

Introduction au DQN avec PyTorch et gym

Dans ce TP, vous allez apprendre à créer un réseau de neurones profond d'estimation de la qualité (DQN) pour résoudre des problèmes d'apprentissage par renforcement en utilisant PyTorch et gym.

Pour ce faire, vous allez suivre les étapes suivantes:

- Installer PyTorch et gym sur votre ordinateur
- Définir un modèle DQN simple en utilisant PyTorch
- Entraîner le modèle DQN en tirant sur des épisodes de l'environnement
- Expérimenter avec différents environnements, hyperparamètres et architectures pour voir comment ils affectent la performance du DQN

Avant de commencer, assurez-vous d'avoir installé Python 3 et les bibliothèques nécessaires sur votre ordinateur. Si vous avez des questions, n'hésitez pas à demander de l'aide à votre enseignant ou à vos camarades. Bon travail!

Rappels

Le DQN apprend à jouer à un jeu en interagissant avec l'environnement et en essayant de maximiser la récompense future. Il fait cela en utilisant un réseau de neurones pour estimer la fonction de valeur, qui est une estimation de la somme des récompenses futures attendues à partir de l'état et de l'action actuelles. Le DQN apprend à jouer en prenant des actions qui maximisent la fonction de valeur estimée et en ajustant ses paramètres en fonction de la différence entre la fonction de valeur estimée et la cible de la fonction de valeur.

Objectifs

- Apprendre les concepts de base de l'apprentissage par renforcement et de l'estimation de la qualité
- Installer et utiliser PyTorch et gym pour créer des modèles DQN
- Comprendre les différents éléments d'un DQN, tels que les fonctions de perte et d'optimisation
- Expérimenter avec différents environnements, hyperparamètres et architectures pour améliorer la performance du DQN

En réalisant ce TP, les étudiants devraient être en mesure de créer des modèles DQN simples et d'utiliser les outils PyTorch et gym pour les entraîner et les tester. Ils devraient également comprendre comment différents éléments d'un DQN peuvent affecter son apprentissage et sa performance.

Ressources

Voici une liste de ressources que vous pouvez utiliser en complément du cours pour en savoir plus sur les DQN, PyTorch et l'apprentissage par renforcement:

- Vidéos YouTube:
 - [Introduction to Deep Q-Networks \(DQN\) with PyTorch](#)
 - [PyTorch Tutorial for Beginners \(Full Course\)](#)
 - [Reinforcement Learning Basics with OpenAI Gym](#)
- Livres:
 - [Deep Learning with PyTorch](#)
 - [Reinforcement Learning: An Introduction](#)
 - [Hands-On Reinforcement Learning with PyTorch](#)

Ces ressources vous permettront de vous familiariser avec les concepts de base des DQN, de PyTorch et de l'apprentissage par renforcement, et de vous fournir des exemples et des astuces pour utiliser ces outils dans vos propres projets. Vous pouvez également consulter d'autres ressources en ligne, telles que des tutoriels, des forums et des documentations pour en apprendre davantage.

RAPPEL : 1/4 de la note finale est liée à la mise en forme :

- Pensez à nettoyer les outputs inutiles (installation, messages de débogage, ...)
- Soignez vos figures : les axes sont-ils faciles à comprendre ? L'échelle est adaptée ?
- Commentez vos résultats : vous attendiez-vous à le avoir ? Est-ce étonnant ? Faites le lien avec la théorie.

Ce TP reprend l'exemple d'un médecin et de ses vaccins. Vous allez comparer plusieurs stratégies et trouver celle optimale. Un TP se fait en groupe de 2 à 4. Aucun groupe de plus de 4 personnes.

Vous allez rendre le TP dans une archive ZIP. L'archive ZIP contient ce notebook au format `.ipynb`, mais aussi exporté en PDF & HTML. L'archive ZIP doit aussi contenir un fichier txt appelé `groupe.txt` sous le format:

```
Nom1, Prenom1, Email1, NumEtudiant1
Nom2, Prenom2, Email2, NumEtudiant2
Nom3, Prenom3, Email3, NumEtudiant3
Nom4, Prenom4, Email4, NumEtudiant4
```

Un script vient extraire vos réponses : ne changez pas l'ordre des cellules et soyez sûrs que les graphes sont bien présents dans la version notebook soumise.

Q1: Quelle est la différence entre un réseau neuronal profond (DQN) et un réseau neuronal classique ?

La principale différence entre un réseau neuronal profond et un réseau neuronal classique est que les réseaux neuronaux profonds ont plusieurs couches cachées, ce qui leur permet d'apprendre des modèles plus complexes à partir de données. Les réseaux neuronaux classiques, en revanche, ont généralement seulement une seule couche cachée, ce qui limite leur capacité à apprendre des modèles complexes. Les réseaux neuronaux profonds sont donc souvent plus performants que les réseaux neuronaux classiques pour des tâches telles que la reconnaissance d'images ou le traitement du langage naturel.

Q2. Dans quel type de problème est utilisé le DQN ?

Dans un problème d'apprentissage par renforcement, l'objectif est de prendre des décisions pour maximiser une récompense à long terme. Le DQN est entraîné pour prédire la récompense future associée à chaque action possible dans un environnement donné, afin de déterminer la meilleure action à prendre. Par exemple, le DQN pourrait être utilisé pour entraîner un robot à se déplacer dans un environnement complexe en prenant des décisions pour éviter les obstacles et atteindre ses objectifs.

Q3. Quelle est la différence entre l'espace des actions et des états dans l'environnement `mountaincar-v0` et `cartpole-v0` ? Comment pouvez-vous visualiser cet environnement ?

In [1]: `%pip install gym`

```
Collecting gym
  Downloading gym-0.26.2.tar.gz (721 kB)
    Installing build dependencies ... done
    Getting requirements to build wheel ... done
    Preparing wheel metadata ... done
Requirement already satisfied: numpy==1.18.0 in /home/leme/RELE/.venv/lib/python3.8/site-packages (from gym) (1.23.5)
Collecting importlib-metadata==4.8.0; python_version < "3.10"
  Downloading importlib_metadata-4.8.0-py3-none-any.whl (21 kB)
Collecting gym-notices==0.0.4
  Downloading gym_notices-0.0.8-py3-none-any.whl (3.0 kB)
Collecting cloudpickle==1.2.0
  Downloading cloudpickle-2.2.0-py3-none-any.whl (25 kB)
Collecting zipp==0.5
  Downloading zipp-3.11.0-py3-none-any.whl (6.6 kB)
Building wheels for collected packages: gym
  Building wheel for gym (PEP 517) ... done
  Created wheel for gym: filename=gym-0.26.2-py3-none-any.whl size=827636 sha256=2f575e8d38da7a95398dd276cc1a79ee123f6b3d896f3643b0ab4cde1378bc
Successfully built gym
Installing collected packages: zipp, importlib-metadata, gym-notices, cloudpickle, gym
Successfully installed cloudpickle-2.2.0 gym-0.26.2 gym-notices-0.0.8 importlib-metadata-4.8.0 zipp-3.11.0
Note: you may need to restart the kernel to use updated packages.
```

In [6]: `import gym
import matplotlib.pyplot as plt

def visualize(env, title):
 # set the initial state of the environment
 state = env.reset()
 print("Shape of the action space of environment", env.action_space.shape)
 print("Observation space of environment", env.observation_space.shape)

 # initialize an empty list to store the state history
 state_history = []

 # define the number of steps to run the simulation for
 num_steps = 1000

 # run the simulation for the specified number of steps
 for step in range(num_steps):
 # choose a random action
 action = env.action_space.sample()

 # take a step in the environment using the chosen action
 state, reward, done, info, _ = env.step(action)

 # add the new state to the state history
 state_history.append(state)

 # check if the simulation is done
 if done:
 break

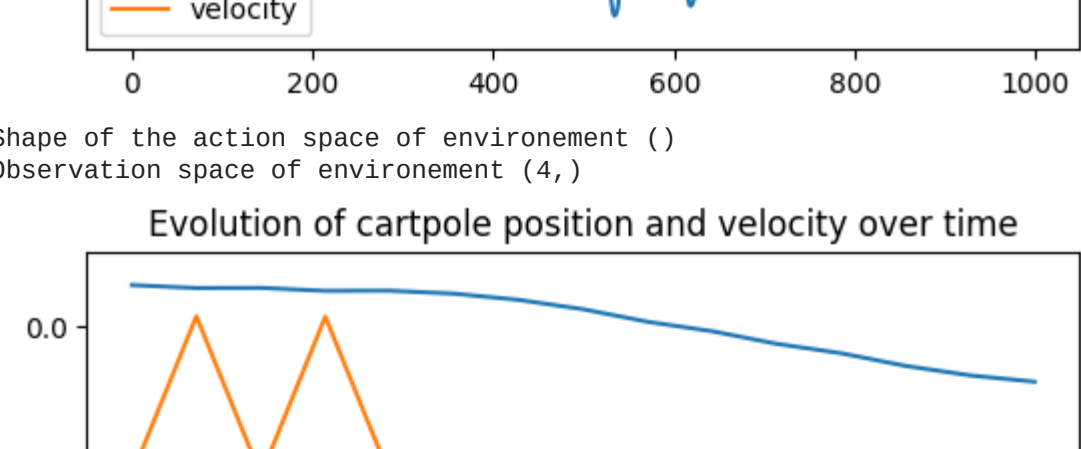
 # extract the position and velocity data from the state history
 positions = [s[0] for s in state_history]
 velocities = [s[1] for s in state_history]

 # plot the position and velocity data
 plt.plot(positions, label='position')
 plt.plot(velocities, label='velocity')
 plt.legend()
 plt.title(f"Evolution of {title} position and velocity over time")
 plt.show()


mountaincar = gym.make("MountainCar-v0")
visualize(mountaincar, "mountaincar")

cartpole = gym.make("CartPole-v0", render_mode="rgb_array")
visualize(cartpole, "cartpole")`

Shape of the action space of environment ()
Observation space of environment (2,)



Shape of the action space of environment ()
Observation space of environment (4,)



Dans l'environnement `MountainCar-v0`, l'espace d'état représente la position et la vitesse de la voiture sur la colline. L'espace des actions définit les différentes actions que l'agent (la voiture) peut effectuer, qui sont soit d'accélérer vers la gauche, soit d'accélérer vers la droite, ou de ne pas accélérer du tout.

Dans l'environnement `CartPole-v0`, l'espace d'état représente la position et la vitesse du chariot, ainsi que l'angle et la vitesse angulaire du bras suspendu. L'espace des actions définit les deux actions que l'agent (le chariot) peut effectuer, qui sont soit de pousser vers la gauche, soit de pousser vers la droite.

Dans l'environnement `MountainCar-v0`, l'espace d'état représente la position et la vitesse de la voiture sur la colline. L'espace des actions définit les différentes actions que l'agent (la voiture) peut effectuer, qui sont soit d'accélérer vers la droite, soit d'accélérer vers la gauche, ou de ne pas accélérer du tout.

Dans l'environnement `CartPole-v0`, l'espace d'état représente la position et la vitesse du chariot, ainsi que l'angle et la vitesse angulaire du bras suspendu. L'espace des actions définit les deux actions que l'agent (le chariot) peut effectuer, qui sont soit de pousser vers la gauche, soit de pousser vers la droite.

Pour visualiser les environnements, on peut réaliser une simulation de 1000 actions.

Avec les deux courbes on peut observer l'évolution de la position et de la vitesse. La simulation de l'environnement `MountainCar-v0`, la vitesse reste plus ou moins stable mais que la position change énormément. Pour une voiture qui essaie de monter sur une montagne ça semble logique.

Pour l'environnement `CartPole-v0`, on voit que la position est stable ce qui nous permet de conclure que sur 1000 actions les décisions prises sont plutôt bonne. Cependant, la vitesse est très instable ce qui nous permet de conclure que les décisions prises ne sont pas toujours bonnes.

Implémentation du DQN

L'architecture neuronale pour résoudre le Deep Q-Learning (DQN) est un réseau de neurones profond. Il est composé d'une couche d'entrée, d'une couche cachée et d'une couche de sortie. La couche d'entrée prend en entrée les données de l'environnement (position et vitesse de la voiture) et les convertit en un vecteur d'entrée. La couche cachée est composée de plusieurs couches de neurones qui traitent les données et les convertissent en un vecteur de sortie. La couche de sortie prend en entrée le vecteur de sortie de la couche cachée et produit une action à effectuer (accélérer, freiner ou ne rien faire).

Q4. Implémentez ce réseau.

In [14]: `import torch
import torch.nn as nn
import gym
import numpy as np

class DQN(nn.Module):
 def __init__(self, input_dim, output_dim):
 super(DQN, self).__init__()
 self.fc1 = nn.Linear(input_dim, 32)
 self.fc2 = nn.Linear(32, 32)
 self.fc3 = nn.Linear(32, output_dim)

 def forward(self, x):
 x = torch.relu(self.fc1(x))
 x = torch.relu(self.fc2(x))
 x = self.fc3(x)
 return x`

La fonction de perte est une fonction qui mesure la différence entre les valeurs prédites et les valeurs réelles. Dans le cas du DQN, la fonction de perte mesure la différence entre la valeur prédite par le réseau de neurones et la valeur réelle obtenue par l'environnement en utilisant la formule suivante :

La fonction de perte est donnée par :

$$L(\theta) = \mathbb{E}_{s_t, a_t \sim \rho_t} [(Q_{\theta}(s_t, a_t) - y_t)^2],$$

où θ est le vecteur de paramètres du réseau de neurones, ρ est la distribution de probabilité de l'état et de l'action, Q_{θ} est la fonction de valeur estimée par le réseau de neurones, et y_t est la cible de la fonction de valeur.

$$y_t = \mathbb{E}_{s_{t+1}, a_{t+1} \sim \rho_{t+1}} [r(s_t, a_t) + \gamma \max_{a'} Q_{\theta}(s_{t+1}, a_{t+1})].$$

Q5. Implémentez une fonction `compute_loss` telle que décrite par :

```
def compute_loss(
    model: nn.Module,
    state: torch.Tensor,
    next_state: torch.Tensor,
    reward: torch.Tensor,
    action: torch.Tensor,
    done: torch.Tensor,
    gamma: float = 0.99,
    batch_size: int = 32
):
    """
    Compute the DQN agent loss
    """
    q_values = model(state)
    q_next_state = model(next_state)
    q_values_target = reward + (1 - done) * gamma * q_next_state.max(dim=1)[0]
    loss = F.mse_loss(q_values[range(batch_size), action], q_values_target)
    return loss
```

```
# DQN Agent
class DQNAgent:
    def __init__(self, env, device):
        self.env = env
        self.device = device
        self.state_size = env.observation_space.shape[0]
        self.action_size = env.action_space.n
        self.memory = t.Deque[t.Tuple[np.ndarray, int, float, np.ndarray, bool]] = deque(maxlen=2000)
        self.batch_size = 32
        self.gamma = 0.95 # discount rate
        self.epsilon_min = 0.01
        self.epsilon_decay = 0.995
        self.learning_rate = 0.001
        self.model = self._build_model()

    def _build_model(self):
        # Neural Net for Deep-Q learning Model
        model = DQN(self.state_size, self.action_size).to(self.device)
        return model

    def remember(self, state: np.ndarray, action: int, reward: float, next_state: np.ndarray, done: bool):
        self.memory.append((state, action, reward, next_state, done))

    def act(self, state: np.ndarray) -> int:
        # Exploration : choisir une action aléatoire avec une probabilité epsilon
        if np.random.rand() <= self.epsilon:
            return self.env.action_space.sample()
        else:
            q_values = self.model(torch.from_numpy(state).float()).to(self.device)
            return torch.argmax(q_values).item()

    def replay(self):
        # Random replay
        minibatch = random.sample(self.memory, self.batch_size)
        states = torch.from_numpy(np.vstack([x[0] for x in minibatch])).float().to(self.device)
        actions = torch.from_numpy(np.vstack([x[1] for x in minibatch])).long().to(self.device)
        rewards = torch.from_numpy(np.vstack([x[2] for x in minibatch])).float().to(self.device)
        next_states = torch.from_numpy(np.vstack([x[3] for x in minibatch])).float().to(self.device)
        dones = torch.from_numpy(np.vstack([x[4] for x in minibatch])).float().to(self.device)

        # Optimisation du réseau de neurones
        loss = compute_loss(self.model, states, next_states, rewards, actions, dones, self.gamma, self.batch_size)
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

    # Réduction d'epsilon
    self.epsilon = max(self.epsilon_min, self.epsilon * self.epsilon_decay)

    def train(self, episodes: int):
        self.optimizer = optim.Adam(self.model.parameters(), lr=self.learning_rate)
        scores = []
        with tqdm.tqdm(range(episodes)) as t:
            for e in t:
                state, _ = self.env.reset()
                done = False
                score = 0
                while not done:
                    action = self.act(state)
                    next_state, reward, done, _, _ = self.env.step(action)
                    self.remember(state, action, reward, next_state, done)
                    state = next_state
                    score += reward
                    if len(self.memory) > self.batch_size:
                        self.replay()
                scores.append(score)
            t.set_postfix(reward=score)
            return scores
```

```
# Initialize environment
env = gym.make("MountainCar-v0")
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# Initialize agent
agent = DQNAgent(env, device)

# Train agent
scores = agent.train(episodes=1000)

# Plot scores
plt.plot(scores)
plt.title("Courbe d'évolution de la récompense")
```

```
25% |██████████| 247/1000 [2:12:00<6:42:27, 32.67s/it, reward=-3528.0]
KeyboardInterrupt Traceback (most recent call last)
Cell In [18], line 111
    108 agent = DQNAgent(env, device)
    109 # Train agent
--> 110 scores = agent.train(episodes=1000)
    113 # Plot scores
    114 plt.plot(scores)

Cell In [18], line 97, in DQNAgent.train(self, episodes)
    95 score += reward
    96 if len(self.memory) > self.batch_size:
--> 97     self.replay()
    98 scores.append(score)
    99 t.set_postfix(reward=score)

Cell In [18], line 73, in DQNAgent.act(self)
    69 dones = torch.from_numpy(np.vstack([x[4] for x in minibatch])).float().to(self.device)
    70 # Optimisation du réseau de neurones
--> 71 loss = compute_loss(self.model, states, next_states, rewards, actions, dones, self.gamma, self.batch_size)
    72 self.optimizer.zero_grad()
    73 self.optimizer.step()
    75 loss.backward()
```

```
Cell In [18], line 26, in compute_loss(model, state, next_state, reward, action, done, gamma, batch_size)
    22 """
    23 Compute the DQN agent loss
    24 """
    25 q_values = model(state)
--> 26 q_next_state = model(next_state)
    27 q_values_target = reward + (1 - done) * gamma * q_next_state.max(dim=1)[0]
    28 loss = F.mse_loss(q_values[range(batch_size), action], q_values_target)
    29 return loss

File ~/RELE/.venv/lib/python3.8/site-packages/torch/nn/modules/module.py:1190, in Module._call_impl(self, *input, **kwargs)
    1186 # If we don't have any hooks, we want to skip the rest of the logic in
    1187 # this function, and just call forward.
    1188 if not (self._backward_hooks or self._forward_hooks or self._forward_pre_hooks or _global_backward_hook
s
    1189         or _global_forward_hooks or _global_forward_pre_hooks):
--> 1190     return forward_call(*input, **kwargs)
    1191 # Do not call functions when jit is used
    1192 full_backward_hooks, non_full_backward_hooks = [], []

Cell In [14], line 14, in DQN.forward(self, x)
    13 x = torch.relu(self.fc1(x))
--> 14 x = torch.relu(self.fc2(x))
    15 x = self.fc3(x)
    16 return x
```

```
File ~/RELE/.venv/lib/python3.8/site-packages/torch/nn/modules/module.py:1190, in Module._call_impl(self, *input, **kwargs)
    1186 # If we don't have any hooks, we want to skip the rest of the logic in
    1187 # this function, and just call forward.
    1188 if not (self._backward_hooks or self._forward_hooks or self._forward_pre_hooks or _global_backward_hook
s
    1189         or _global_forward_hooks or _global_forward_pre_hooks):
--> 1190     return forward_call(*input, **kwargs)
    1191 # Do not call functions when jit is used
    1192 full_backward_hooks, non_full_backward_hooks = [], []

File ~/RELE/.venv/lib/python3.8/site-packages/torch/nn/modules/linear.py:114, in Linear.forward(self, input)
--> 113 def forward(self, input: Tensor) -> Tensor:
    114     return F.linear(input, self.weight, self.bias)
```

Q7. Visualisez la vidéo d'un cas d'échec et d'un cas de réussite.

In []: `# Ajoutez votre code ici`

Le temps de calcul étant très long, nous n'avons pas pu obtenir de résultats concluants.

Double DQN

Le Double DQN est un algorithme d'apprentissage par renforcement qui combine le DQN avec une technique appelée Prise de décision double. Le Double DQN permet à l'agent d'explorer plus intelligemment des environnements complexes et imprévisibles. L'algorithme est conçu pour maximiser la valeur estimée sur la base des informations disponibles.

L'algorithme commence par initialiser des paramètres pour le modèle de réseau neuronal profond et deux cibles de réseaux distincts, X et Y. Les deux cibles sont des copies parallèles, mais le modèle X est mis à jour plus fréquemment que le modèle Y. Quand un agent prend une action, il utilise le réseau X pour calculer la valeur estimée et recueille le feedback après l'action. L'agent utilise ensuite le réseau Y pour déterminer quelle était l'action optimale à partir de cette valeur estimée.

Cette procédure est appelée *prise de décision double*. Elle réduit l'instabilité et les biais de l'exploration liés à la maximisation prématurée.

Q8. Ajoutez le Double DQN sur votre implémentation précédente.

```
In [ ]: # Ajoutez votre code ici
from collections import deque, namedtuple
import torch.nn.functional as F

class DDQNAgent:
    def __init__(self, env, device):
        self.env = env
        self.device = device
        self.state_size = env.observation_space.shape[0]
        self.action_size = env.action_space.n
        self.memory = t.Deque[t.Tuple[np.ndarray, int, float, np.ndarray, bool]] = deque(maxlen=2000)
        self.batch_size = 32
        self.gamma = 0.95 # discount rate
        self.epsilon_min = 1.0 # exploration rate
        self.epsilon_min = 0.01
        self.epsilon_decay = 0.995
        self.learning_rate = 0.001
        self.model = self._build_model()
        self.target_model = self._build_model()

    def _build_model(self):
        # Neural Net for Deep-Q learning Model
        model = DQN(self.state_size, self.action_size).to(self.device)
        return model

    def remember(self, state: np.ndarray, action: int, reward: float, next_state: np.ndarray, done: bool):
        self.memory.append((state, action, reward, next_state, done))

    def act(self, state: np.ndarray) -> int:
        # Exploration : choisir une action aléatoire avec une probabilité epsilon
        if np.random.rand() <= self.epsilon:
            return self.env.action_space.sample()
        else:
            q_values = self.model(torch.from_numpy(state).float()).to(self.device)
            return torch.argmax(q_values).item()

    def replay(self):
        # Random replay
        minibatch = random.sample(self.memory, self.batch_size)
        states = torch.from_numpy(np.vstack([x[0] for x in minibatch])).float().to(self.device)
        actions = torch.from_numpy(np.vstack([x[1] for x in minibatch])).long().to(self.device)
        rewards = torch.from_numpy(np.vstack([x[2] for x in minibatch])).float().to(self.device)
        next_states = torch.from_numpy(np.vstack([x[3] for x in minibatch])).float().to(self.device)
        dones = torch.from_numpy(np.vstack([x[4] for x in minibatch])).float().to(self.device)

        q_values = self.model(states)
        q_next_state = self.model(next_states)
        q_target_next_state = self.target_model(next_states)
        q_expected = q_values.gather(1, actions)
        q_max = q_target_next_state.gather(1, q_argmax.unsqueeze(1)).squeeze(1)

        # Calculer la cible
        q_target = rewards + (self.gamma * q_max * (1 - dones))

        # Optimisation du réseau de neurones
        loss = F.mse_loss(q_expected, q_target)
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

    # Réduction d'epsilon
    self.epsilon = max(self.epsilon_min, self.epsilon * self.epsilon_decay)

    def train(self, episodes):
        self.optimizer = optim.Adam(self.model.parameters(), lr=self.learning_rate)
        scores = []
        for e in tqdm.tqdm(range(episodes)):
            state, _ = self.env.reset()
            done = False
            score = 0
            while not done:
                action = self.act(state)
                next_state, reward, done, _, _ = self.env.step(action)
                self.remember(state, action, reward, next_state, done)
                state = next_state
                score += reward
                if len(self.memory) > self.batch_size:
                    self.replay()
            scores.append(score)
            if e % 10 == 0:
                self.target_model.load_state_dict(self.model.state_dict())
            return scores

# Initialize environment
env = gym.make("MountainCar-v0")
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# Initialize agent
agent = DDQNAgent(env, device)

# Train agent
scores = agent.train(episodes=1000)

# Solution fin

# Plot scores
plt.plot(scores)
plt.title("Courbe d'évolution de la récompense")
```

Le temps de calcul étant très long, nous n'avons pas pu obtenir de résultats.

Q9. En cherchant sur Internet, proposez une série d'améliorations possibles et décrivez les

Il y a plusieurs façons d'améliorer un modèle de Double DQN:

- Utiliser des réseaux de neurones plus complexes, comme des réseaux de neurones à convolution (CNN) ou des réseaux de neurones récurrents (RNN). Ces modèles peuvent capturer des patterns spatiaux et temporels plus complexes, ce qui peut améliorer les performances du modèle.
- Utiliser un replay memory plus grand. Le replay memory est une partie cruciale du fonctionnement de l'algorithme de Double DQN. Plus le replay memory est grand, plus le modèle peut apprendre de l'expérience passée, ce qui peut améliorer les performances.
- Utiliser une politique d'exploration différente. La politique d'exploration détermine comment le modèle explore l'environnement et s'engage dans de nouvelles expériences. En utilisant une politique d'exploration qui encourage l'exploration de l'espace d'actions de manière plus systématique, il est possible d'améliorer les performances du modèle.
- Utiliser des fonctions de récompense modifiées pour guider l'apprentissage du modèle. En modifiant les fonctions de récompense utilisées par le modèle, il est possible de guider l'apprentissage de manière à ce qu'il se concentre sur les aspects de l'environnement qui sont les plus pertinents pour atteindre les objectifs du modèle.
- Entraîner le modèle pendant plus longtemps. Plus le modèle est entraîné, plus il a l'opportunité de s'adapter à l'environnement et d'améliorer ses performances. Cependant, il est important de veiller à ce que le modèle ne surajuste pas (overfitting) aux données d'entraînement et de tester le modèle sur des données de validation pour s'assurer qu'il généralise bien à de nouvelles situations.