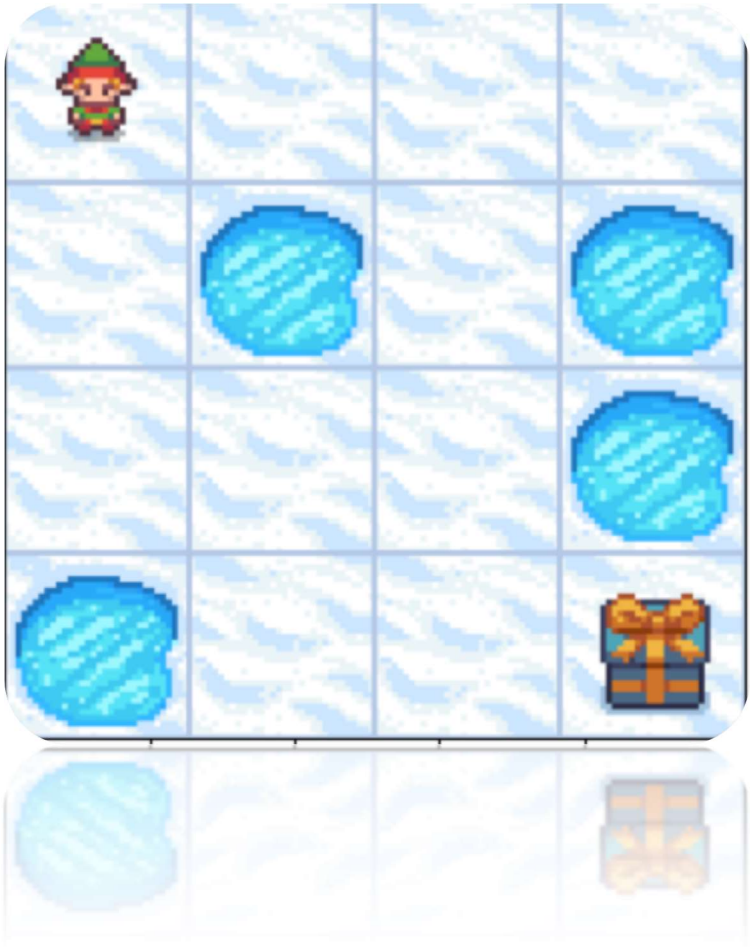
Rapport PIAI5 : FROZEN LAKE



boubacar balde Mathieu Sacewicz
13/12/2023

Table des matières

But du projet

L'agent doit atteindre trésor sans tomber dans un des trous.

Il sera à la position [0,0] du plateau et il peut se déplacer a gauche, en bas, à droite et en haut

- LEFT : 0
- DOWN : 1
- RIGHT : 2
- UP : 3

Exploration et exploitation

L'exploration et l'exploitation sont des concepts très important dans un apprentissage par renforcement car l'agent doit explorer et exploiter son environnement.

Le principe de l'exploration consiste à ce que l'agent fasse un déplacement totalement aléatoire et essayer d'avoir des récompenses en fonction de son déplacement afin de recueillir des nouvelles informations sur l'environnement.

Dans ce projet nous avons mis la récompense à 0 lorsqu'il tombe dans un trou (H) ou lorsqu'il se trouve dans une case glacée (F) et lorsqu'il atteint le trésor (G) aura une récompense de +1.

Le principe de l'exploitation consiste a ce que l'agent utilise les connaissances qu'il a déjà acquis grâce à l'exploration pour prendre des décisions qui maximisent la récompense immédiate

Principe du Qlearning

Pour l'apprentissage par renforcement nous utilisons l'algorithme de Qlearning celle-ci va nous permettre de chercher a trouver la meilleure action en tenant compte de l'état actuel et elle va utiliser une politique qui maximise la récompense total

$$Q(s,a) := Q(s,a) + LR[r+Y\max_{a'}*Q(s',a')-Q(s,a)]$$

s : état

a : action

LR : Learning rate

R : recompense

max $_{a'}$: maximisation de la policy

s' = état suivant

a' = action suivante

Y = gamme

Mise en pratique

Importation :

```
import gymnasium as gym
import numpy as np
import matplotlib.pyplot as plt
import pickle
```

Fonction **run** :

Elle comprends 3 paramètres :

```
def run(episodes, nb, is_training=True,render=False):
```

- episode : nombre d'épisodes
- nb = taille du plateau
- is_training : un booléen qui va entrainer l'agent si True sinon il va enregistrer le model qui a été entrainé dans un fichier
- render : le rendu de la fonction si render = true alors on aura un rendu humain sinon rien du tout.

```
#ce bout de code va initialiser l'environnement du projet

env = gym.make('FrozenLake-v1', map_name=str(nb)+"x"+str(nb),
is_slippery=False, render_mode='human' if render else None)

#si is_training = true il va entrainer le model sinon il va ouvrir le
fichier et les lires les données du model entrainé

if(is_training):
    q = np.zeros((env.observation_space.n, env.action_space.n)) # init a 64
x 4 array
else:
    f = open('frozen_lake8x8.pkl', 'rb')
    q = pickle.load(f)
    f.close()
```

```
#Hyperparametre du model
learning_rate_a = 0.9 # alpha or learning rate
discount_factor_g = 0.9 # gamma or discount rate. Near 0: more
weight/reward placed on immediate state. Near 1: more on future state.
epsilon = 1         # 1 = 100% random actions
epsilon_decay_rate = 0.0001        # epsilon decay rate. 1/0.0001 = 10,000
```

```
rng = np.random.default_rng()    # random number generator
rewards_per_episode = np.zeros(episodes)
for i in range(episodes):
    state = env.reset()[0]  # states: 0 to 63, 0=top left corner,63=bottom
right corner

    terminated = False      # True when fall in hole or reached goal
    truncated = False       # True when actions > 200

    while(not terminated and not truncated):
#epsilon greedy algorithme le principe est de choisir un nombre aleatoire
et si elle est inferieur a epsilon il va faire une exploration sinon une
exploitation
        if is_training and rng.random() < epsilon:
            action = env.action_space.sample() #choisie une action
        else:
            action = np.argmax(q[state,:])

        new_state,reward,terminated,truncated,_ = env.step(action)

        if is_training:
# algorithme de Qlearning qui va permettre d'update les valeurs de la
qtable pour que l'agent puisse prendre des décisions dans un environnement
en prenant en compte les récompenses immédiates et les estimations de
valeurs futures.

            q[state,action] = q[state,action] + learning_rate_a * (
                reward + discount_factor_g * np.max(q[new state,:]) -
q[state,action]
            )

        state = new state
#diminue de l'epsilon par la epsilon_decay_rate
    epsilon = max(epsilon - epsilon_decay_rate, 0)

    if(epsilon==0):
        learning_rate_a = 0.0001

    if reward == 1:
        rewards_per_episode[i] = 1

env.close()
#ajoute un fichier pour voir les statistiques de recompenses par episode
sum_rewards = np.zeros(episodes)
for t in range(episodes):
    sum_rewards[t] = np.sum(rewards_per_episode[max(0, t-100):(t+1)])
plt.plot(sum_rewards)
plt.savefig('frozen_lake8x8.png')
#si is_training = true il va enregistrer le model entrainer dans un fichier
if is_training:
    f = open("frozen_lake8x8.pkl","wb")
    pickle.dump(q, f)
    f.close()
```

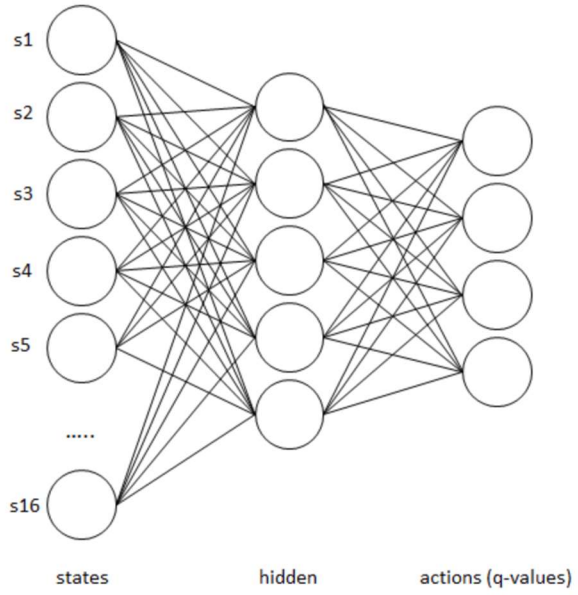
Nous allons parcourir le nombre d'episode

Principe du Deep Qlearning

Le but du Deep Qlearning se repose sur l'idée de fusionner le concept de l'apprentissage par renforcement avec l'utilisation de réseaux de neurone pour que l'IA apprenne à prendre des décisions dans un environnement complexe.

On utilise le Deep Qlearning lorsque le nombre d'état est grand car niveau complexité on ne peut pas la mettre dans une Qtable.

Le but est de faire une estimation de l'espérance sans la stocker dans un réseau de neurone pour se faire on aura un réseau de neurone qui va prendre en entrée l'état et qui va retourner 4 sorties et pour ces sorties elle va prendre la valeur de l'espérance du nombre de récompense qu'on peut espère avoir dans l'état s avec l'action a.



Mise en pratique

```
import gymnasium as gym
import numpy as np
import matplotlib.pyplot as plt
from collections import deque
import random
import torch
from torch import nn
import torch.nn.functional as F


#On definit une classe pour le model
class DQN(nn.Module):
    def __init__(self, in_states, h1_nodes, out_actions):
        super().__init__()

        # Define network layers
        self.fc1 = nn.Linear(in_states, h1_nodes)  # first fully connected
layer
        # sortie de la couche w h1 = weight
        self.out = nn.Linear(h1_nodes, out_actions)

    def forward(self, x):
        x = F.relu(self.fc1(x))  #applique la function d'activation relu
        x = self.out(x)  # Calcule la sortie
```

```
        return x


# Define memory for Experience Replay celle ci est important dans le dql car elle va contribuer a la stabilité de l'apprentissage et a ameliorer l'efficacité de l'apprentissage.
class ReplayMemory():
    def __init__(self, maxlen):
        self.memory = deque([], maxlen=maxlen)

    def append(self, transition):
        self.memory.append(transition)

    def sample(self, sample_size):
        return random.sample(self.memory, sample_size)

    def __len__(self):
        return len(self.memory)


# on definit une classe frozen lake DQL
class FrozenLakeDQL():
    # hyperparametre
    learning_rate_a = 0.001  # learning rate (alpha)
    discount_factor_g = 0.9  # discount rate (gamma)
    network_sync_rate = 10  # number of steps the agent takes before syncing the policy and target network
    replay_memory_size = 1000  # size of replay memory
    mini_batch_size = 32  # size of the training data set sampled from the replay memory

    # Neural Network
    loss_fn = nn.MSELoss()  # NN Loss function. MSE=Mean Squared Error can be swapped to something else.
    optimizer = None  # NN Optimizer. Initialize later.

    ACTIONS = ['L', 'D', 'R', 'U']  # for printing 0,1,2,3 => L(eft),D(own),R(ight),U(p)

    # Entraine environment frozen lake
    def train(self, episodes, render=False, is_slippery=False):
        # crée une instance de l'environnement
        env = gym.make('FrozenLake-v1', map_name="4x4", is_slippery=is_slippery,
                       render_mode='human' if render else None)
        num_states = env.observation_space.n
        num_actions = env.action_space.n

        epsilon = 1  # 1 = 100% random actions
        memory = ReplayMemory(self.replay_memory_size)

        # Create policy and target network. Number of nodes in the hidden layer can be adjusted.
        policy_dqn = DQN(in_states=num_states, h1_nodes=num_states, out_actions=num_actions)
        target_dqn = DQN(in_states=num_states, h1_nodes=num_states, out_actions=num_actions)

        # Make the target and policy networks the same (copy weights/biases from one network to the other)
        target_dqn.load_state_dict(policy_dqn.state_dict())
```

```
        print('Policy (random, before training):')
        self.print_dqn(policy_dqn)

        # Policy network optimizer. "Adam" optimizer can be swapped to
something else.
        self.optimizer = torch.optim.Adam(policy_dqn.parameters(),
lr=self.learning_rate_a)

        # List to keep track of rewards collected per episode. Initialize
list to 0's.
        rewards_per_episode = np.zeros(episodes)

        # List to keep track of epsilon decay
        epsilon_history = []

        # Track number of steps taken. Used for syncing policy => target
network.
        step_count = 0

        for i in range(episodes):
            state = env.reset()[0]  # Initialize to state 0
            terminated = False  # True when agent falls in hole or reached
goal
            truncated = False  # True when agent takes more than 200
actions

            # Agent navigates map until it falls into hole/reaches goal
(terminated), or has taken 200 actions (truncated).
            while (not terminated and not truncated):

                # Select action based on epsilon-greedy
                if random.random() < epsilon:
                    # select random action
                    action = env.action_space.sample()  # actions:
0=left,1=down,2=right,3=up
                else:
                    # select best action
                    with torch.no_grad():
                        action = policy_dqn(self.state_to_dqn_input(state,
num_states)).argmax().item()

                # Execute action
                new_state, reward, terminated, truncated, _ =
env.step(action)

                # Save experience into memory
                memory.append((state, action, new_state, reward,
terminated))

                # Move to the next state
                state = new_state

                # Increment step counter
                step_count += 1

            # Keep track of the rewards collected per episode.
            if reward == 1:
                rewards_per_episode[i] = 1

            # Check if enough experience has been collected and if at least
```

```
1 reward has been collected
            if len(memory) > self.mini_batch_size and
np.sum(rewards_per_episode) > 0:
                mini_batch = memory.sample(self.mini_batch_size)
                self.optimize(mini_batch, policy_dqn, target_dqn)

                # Decay epsilon
                epsilon = max(epsilon - 1 / episodes, 0)
                epsilon_history.append(epsilon)

                # Copy policy network to target network after a certain
number of steps
                if step_count > self.network_sync_rate:
                    target_dqn.load_state_dict(policy_dqn.state_dict())
                    step_count = 0

        # Close environment
        env.close()

        # Save policy
        torch.save(policy_dqn.state_dict(), "frozen_lake_dql.pt")

        # Create new graph
        plt.figure(1)

        # Plot average rewards (Y-axis) vs episodes (X-axis)
        sum_rewards = np.zeros(episodes)
        for x in range(episodes):
            sum_rewards[x] = np.sum(rewards_per_episode[max(0, x - 100):(x
+ 1)])
        plt.subplot(121)  # plot on a 1 row x 2 col grid, at cell 1
        plt.plot(sum_rewards)

        # Plot epsilon decay (Y-axis) vs episodes (X-axis)
        plt.subplot(122)  # plot on a 1 row x 2 col grid, at cell 2
        plt.plot(epsilon_history)

        # Save plots
        plt.savefig('frozen_lake_dql.png')

    # Optimize policy network
    def optimize(self, mini_batch, policy_dqn, target_dqn):

        # Get number of input nodes
        num_states = policy_dqn.fc1.in_features

        current_q_list = []
        target_q_list = []

        for state, action, new_state, reward, terminated in mini_batch:

            if terminated:
                # Agent either reached goal (reward=1) or fell into hole
(reward=0)
                # When in a terminated state, target q value should be set
to the reward.
                target = torch.FloatTensor([reward])
            else:
                # Calculate target q value
                with torch.no_grad():
                    target = torch.FloatTensor(
```

```
                        reward + self.discount_factor_g * target_dqn(
                            self.state_to_dqn_input(new_state,
num_states)).max()
                    )

            # Get the current set of Q values
            current_q = policy_dqn(self.state_to_dqn_input(state,
num_states))
            current_q_list.append(current_q)

            # Get the target set of Q values
            target_q = target_dqn(self.state_to_dqn_input(state,
num_states))
            # Adjust the specific action to the target that was just
calculated
            target_q[action] = target
            target_q_list.append(target_q)

        # Compute loss for the whole minibatch
        loss = self.loss_fn(torch.stack(current_q_list),
torch.stack(target_q_list))

        # Optimize the model
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

    '''
    Converts an state (int) to a tensor representation.
    For example, the FrozenLake 4x4 map has 4x4=16 states numbered from 0
to 15.

    Parameters: state=5, num_states=16
    Return: tensor([0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0.])
    '''

    def state_to_dqn_input(self, state: int, num_states: int) ->
torch.Tensor:
        input_tensor = torch.zeros(num_states)
        input_tensor[state] = 1
        return input_tensor

    # Methode qui va tester l'environnement entraine avec la bonnne policy
    def test(self, episodes, is_slippery=False):
        # Create FrozenLake instance
        env = gym.make('FrozenLake-v1', map_name="4x4",
is_slippery=is_slippery, render_mode='human')
        num_states = env.observation_space.n
        num_actions = env.action_space.n

        # Load learned policy
        policy_dqn = DQN(in_states=num_states, h1_nodes=num_states,
out_actions=num_actions)
        policy_dqn.load_state_dict(torch.load("frozen_lake_dql.pt"))
        policy_dqn.eval()  # switch model to evaluation mode

        print('Policy (trained):')
        self.print_dqn(policy_dqn)

        for i in range(episodes):
```

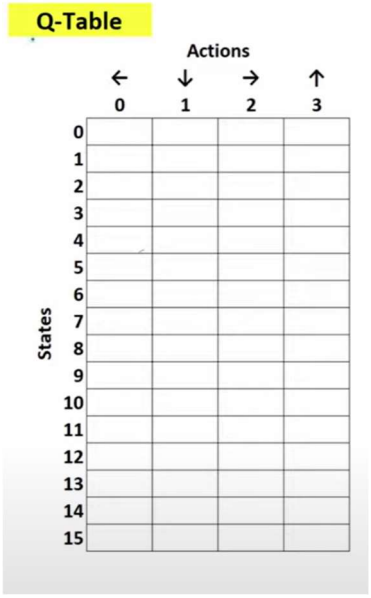
```
            state = env.reset()[0]  # Initialize to state 0
            terminated = False  # True when agent falls in hole or reached
goal
            truncated = False  # True when agent takes more than 200
actions

            # Agent navigates map until it falls into a hole (terminated),
reaches goal (terminated), or has taken 200 actions (truncated).
            while (not terminated and not truncated):
                # Select best action
                with torch.no_grad():
                    action = policy_dqn(self.state_to_dqn_input(state,
num_states)).argmax().item()

                # Execute action
                state, reward, terminated, truncated, _ = env.step(action)

        env.close()

    # Print DQN: state, best action, q values
    def print_dqn(self, dqn):
        # Get number of input nodes
        num_states = dqn.fc1.in_features

        # Loop each state and print policy to console
        for s in range(num_states):
            #  Format q values for printing
            q_values = ''
            for q in dqn(self.state_to_dqn_input(s, num_states)).tolist():
                q_values += "{:+.2f}".format(q) + ' '  # Concatenate q
values, format to 2 decimals
            q_values = q_values.rstrip()  # Remove space at the end

            # Map the best action to L D R U
            best_action = self.ACTIONS[dqn(self.state_to_dqn_input(s,
num_states)).argmax()]

            # Print policy in the format of: state, action, q values
            # The printed layout matches the FrozenLake map.
            print(f'{s:02},{best_action},[{q_values}]', end=' ')
            if (s + 1) % 4 == 0:
                print()  # Print a newline every 4 states


if __name__ == '__main__':
    frozen_lake = FrozenLakeDQL()
    is_slippery = True
    frozen_lake.train(3000, is_slippery=is_slippery)
    frozen_lake.test(10, is_slippery=is_slippery)
```

Qlearning vs Deep Qlearning

Qlearning	Deep Qlearning
- On utilise une Qtable ou chaque cellule de la table correspond a une paire état et action avec une valeur associée - Type d'espace : Etat discret - Exploration/exploitation : epsilon	- Q-Network : un réseau de neurone qui représente la fonction Q. - Type d'espace : Etat continue - Exploration/exploitation : apprends à explorer automatiquement - Mis a jour par lots après une série d'expérience

greedy

- Mis à jour après chaque etape de l'environnement



donc améliore la stabilité de l'apprentissage