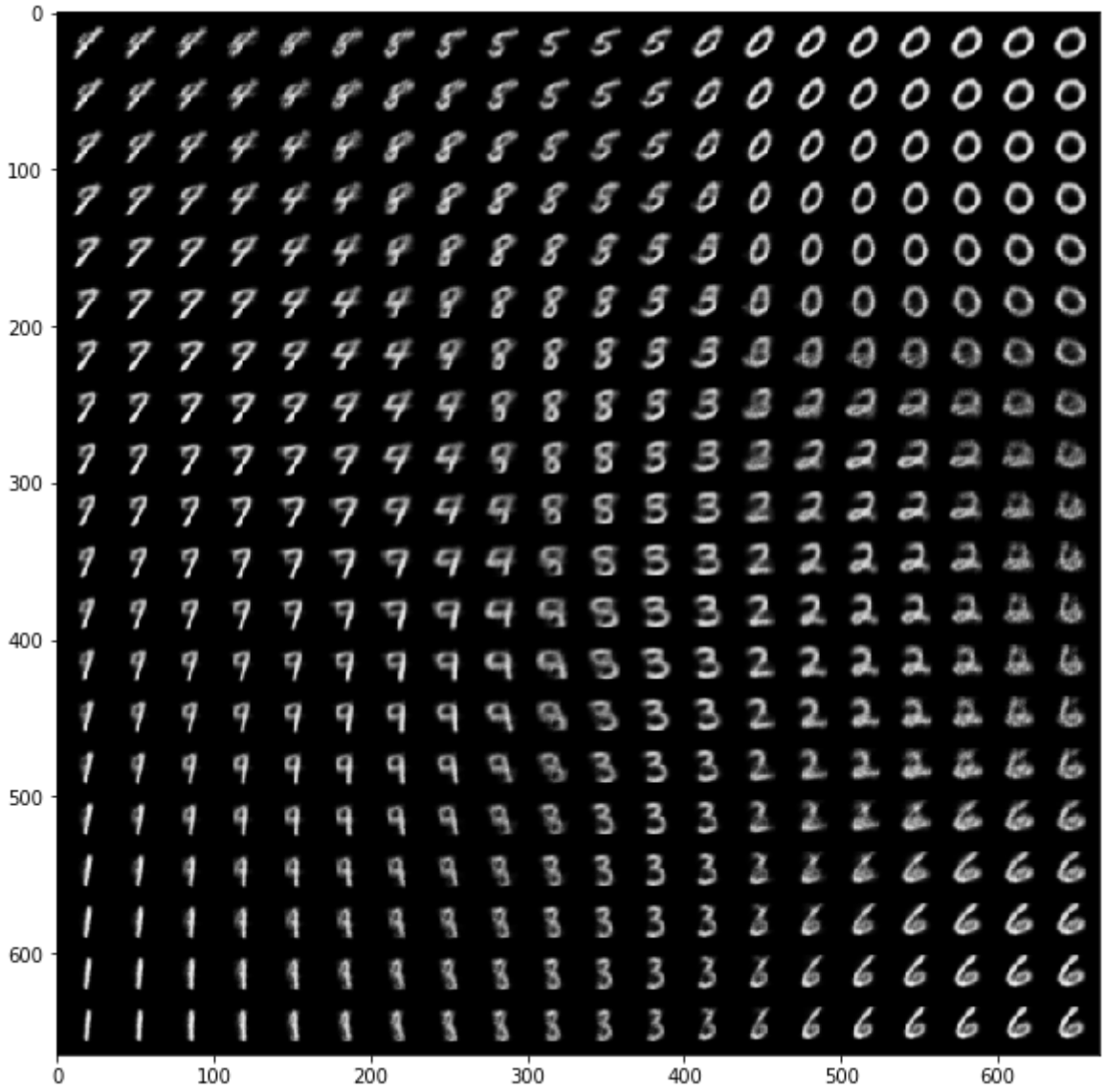
Interpolation (convolution, dim 2)



Interpolation (convolution, dim 10)

Il apparait que l'interpolation du modèle de dimension 2 passe toujours par des données vraisemblables (1, 9, et 7). Cela est moins flagrant pour le modèle de dimension 10 où les chiffres sont plus difficiles à distinguer.

Finalement, on génère des images depuis une grille de données latentes $[-1.5, 1.5]^2$. On retrouve une continuité dans les données produites. Par ailleurs, il est possible de visualiser l'interpolation produite plus haut. Finalement, le 8 semble jouer un rôle central dans cette distribution. Cela semble compréhensible au vu de sa géométrie calligraphique occupant un grand espace.



Conclusion

Pour conclure, on a vu qu'il était possible de faire apprendre une distribution normale à des données par encodage. Si les modèles linéaires apprennent correctement, leurs homologues convolutifs semblent toutefois les surpasser. Le choix de la dimension latente n'est pas évident. Si des modèles à dimension plus élevée permettent une meilleur diminution de l'erreur, on peut apercevoir des aberrations trop contrastées lors de la génération. Un modèle à faible dimension n'offre pas autant de contraste lors de la reconstitution mais cela lui permet de ne pas accentuer ses défauts lors de la génération.

Rapport TP10 : MADDPG

Victor Duthoit, Pierre Wan-Fat

On met en place le modèle MADDPG sur les trois environnements proposés.

Simple spread

Dans cet environnement, les agents ne semblent apprendre que très peu, ou du moins de manière très lente. Comme on peut le voir dans la figure ci-dessous, l'agent commence d'abord à perdre en nombre quantité de récompenses.

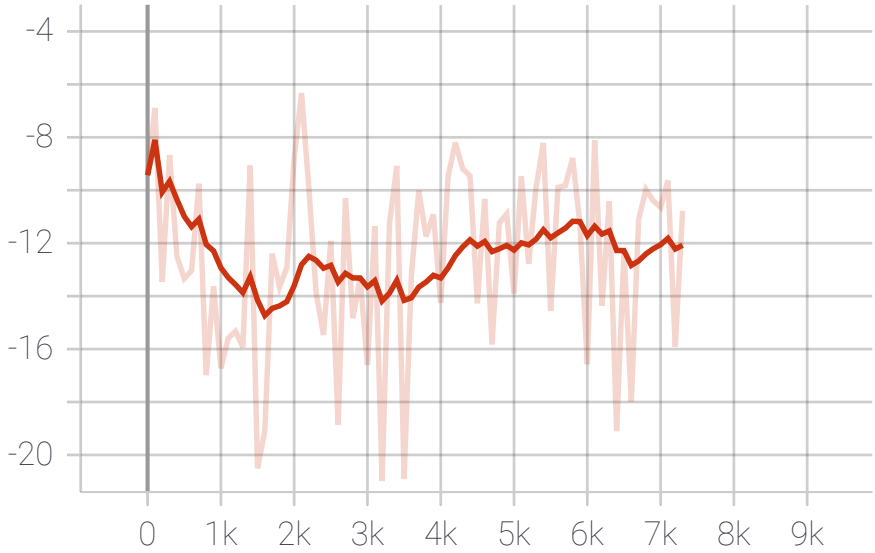


Fig. 1. Récompenses de l'agent 0 pour les 20000 premières trajectoires

Néanmoins, on note bien une descente du coût TD. L'erreur augmente lors du remplissage du *replay buffer* puis commence à descendre. La recherche d'hyperparamètres par optimisation automatique (avec *Optuna*) n'a pas donné de meilleurs résultats.

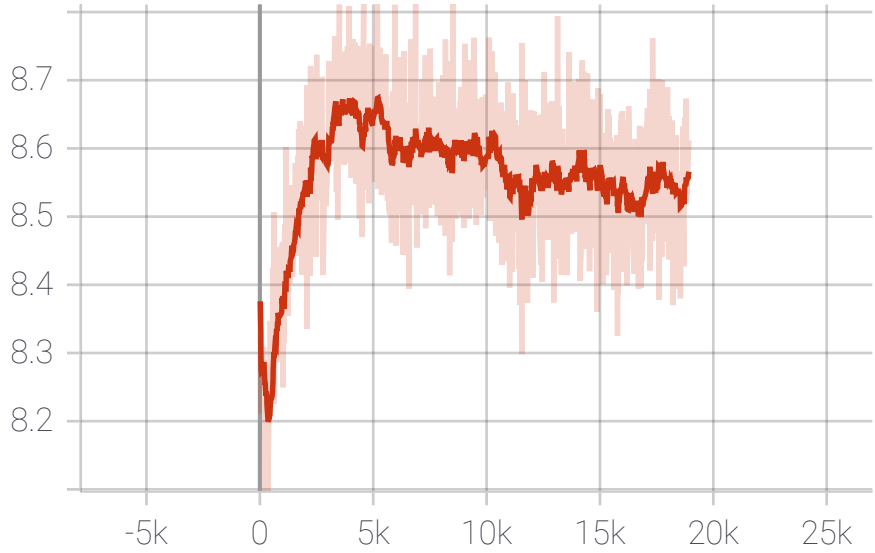


Fig. 2. Fonction de coût sur les 2000 premières descentes de gradient

Simple adversary

On peut voir que l'agent apprend correctement sur cet environnement bien que les agents manquent de stabilité dans leur récompenses à long terme. Il est important de visualiser la récompense totale des agents car elles sont spécifiques par individu dans cet environnement. Par ailleurs, il est intéressant de noter que les récompenses sont plutôt similaires entre l'agent 0 et les deux autres agents 1 et 2 qui sont adversaires. Les récompenses semblent ainsi relativement bien équilibrées pour éviter qu'un des agents prenne le dessus.

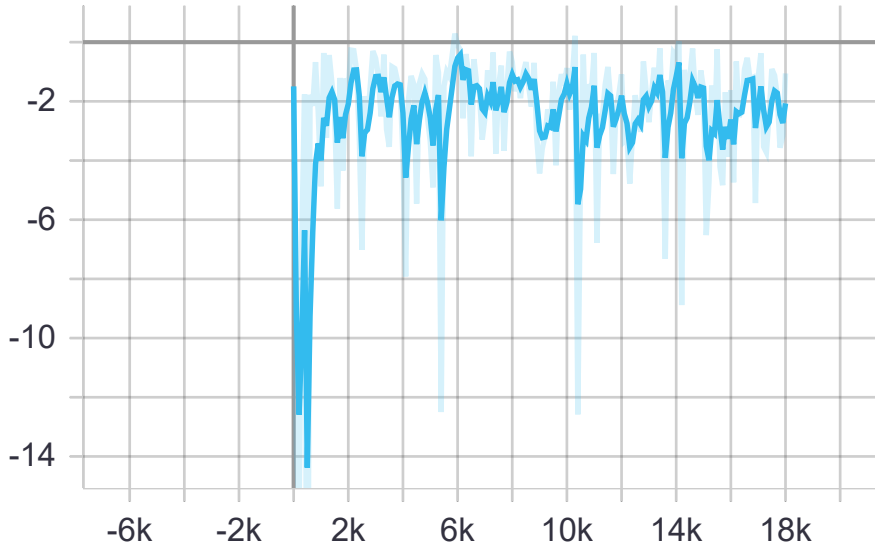


Fig. 3. Somme sur les 3 agents des moyennes des rewards sur 100 trajectoires tout les 1000 trajectoires

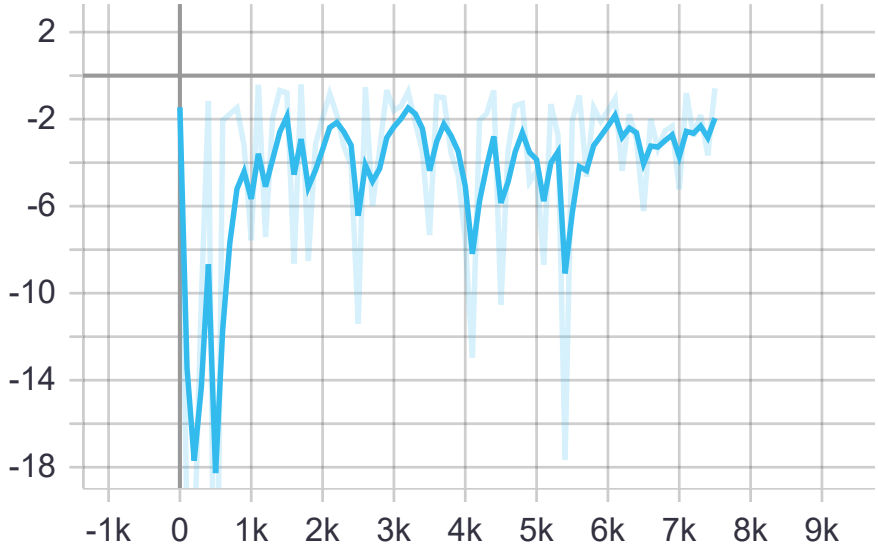


Fig. 4. Moyennes des rewards de l'agent 0 sur 100 trajectoires tout les 1000 trajectoires

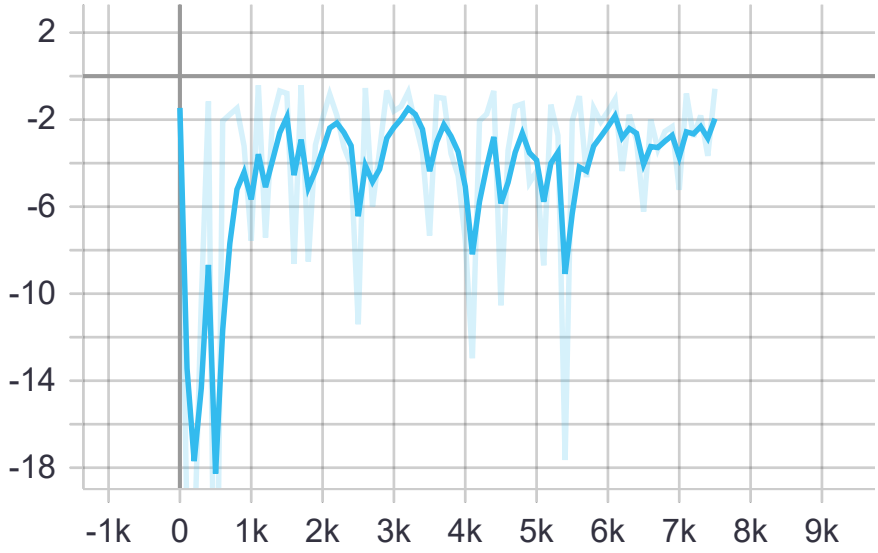


Fig. 5. Moyennes des rewards de l'agent 1 sur 100 trajectoires tout les 1000 trajectoires

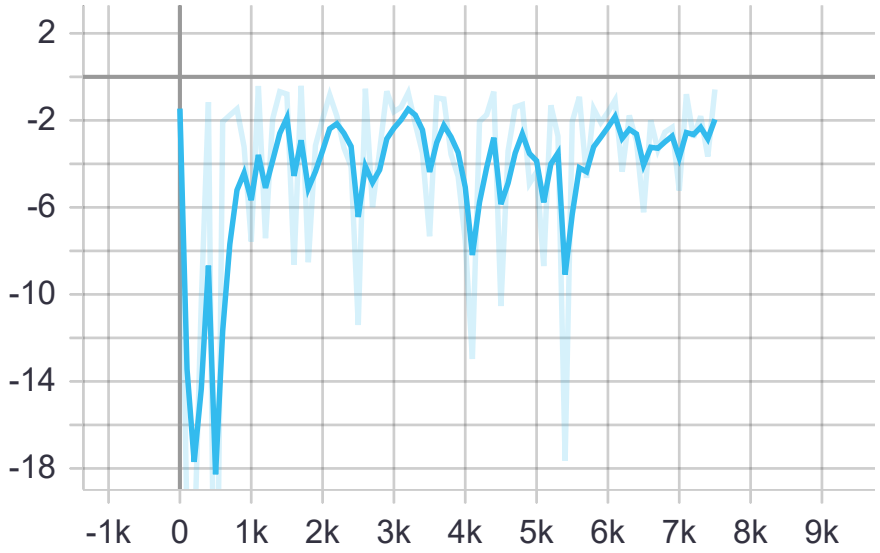


Fig. 6. Moyennes des rewards de l'agent 2 sur 100 trajectoires tout les 1000 trajectoires

Simple tag

L'apprentissage du modèle sur cet environnement n'est pas certain comme l'atteste la figure ci-dessous :

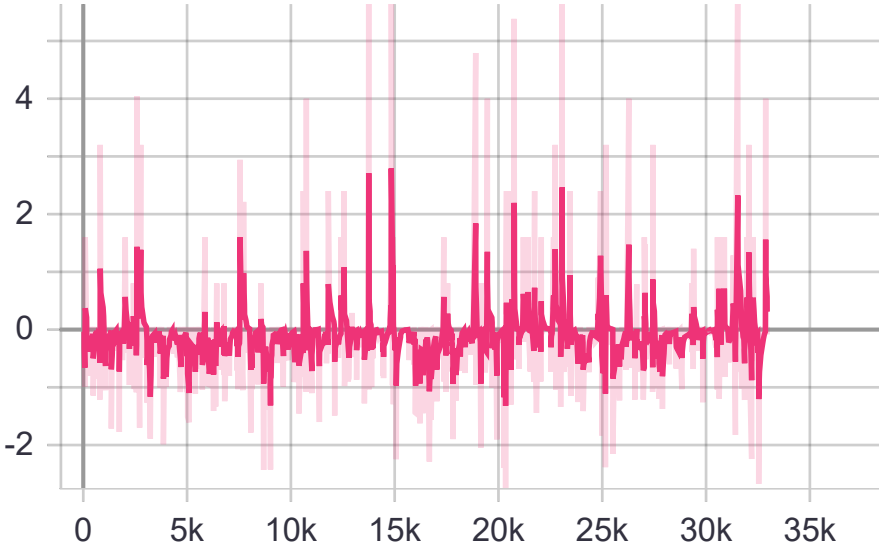


Fig. 7. Somme sur les 3 agents des moyennes des rewards sur 100 trajectoires tout les 1000 trajectoires (30000 première trajectoires)

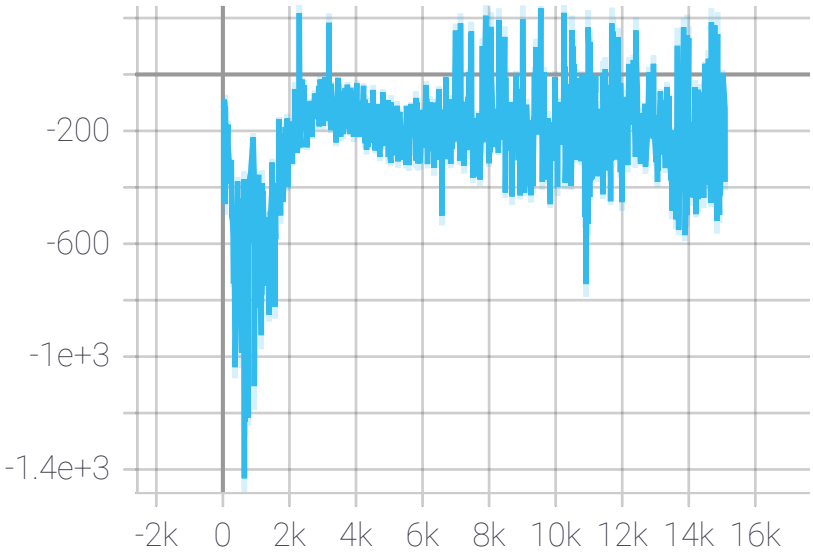
Par ailleurs, on remarque que les récompenses sont plus faible que pour les autres environnements, cela nécessite un ajustement des pas d'apprentissage.

Rapport TP11: Imitation Learning

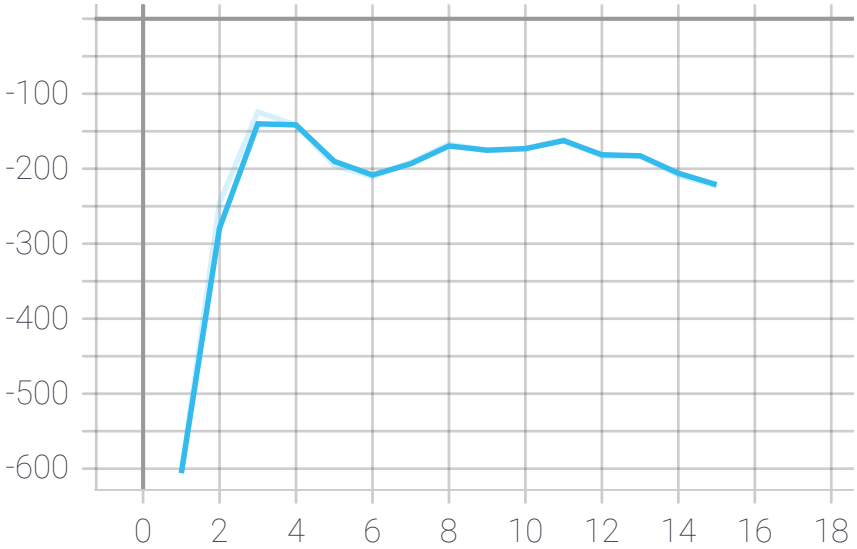
Victor Duthoit, Pierre Wan-Fat

1. Behavior cloning

On met dans un premier temps en place un agent *Behavior cloning* cherchant à maximiser la probabilité de faire les mêmes actions que l'expert. Les figures ci-dessous montrent qu'un tel agent arrive à apprendre quelque peu. On note qu'il arrive à effectuer des trajectoires à récompenses positives. Néanmoins, il est probable que ces trajectoires partent d'un état initial proche de l'état initial de la trajectoire experte. Ainsi, il est possible pour l'agent de suivre complètement les actions choisies par l'expert. Néanmoins, l'agent ne saurait pas quoi faire pour des états qui n'ont pas été rencontrés dans la trajectoire experte. Cet effet s'accentue avec le sur-apprentissage au fur et à mesure de la descente de gradient.



Récompenses par trajectoire effectuée



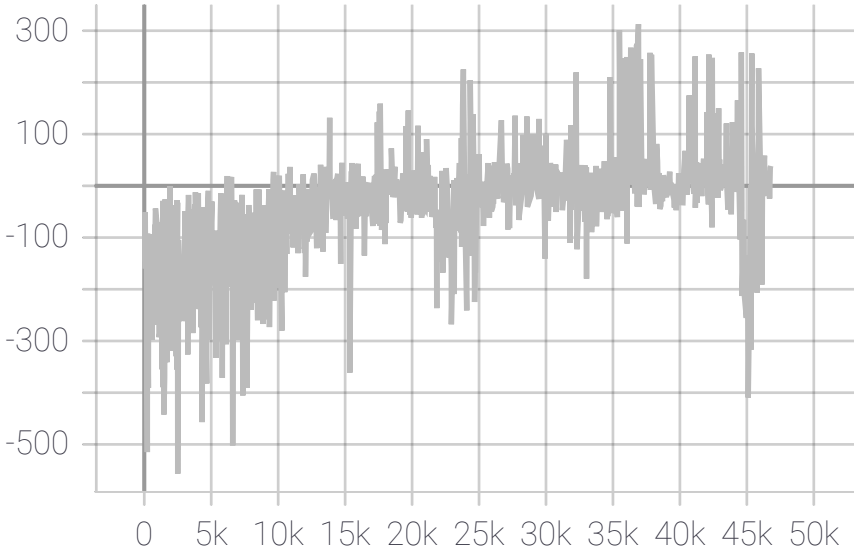
Récompenses sur moyenne de 100 trajectoires toutes les 1000 trajectoires

2. GAIL

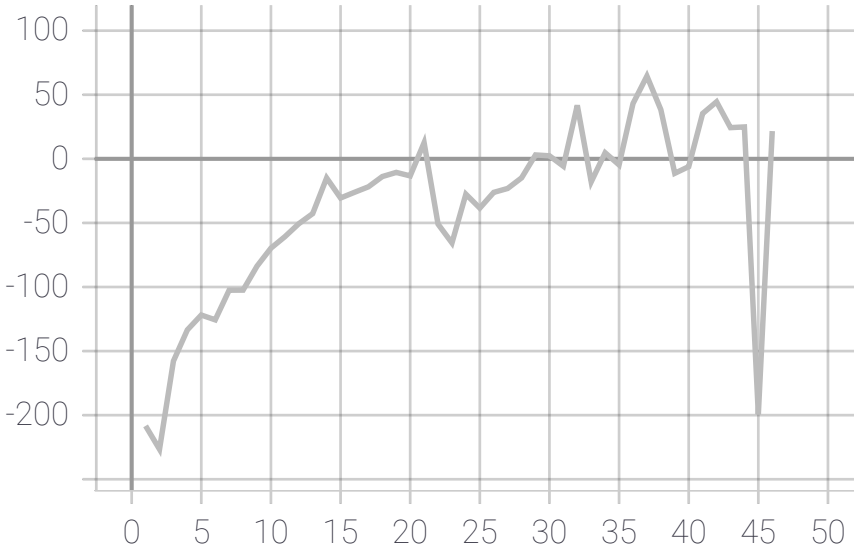
Dans cette partie, on a mis en place le modèle d'imitation par méthode adversaire. On utilise les mêmes hyperparamètres que proposés par l'énoncé.

Résultats

Après environ 20 000 événements, l'agent atteint des récompenses positives. Il atteint aux alentours des 35 000 événement des récompenses très satisfaisantes semblables à l'expert. On note toutefois une légère instabilité de l'agent qui bien que récupérant rapidement son apprentissage, retombe lors de quelques trajectoires à récompenses négatives. On peut imaginer une diminution de ϵ (limite du ratio de probabilité) au fur et à mesure de l'apprentissage pour limiter les pas catastrophiques.



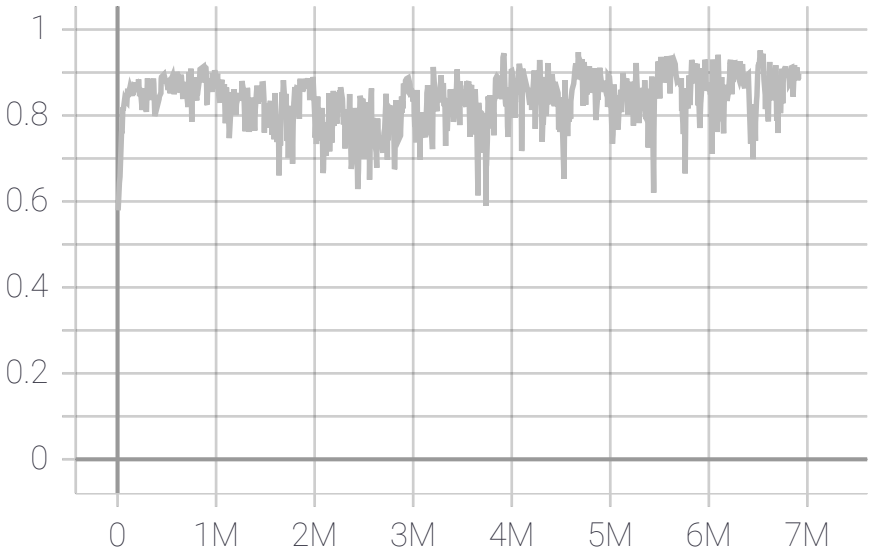
Récompenses par trajectoire effectuée



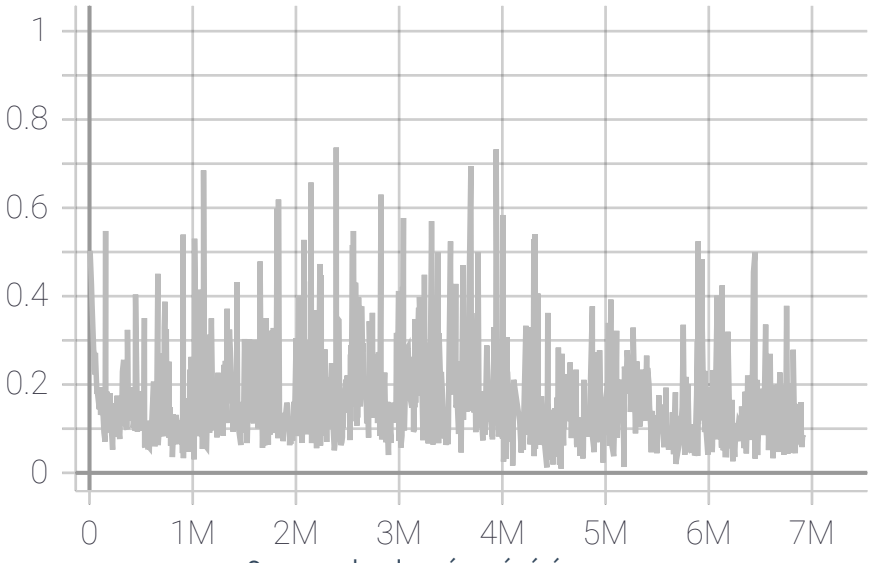
Récompenses sur moyenne de 100 trajectoires toutes les 1000 trajectoires

Le discriminant

On peut noter que les scores attribués aux données expertes et aux données générées sont relativement bien équilibrés. Les trajectoires expertes sont hautes en restant toutefois à une valeur acceptable (environ 0,9). Il n'y a apparemment pas de sur-apprentissage qui serait néfaste à l'agent. On remarque par ailleurs que l'agent est capable de parfois tromper son discriminant en générant des trajectoires proches de l'expert.

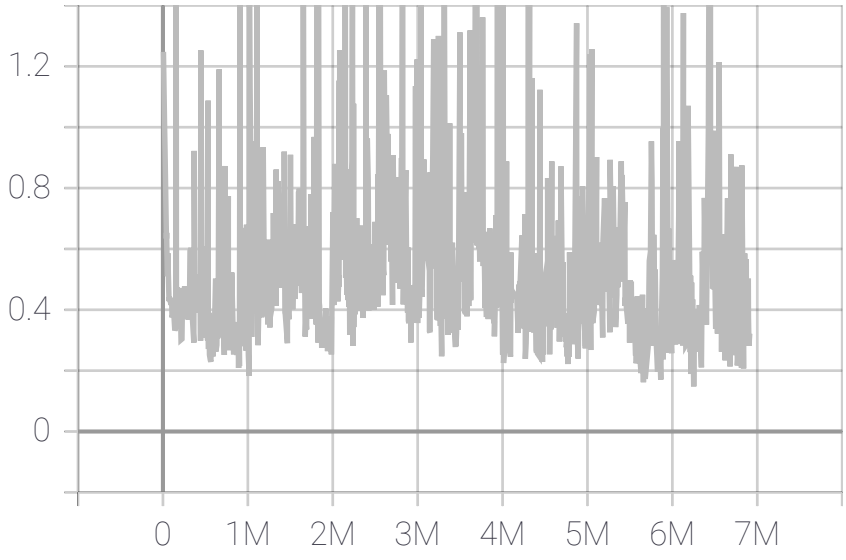


Score sur les données expertes



Score sur les données générées

Cet équilibre se retrouve dans les valeurs prises par la fonction de coût : le discriminant n'apprend pas trop vite, cela permet à l'agent d'avoir des récompenses éparses qui le guident vers les trajectoires expertes petit à petit.



Loss du discriminant

Rapport TME12 : Implicit Curriculum Learning

Victor Duthoit, Pierre Wan-Fat

1. DQN avec goals

Comme proposé, on met en place un agent DQN qui utilise les buts générés par l'environnement. Les récompenses ci-dessous montrent que l'agent apprend correctement et très rapidement.

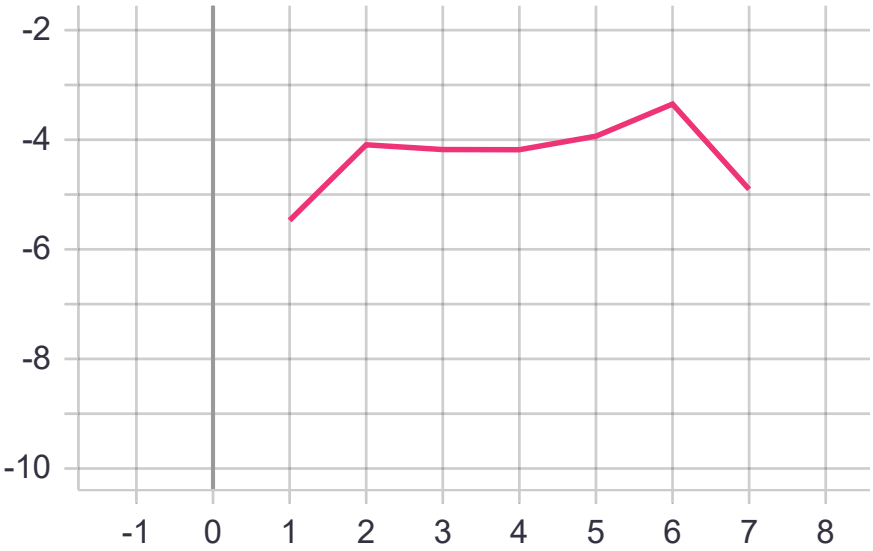


Fig. 1. Moyenne des rewards sur 100 trajectoires toutes les 1000 trajectoires

La récompense à l'épisode 0 est de -10 environ. L'agent a donc appris beaucoup lors du premier épisode.

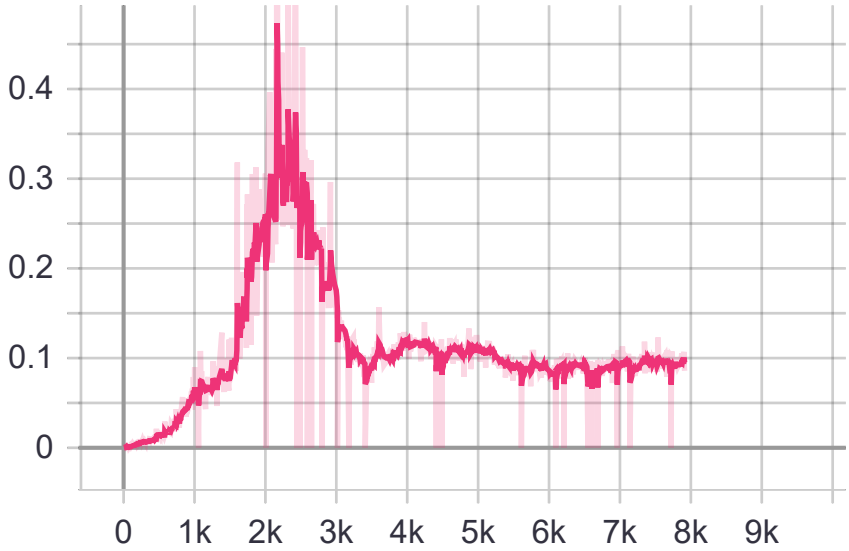


Fig. 2. Évaluation de la fonction de coût lors de l'apprentissage

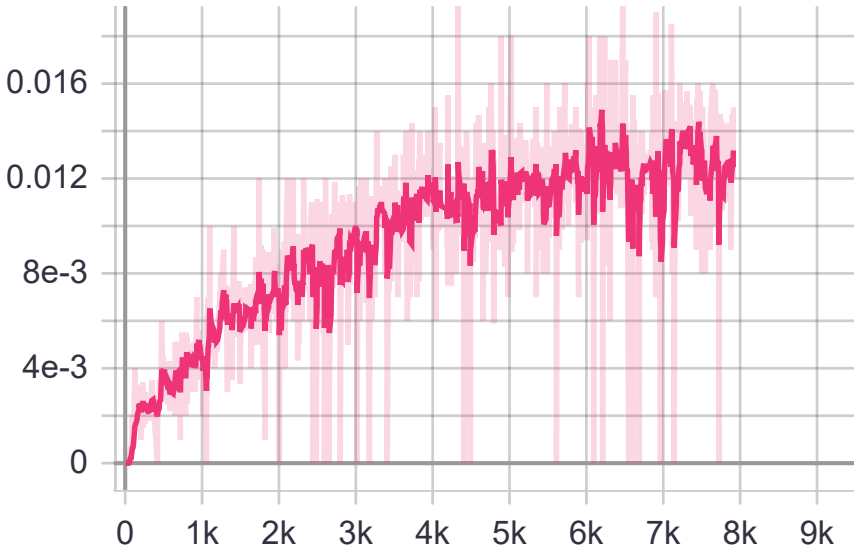


Fig. 3 Taux de présence de transition avec reward de 1 dans le Replay Buffer

On peut voir dans la figure 2 que la descente de gradient ne se produit pourtant qu'après 2000 trajectoires. En effet, il est nécessaire que l'agent récupère suffisamment de transition avec des récompenses de 1 pour les apprendre. De nombreuses tentatives ont échoué car l'agent ne récupérait pas assez de récompenses lors des premières trajectoires. On peut apprécier figure 3 le taux de récompenses égales à 1 dans le Replay Buffer. Il est important de garder ce taux suffisamment haut pour que la critique prenne en compte la récompense relativement *sparse* .

2. Hinsight Experience Replay

L'implémentation du modèle HER fonctionne correctement. Les premiers objectifs atteints arrivent après 2 000 trajectoires et les récompenses se densifient peu à peu, comme le montrent les figures ci-dessous. Par ailleurs, l'apprentissage ne semble pas encore stabilisé après 9 000 trajectoires.

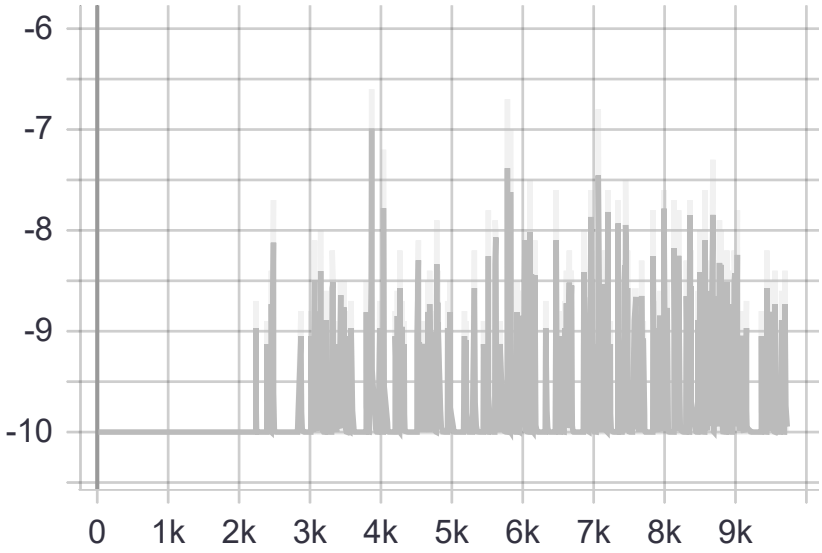


Fig. 4. Récompenses lors des 10 000 premières trajectoires

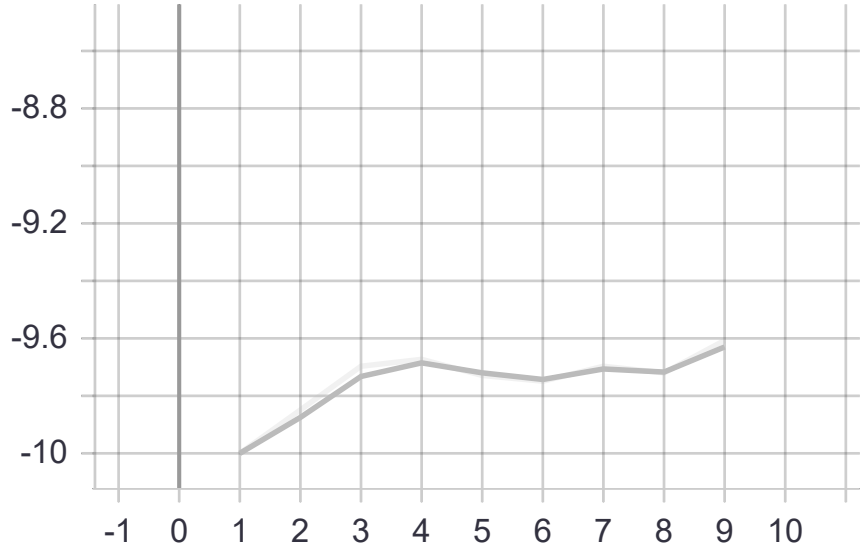


Fig. 5. Récompenses moyennes sur 100 trajectoires toutes les 1 000 trajectoires

En visualisant la position finale de chaque trajectoire, on peut voir apparaitre des "étages" qui peuvent être représentatifs des "salles" formées dans le **plan2.txt**. On remarque par ailleurs que la plupart des trajectoires après 2 000 trajectoires sont relativement proches de l'objectif. Atteindre l'objectif en un temps restreint peut donc être rendu difficile par la part d'actions aléatoires qui ne diminue pas avec le temps.

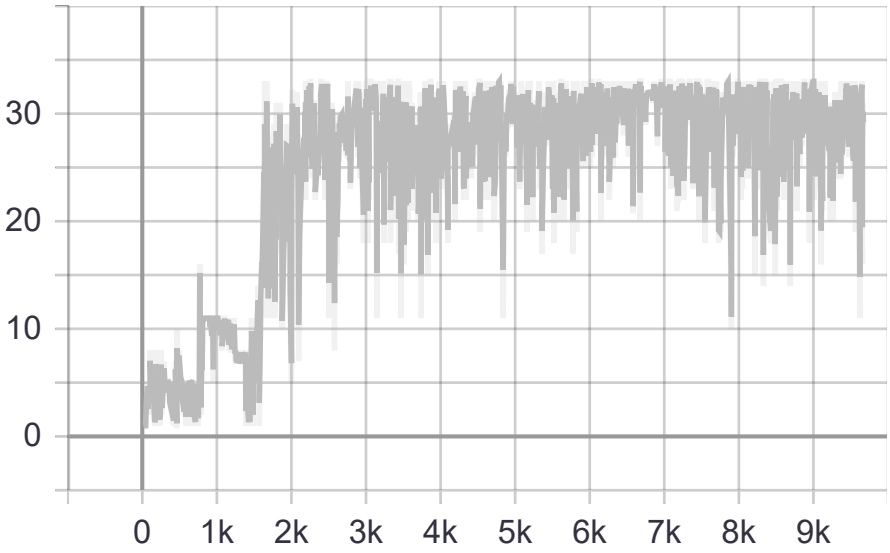


Fig. 6. Évolution de la position finale de l'agent (coordonnées x)

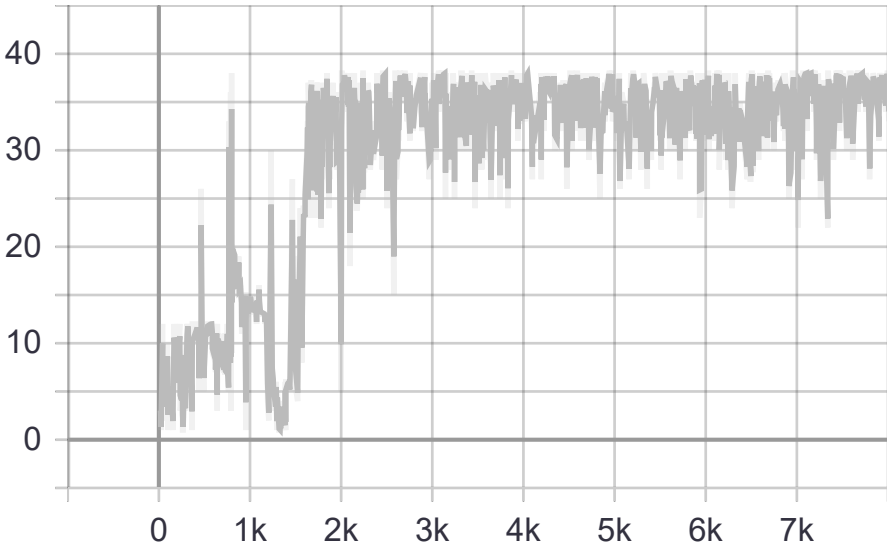


Fig. 7. Évolution de la position finale de l'agent (coordonnées y)

3. Échantillonnage itératif de buts

La mise en place du modèle Iterative Goal Sampling fonctionne correctement. L'agent apprend rapidement sur *gridworld3* . On note notamment un rapide apprentissage très important lors des 1 000 premières trajectoires.

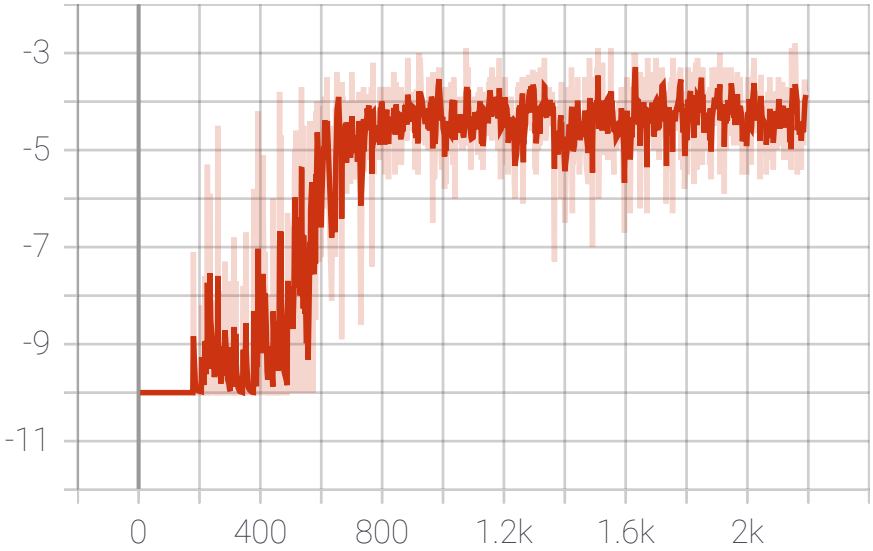


Fig. 8. Reward sur les 2 000 premières trajectoires

On note par ailleurs que la stabilisation des récompenses s'opère au même moment que la descente de gradient. En effet, on peut supposer que de nouveaux objectifs sont constamment ajoutés au buffer lors des 800 premières trajectoires mais ensuite, aucun nouvel objectif n'est ajouté, on peut espérer une descente de gradient plus stable.

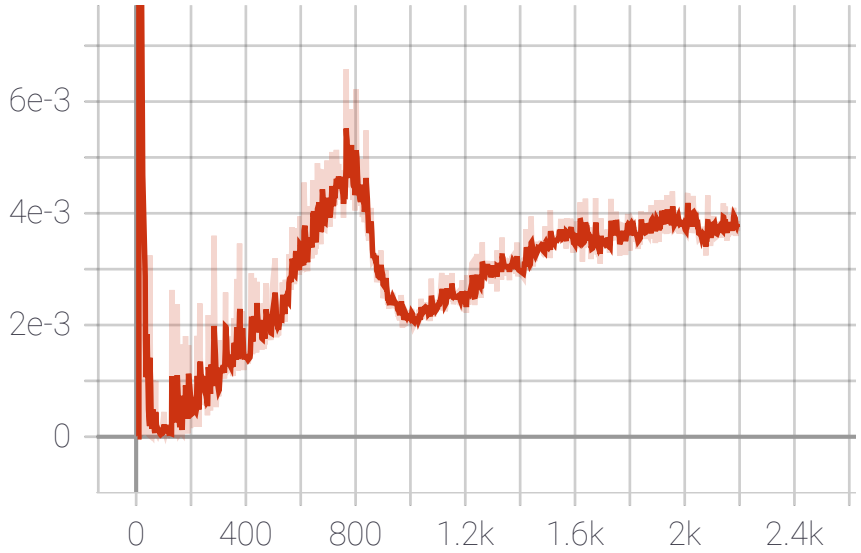


Fig. 9. Descente de gradient de la fonction de coût (TD)

Finalement, en dessinant les coordonnées du point final occupé par l'agent on remarque une évolution intéressante caractéristique de la topologie de l'environnement :

1. l'agent apprend à se déplacer vers la droite : zone $x = 10$
2. l'agent apprend à descendre : zone $y = 10$
3. l'agent apprend à aller vers la gauche : zone $x = 1$
4. l'agent apprend à remonter : zone $y = 1$
5. L'agent se déplace vers le centre : zone $x = 4$

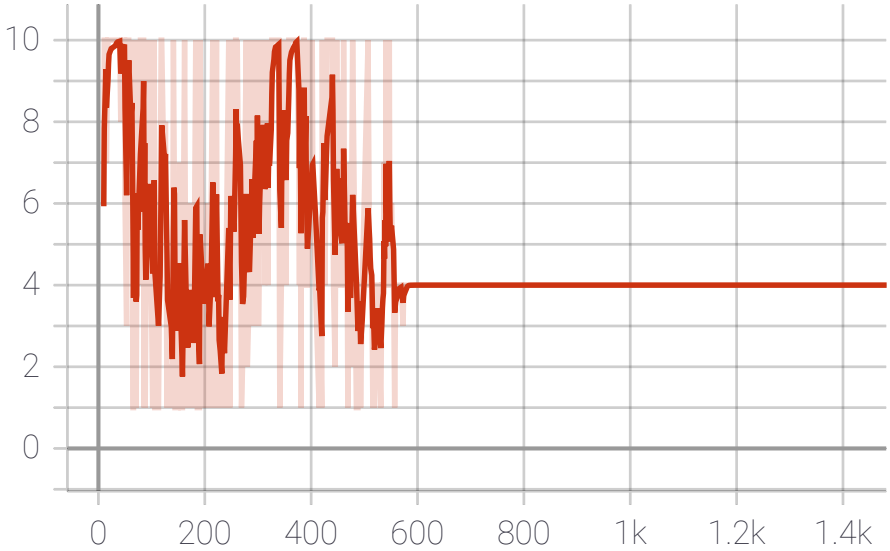


Fig. 10. Évolution de la position finale de l'agent (coordonnées x)

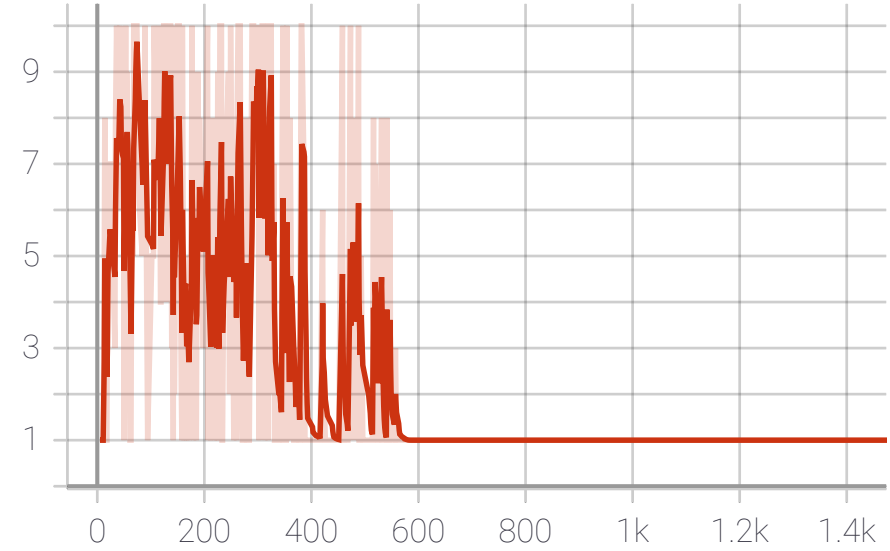


Fig. 11. Évolution de la position finale de l'agent (coordonnées y)

Finalement, on note une très grande stabilité dans l'état final qui sera atteint plus ou moins vite. Le paramétrage utilisé est le suivant :

- $\beta = 0.5$
- $\alpha = 0.1$