

# RLD – Rapport

Maxime DARRIN

17 février 2020

## Table des matières

<b>1 TP 1 – Bandits multi-bras</b>	<b>2</b>
1.1 Baselines . . . . .	2
1.2 UCB . . . . .	2
1.3 LinUCB . . . . .	3
<b>2 TP 2 – Programmation dynamique</b>	<b>5</b>
2.1 Policy Iteration . . . . .	5
2.2 Value Iteration . . . . .	6
<b>3 TP 3 – Q Learning</b>	<b>7</b>
3.1 SARSA . . . . .	7
3.2 Dyna Q . . . . .	7
<b>4 TP 4 – Deep Q Learning</b>	<b>9</b>
<b>5 TP 5 – Policy Gradient</b>	<b>9</b>
<b>6 TP 6 – PPO</b>	<b>11</b>
<b>7 TP 7 – DDPG</b>	<b>12</b>
<b>8 TP 8 – MADDPG</b>	<b>12</b>
<b>9 TP 9 – Generative Adversarial Networks</b>	<b>13</b>
<b>10 TP 10 – Variational Autoencoders</b>	<b>13</b>

# 1 TP 1 – Bandits multi-bras

## 1.1 Baselines

```
1 def random_policy():
2     """
3         Return a random walk of the agent, taking uniformly each possible action
4     """
5     return [np.random.randint(0, 10) for i in range(5000)]
6
7 def staticbest_policy(click_rates):
8     """
9         Takes the action which maximises the score on that trajectory
10    """
11    a = np.argmax(np.sum(click_rates, axis=0))
12    return [a for i in range(5000)]
13
14
15 def opt_policy(click_rates):
16     """
17         At each timestep takes the best action
18     """
19     return np.argmax(click_rates, axis=1)
```

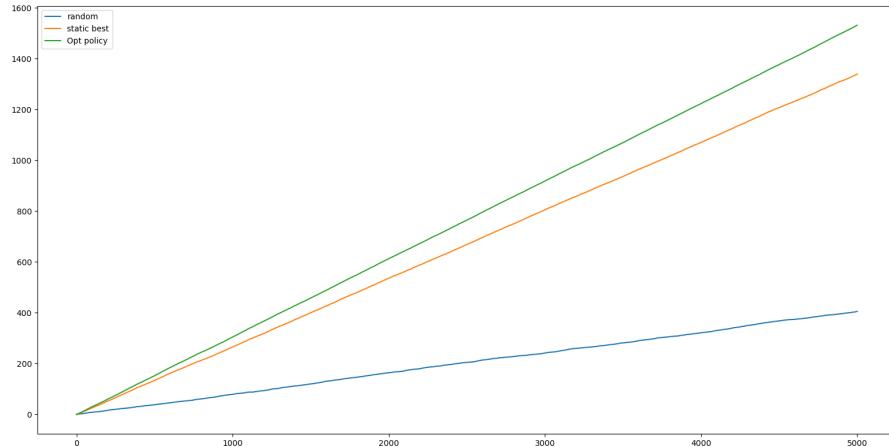


FIGURE 1 – Baselines – agents omniscients

## 1.2 UCB

```
1 def upper_bound(t, N, mu):
2     """
3         Computes the upper bound of the confidence interval of mean mu
4     """
5     return mu + np.sqrt(2 * np.log(t) / N)
6
7 def ucb_policy(click_rates):
8     """
9         Return the trajectory followed by the agent using ucb policy.
```

```

10      """
11
12      # Cumulative reward got by each actions
13      histo = np.array([click_rates[i][i] for i in range(10)])
14
15      # Number of times we took each action
16      counter = [1 for i in range(10)]
17
18      # List of the taken action
19      action_list = [i for i in range(10)]
20
21      for t in range(10, 5000):
22          action = np.argmax([upper_bound(t, counter[i], histo[i] / counter[i])
23                               for i in range(10)])
24          counter[action] += 1
25          histo[action] += click_rates[t][action]
26
27          action_list.append(action)
28      return action_list

```

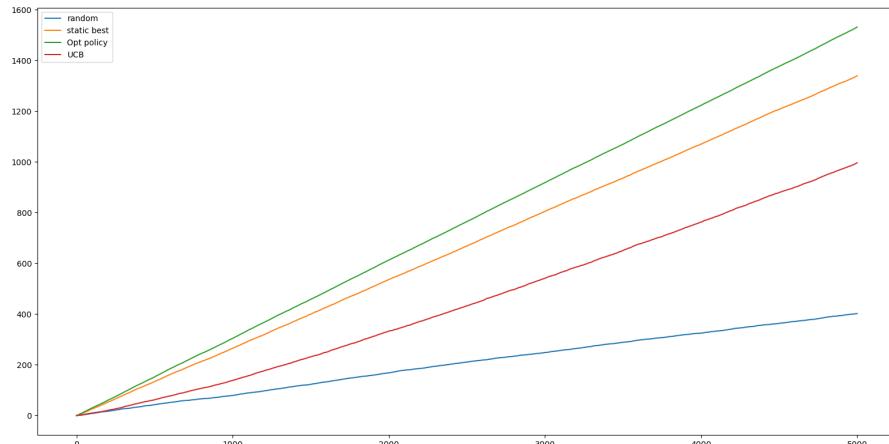


FIGURE 2 – Algorithme UCB VS Baselines

### 1.3 LinUCB

```

1 def linucb_policy(alpha, articles, click_rates):
2     A = [np.identity(5, dtype=float) for i in range(10)]
3     b = [np.zeros((5, 1), dtype=float) for i in range(10)]
4
5     theta = [None for i in range(10)]
6     pt = [None for i in range(10)]
7
8     actions_list = []
9
10    for t in range(0, 5000):
11        for i in range(10):
12            theta[i] = np.dot(np.linalg.inv(A[i]), b[i])

```

```

14             pt[i] = (np.dot(np.transpose(theta[i]), articles[t]) + alpha * np.sqrt(
15                 np.dot(
16                     np.dot(np.transpose(articles[t]),
17                         np.linalg.inv(A[i])),
18                         articles[t]))) [0]
19
20             at = np.argmax(pt)
21             rt = click_rates[t][at]
22
23             A[at] = A[at] + np.dot(np.transpose(articles[t]), articles[t])
24             b[at] = b[at] + rt * articles[t]
25
26             actions_list.append(at)
27
28     return actions_list

```

Dans figure suivante on compare *Lin UCB* aux *baselines* en faisant varier  $\alpha$ . En particulier on vérifie bien qu'un  $\alpha$  bas correspond à une forte exploration (ici  $\alpha = 0.01$  donne des résultats proches de l' aléatoire) tandis que des valeurs plus importantes permettent un meilleur compromis exploration-exploitation.

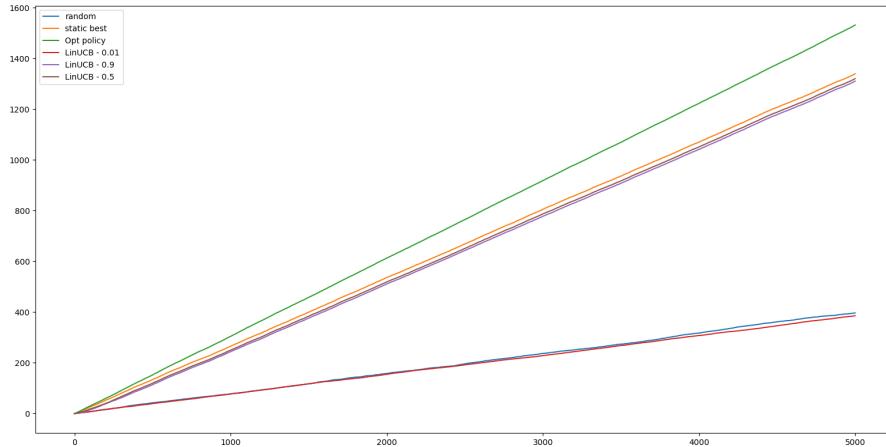


FIGURE 3 – Algorithme LinUCB VS Baselines

Par ailleurs, on observe bien que *lin UCB* est clairement meilleur que *UCB*. En effet, en utilisant le contexte pour prendre des décisions plus averties.

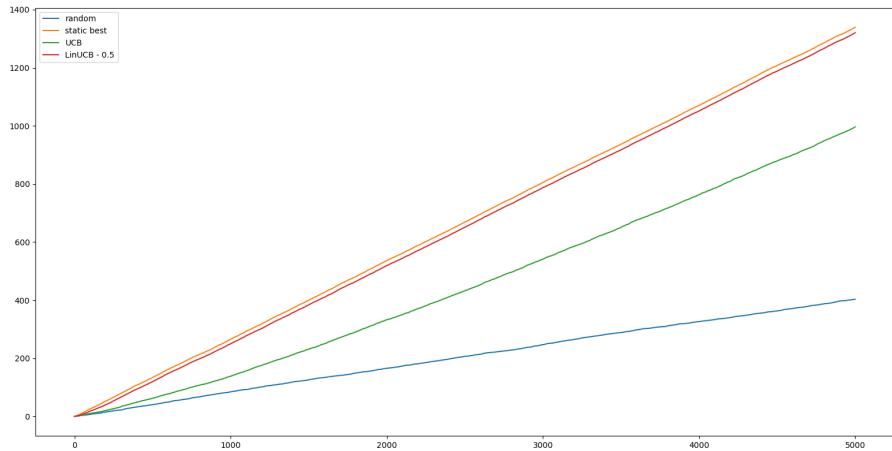


FIGURE 4 – Algorithme LinUCB VS UCB

## 2 TP 2 – Programmation dynamique

Les méthodes présentées ici supposent que l'on a accès à la chaîne de Markov sous-tendant l'environnement dans lequel on évolue.

### 2.1 Policy Iteration

Implémentation de l'entraînement de l'algorithme de policy iteration.

```

1   policy = np.zeros(max(self.states) + 1)
2   self.policy = np.random.randint(0, self.actions, max(self.states) + 1)
3
4   while True:
5       V = np.random.uniform(-5, 5, max(self.states) + 1)
6
7       while True:
8           Vprev = np.copy(V)
9           for s in self.states:
10               V[s] = sum([p * (r + gamma * Vprev[self.state_dict[sp]]) for p, sp, r, done in self.P[s][self.policy[s]]])
11
12           err = np.sum(np.fabs(Vprev - V))
13           print(err)
14           if err <= eps:
15               break
16
17       for s in self.states:
18           policy[s] = np.argmax([sum([p * (r + gamma * V[self.state_dict[sp]]) for p, sp, r, done in self.P[s][a]]) for a in range(self.actions)])
19
20       if all(policy == self.policy):
21           self.policy = policy
22           break
23
24
25
26

```

```

27     self.policy = policy
28
29     return policy

```

## 2.2 Value Iteration

Implémentation de l'algorithme de value iteration.

```

1   V = np.random.uniform(-5, 5, max(states)+1)
2   Q = np.zeros((max(states)+1, actions))
3
4   V_prev = np.copy(V)
5
6   for i in range(max_iter):
7       for s in states:
8           V_prev = np.copy(V)
9           for a in range(0,actions):
10              Q[s, a] = np.sum([p * (r + gamma * V_prev[self.state_dict[sp]])
11                                for p, sp, r, done in P[s][a]])
12
13      V[s] = np.max(Q[s, :])
14
15      if np.sum(np.fabs(V_prev - V)) <= eps:
16          self.Q = Q
17          self.V = V
18          return Q, V, True, i
19
20
21 self.Q = Q
22 self.V = V

```

### 3 TP 3 – Q Learning

#### 3.1 SARSA

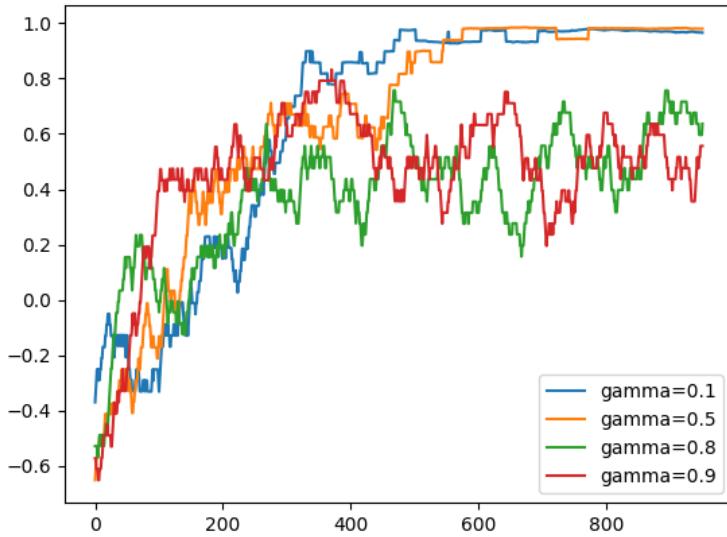


FIGURE 5 – Algorithme Sarsa

#### 3.2 Dyna Q

```
1  def updateQ(self, st, at, rt, stp, alpha=0.9, gamma=0.9, alphar=0.8):
2
3      st = self.state_dict[st.dumps()]
4      stp = self.state_dict[stp.dumps()]
5      self.Q[st, at] = (1 - alpha) * self.Q[st, at] + alpha * (rt + gamma * np.max(self.Q[stp, :]))
6
7      self.R[st, at, stp] = (1 - alphar) * self.R[st, at, stp] + alphar * rt
8      self.P[stp, st, at] = (1 - alphar) * self.P[stp, st, at] + alphar
9
10     for s in range(max(self.states) + 1):
11         if s != stp:
12             self.P[s, st, at] = (1 - alphar) * self.P[s, st, at]
13
14     self.P = softmax(self.P)
15
16     states = list(self.state_dict.values())
17
18     s = list(np.random.choice(states, self.k))
19     a = list(np.random.choice(self.actions, self.k))
20
21     for i in range(self.k):
22         self.Q[s[i], a[i]] = (1 - alpha) * self.Q[s[i], a[i]] + alpha * \
23                             (sum([self.P[sp, s[i], a[i]] * (self.R[s[i], a[i], sp] +
24                                              gamma * np.max(self.Q[sp, :])) for sp in s]))
```

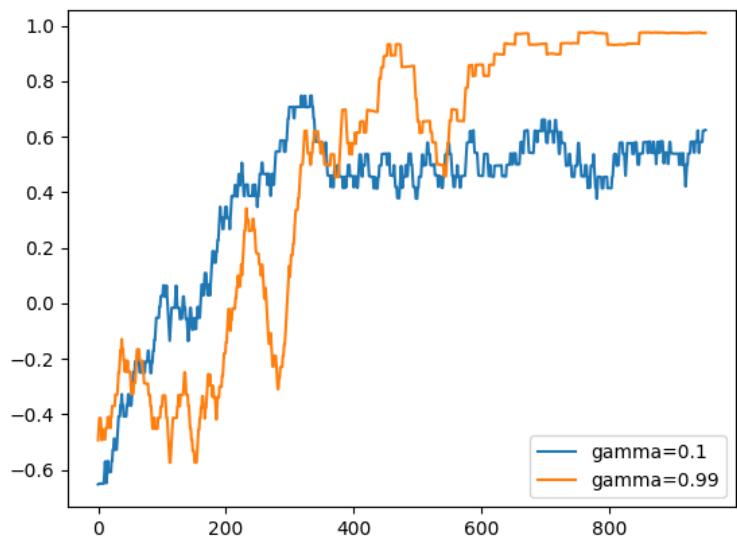


FIGURE 6 – Algorithme DynaQ

## 4 TP 4 – Deep Q Learning

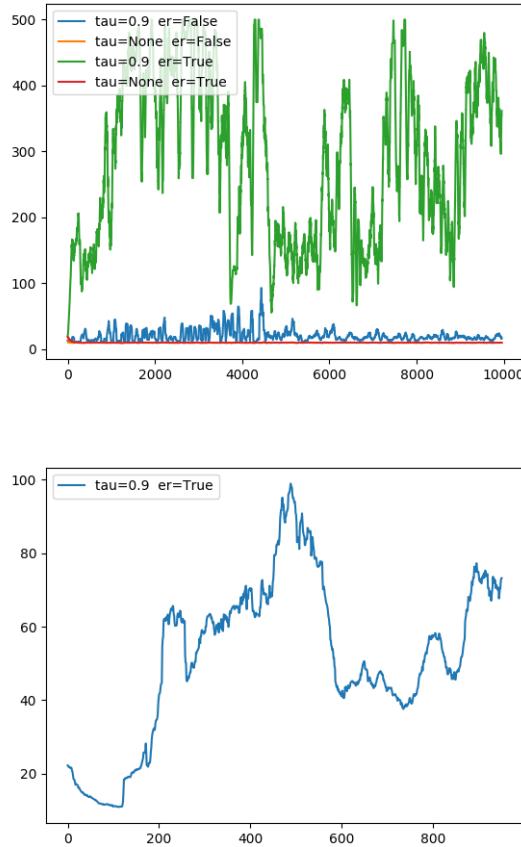


FIGURE 7 – Entraînement et utilisation de DQL pour CartPole

## 5 TP 5 – Policy Gradient

```
1 def batch_training(self, trajectory):
2     Y = []
3     X = []
4     cumulative_reward = 0
5
6     traj = copy(trajectory)
7     traj.reverse()
8
9     for state, action, reward, next_state, done in traj:
10         if not done:
11             cumulative_reward = reward + self.gamma * cumulative_reward
12         else:
13             cumulative_reward = reward
14
15         y = (1-self.alpha) * self.V(state).detach().numpy() + self.alpha * cumulative_reward
16
17         X.append(state)
18         Y.append(y)
19
```

```

20     self.update_value_function(X, Y)
21
22     self.policy.zero_grad()
23
24     logpi = - sum(torch.log(self.policy(state)[action]) *
25                   self.advantage_function(reward, state, next_state)
26                   for state, action, reward, next_state, done in traj) / len(traj)
27
28     logpi.backward()
29
30     self.policy_optimizer.step()
31     self.policy_optimizer.zero_grad()
32     self.policy.zero_grad()
33
34 def update_policy(self, state, action, reward, next_state):
35     self.policy.zero_grad()
36
37     logpi = torch.log(self.policy(state)[action])
38     logpi.backward()
39
40     A = self.advantage_function(reward, state, next_state)
41     for p in self.policy.parameters():
42         p.grad *= - A
43
44
45     self.policy_optimizer.step()
46     self.policy_optimizer.zero_grad()
47     self.policy.zero_grad()
48
49 def advantage_function(self, r, state, next_state):
50     return r + self.gamma*self.V(next_state) - self.V(state)

```

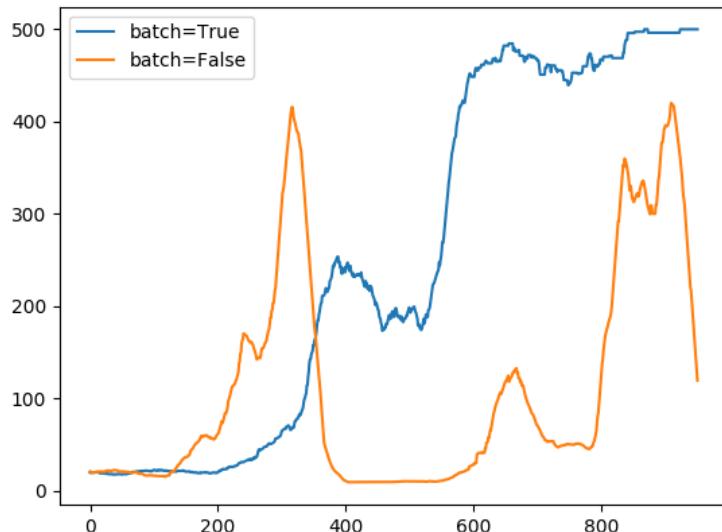


FIGURE 8 – A2C avec et sans batch sur cartpole

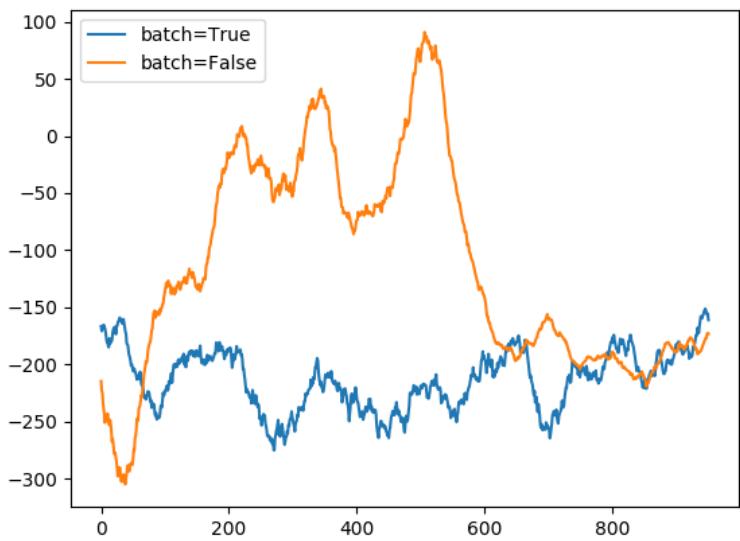


FIGURE 9 – A2C avec et sans batch sur LunarLander

## 6 TP 6 – PPO

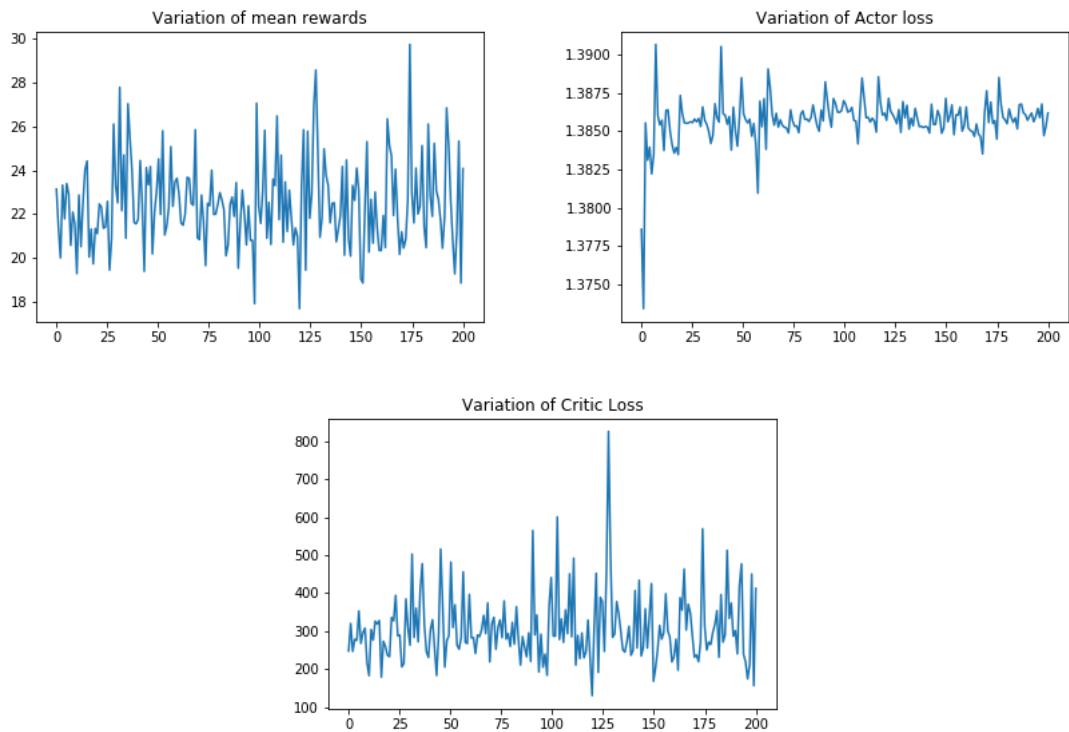


FIGURE 10 – Entraînement et utilisation de PPO pour CartPole

## 7 TP 7 – DDPG

Dans ce cadre on s'intéresse aux environnements dont l'espace des actions est continu. On va donc entraîner une policy continue avec un critic par policy gradient.

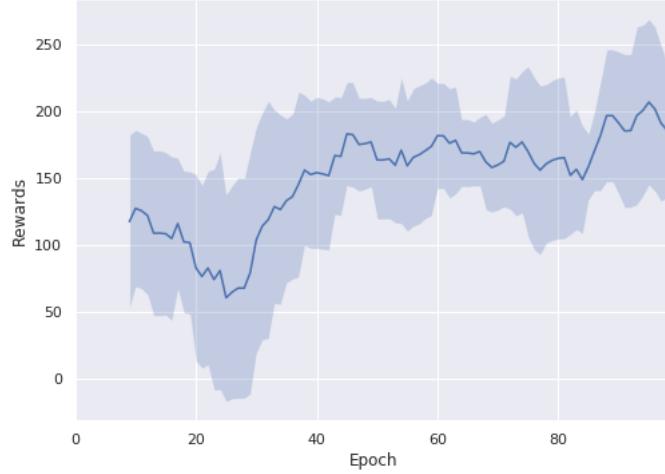


FIGURE 11 – Entraînement et utilisation de DDPG pour MountainCar

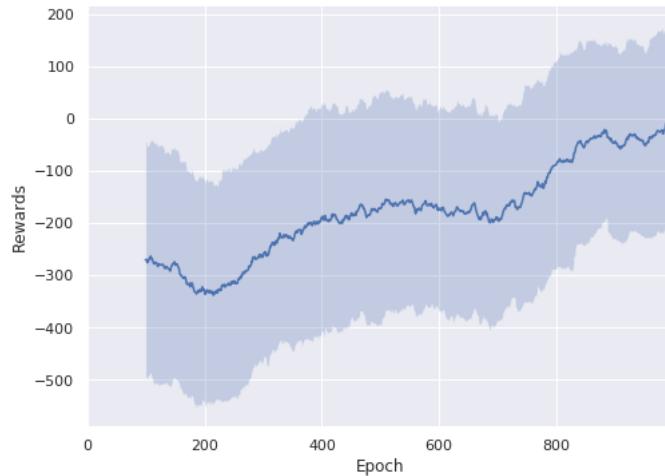


FIGURE 12 – Entraînement et utilisation de DDPG pour LunarLander

## 8 TP 8 – MADDPG

On se rend rapidement compte qu'appliquer les méthodes usuels au cas multi agent ne fonctionne pas. Cela s'explique par le fait que lors de l'entraînement on a besoin de la propriété de markov sur les états. Or on perd cette propriété lorsqu'il y a plusieurs agents pour chaque agent. En effet, l'état suivant ne dépend pas seulement de ce que voit l'agent lui même et de ses actions mais aussi des actions prises par les autres agents.

On règle ce problème en passant à la critique les informations sur tous les agents pendant l'entraînement. On peut ensuite utiliser les agents en phase de production sans les critiques (et donc sans l'aspect omniscient).

Je n'ai pas réussi à obtenir de résultats corrects (d'autant que depuis quelques mј la librairie a cessé de fonctionner.)

## 9 TP 9 – Generative Adversarial Networks

L'objectif des Generative Adversarial Networks est d'entraîner à la fois un générateur et un classifieur. Le classifieur est entraîné à détecter les contrefaçons produites par le générateur, par rapport au jeu de données original. Le générateur est lui entraîné à essayer de berner le classifieur. En entraînant chacun à son tour, on obtient un générateur approximant la distribution réelle des données.



FIGURE 13 – Exemples de visages construits par le générateur au fur et à mesure de l'entraînement

## 10 TP 10 – Variational Autoencoders

L'objectif d'un *Variational Autoencoder* est d'apprendre un encoder et un decoder comme dans un autoencoder usuel à la différence près que l'encoder du VAE va apprendre les paramètres d'une distribution de probabilité (une loi normale de paramètres  $\mu, \sigma$ ) et que lors de l'entraînement on échantillonne cette distribution à l'entrée du decoder. Cela permet d'avoir un générateur d'échantillons décodés à partir d'un bruit gaussien passé en entrée du decoder (voir figure 15).

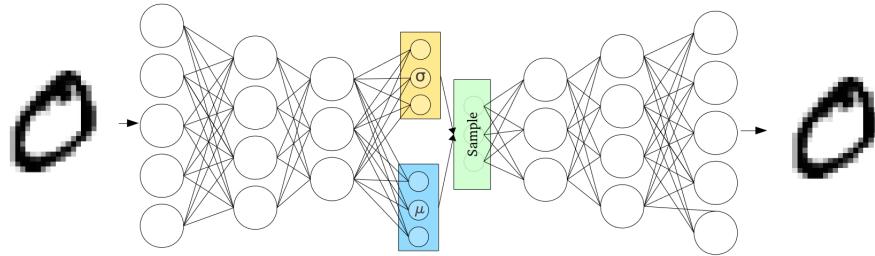


FIGURE 14 – Schéma d'un VAE



FIGURE 15 – Exemples d'échantillons produits par le decoder à partir d'un bruit Gaussien



FIGURE 16 – Comparaison de la reconstruction avec les entrées