MDS616 - Assessment 4 - Group Project - DQN algorithms for Cart Pole game

Author: Xiuwen ZHENG
Date: 12/05/2024
Course: MDS616 - Reinforcement Learning
Tutor: Anjin Liu

[3] 5秒
```
import gym
import os
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
#import torch.nn.functional as F
import matplotlib.pyplot as plt
#from IPython.display import clear_output
from IPython import display
import random
```

Part 1: Load the gymanium environment and check the data

[4] 0秒
```
# use gymnasium CartPole-v1 encironment, render mode is rgb_array
env = gym.make("CartPole-v1")
```

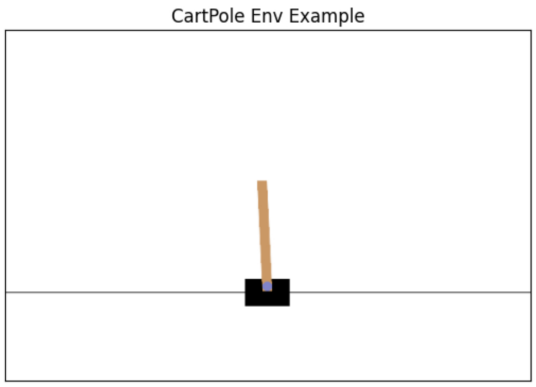
```
/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283: DeprecationWarning: `should_run_async` will not call `transform_cell` automatica
  and should_run_async(code)
/usr/local/lib/python3.10/dist-packages/gym/core.py:317: DeprecationWarning: WARN: Initializing wrapper in old step API which returns one bool inst
  deprecation(
/usr/local/lib/python3.10/dist-packages/gym/wrappers/step_api_compatibility.py:39: DeprecationWarning: WARN: Initializing environment in old step A
  deprecation(
```

[6] 0秒
```
# reset the environment
env.reset()
```

```
array([-0.00667643,  0.01785587, -0.01547993, -0.03446467], dtype=float32)
```

[7] 0秒
```
# convert env to a contiguous array and plot a picture
# please pip install gym<=0.26.0, I recommened 0.23.0
screen = env.render('rgb_array')
screen = np.ascontiguousarray(screen, dtype = np.float32) / 255
```

[ ]
```
plt.figure()
plt.imshow(screen)
plt.title("CartPole Env Example")
plt.xticks([])
plt.yticks([])
plt.show()
```



Part 2: Baseline Model and Deep Q learning

1. Baseline Model —— Random Policy

0秒
```
# define a funtion to plot the results
def plot_result(values, title=""):
        # draw a new picture when freshing
        plt.pause(0.02)
        display.clear_output(wait=True)
```

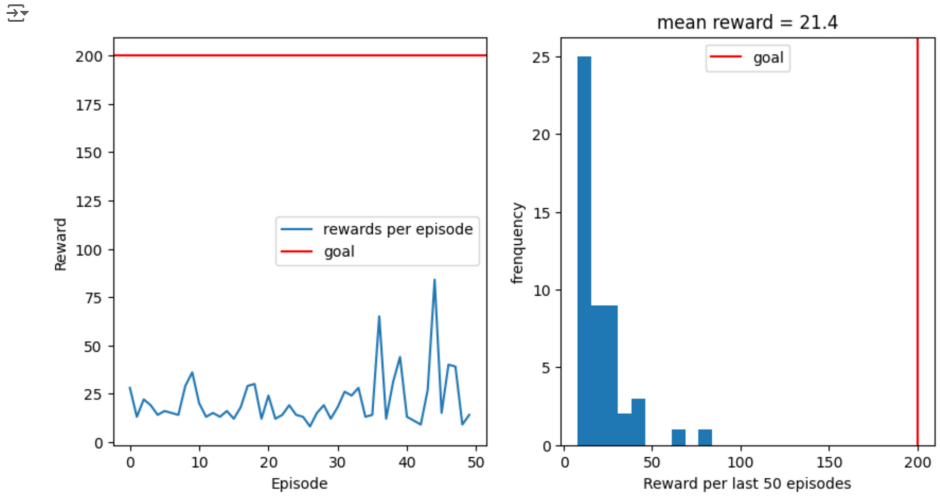
```
        fig, ax = plt.subplots(1,2, figsize=(10,5) ) # 2 subplots in one figure
        fig.suptitle(title)

        # plot 1 -- rewards per episode
        ax[0].plot(values, label="rewards per episode")
        ax[0].axhline(200, c='red', label='goal')
        ax[0].set_xlabel('Episode')
        ax[0].set_ylabel('Reward')
        ax[0].legend()

        # plot 2 -- mean reward of the lst 50 episodes
        ax[1].set_title("mean reward = {}".format(sum(values[-50:])/50))
        ax[1].hist(values[-50:])
        ax[1].axvline(200, c='red', label='goal')
        ax[1].set_xlabel('Reward per last 50 episodes')
        ax[1].set_ylabel('frenquency')
        ax[1].legend()

    plt.show()
```

0秒

```
[9] # random policy
    def random_policy(env, episodes):
        rewards = []
        for _ in range(episodes):
            env.reset()
            total_reward = 0
            done = False # whether the episodes ends
            while not done:
                action = env.action_space.sample() # randomly sample
                next_state, reward, done, info = env.step(action)
                total_reward += reward
            rewards.append(total_reward)
            plot_result(rewards)
        env.close()
```

30秒

```
[10] random_policy(env, 50)
```



2. Deep Q Learning

We defined a deep Q learning class which contains the structure of a neural network, a update function and a predict function.

0秒

```
[11] print(torch.__path__)
```

```
['/usr/local/lib/python3.10/dist-packages/torch']
```

We defined a simple DQN1 with 3 linear layers.

0秒

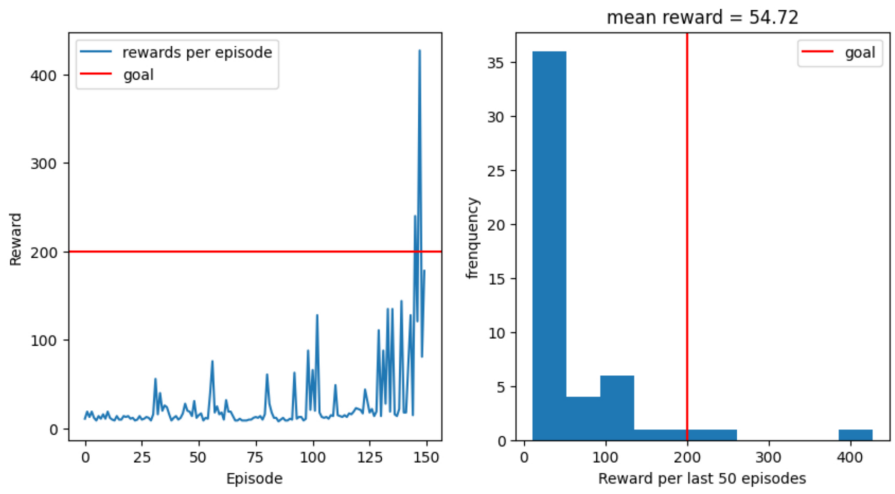
```
[12] class DQN1:
        def __init__(self, state_dim, action_dim, hidden_dim = 64, lr=0.001):
            """
            state_dim: input dimensions
            action_dim: output dimension(s)
            hidden_dim: set 64 hidden dimensions
            lr: learning rate
            """
            # define a Neural Network
            self.model = torch.nn.Sequential(
                torch.nn.Linear(state_dim, hidden_dim),
                torch.nn.ReLU(),
                torch.nn.Linear(hidden_dim,hidden_dim*2),
                torch.nn.ReLU(),
                torch.nn.Linear(hidden_dim*2, action_dim)
            )
            self.criterion = torch.nn.MSELoss() # use MSE as criterion
            self.optimizer = torch.optim.Adam(self.model.parameters(), lr=lr) # use Adam optimizer

        def update(self, state, q):
```

```
                q_pred = self.model(torch.Tensor(state)) # predict q value
                loss = self.criterion(q_pred, torch.Tensor(q)) # calculate loss function
                self.optimizer.zero_grad() # reset the gradient to 0
                loss.backward() # backward process
                self.optimizer.step() # update the optimizer

        def predict(self, state:np.ndarray) -> torch.Tensor:
                with torch.no_grad():
                        return self.model(torch.Tensor(state)) # prediction value should not be regarded as the "update value"
```

We defined a **DQN_process** function with the epsilon greedy policy.

```
[13] def DQN_process(env, model, episodes, gamma=0.9, epsilon=0.3, epsilon_decay=0.99, title="DQN_model"):
        """
        env: environment
        gamma: Markov procession gamma
        epsilon: we use epsilon greedy algorithm
        epsilon_decay: epsilon decay rate
        """
        rewards = []
        for _ in range(episodes):
                state = env.reset()
                total_reward = 0
                done = False
                while not done:
                        # epsilon greedy: if the random value less than epsilon, choose the random value, otherwise use the predi
                        if random.random() < epsilon:
                                action = env.action_space.sample()
                        else:
                                q_values = model.predict(state)
                                action = torch.argmax(q_values).item() # find the maximum item in q_values by their index

                        next_state, reward, done, info = env.step(action) # the four parameters by doing step"action"
                        total_reward += reward

                        q_values = model.predict(state)
                        # q value of the current step = former reward + gamma * the best prediction
                        # tell the agend the episode if it is end
                        q_values[action] = (reward + (0 if done else gamma * torch.max(model.predict(next_state)).item()) )
                        model.update(state, q_values) # update the model base on the new q value
                        state = next_state # jump to the next state

                epsilon = max(epsilon * epsilon_decay, 0.001) # epsilon decay until 0.001
                rewards.append(total_reward)
                plot_result(rewards)
```

```
[14] # result of DQN1
     dqn = DQN1(state_dim=4, action_dim=2, hidden_dim = 64, lr=0.001)
     DQN_process(env=env, model=dqn, episodes=150, gamma=0.9, epsilon=0.5)
```




```
[ ] 开始借助 AI 编写或生成代码。
```

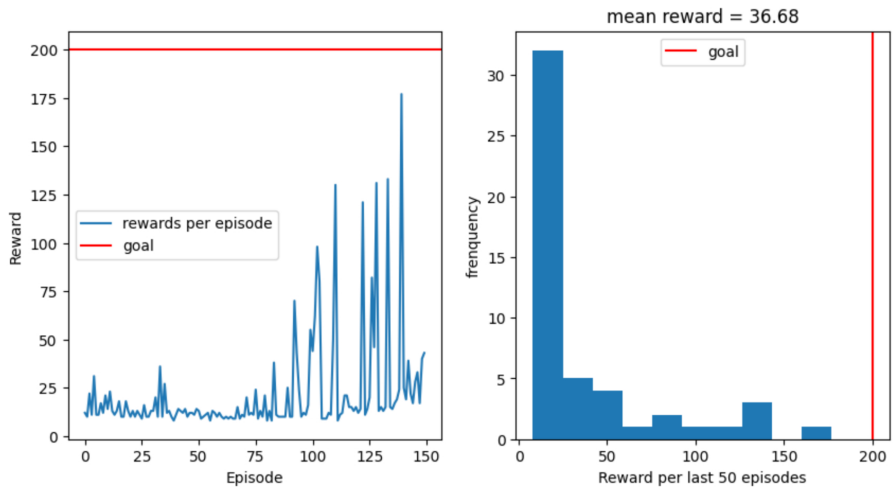
Secondly, We defined a DQN2 by adding two linear layer to see the difference comparing to DQN.

```
[16] import torch
     import numpy as np

     class DQN2:
             def __init__(self, state_dim, action_dim, hidden_dim=64, lr=0.001):
                     """
                     state_dim: input dimensions
                     action_dim: output dimension(s)
                     hidden_dim: set 64 hidden dimensions
                     lr: learning rate
                     """
                     # define a Neural Network with two additional hidden layers
```

```
                self.model = torch.nn.Sequential(
                        torch.nn.Linear(state_dim, hidden_dim),
                        torch.nn.ReLU(),
                        torch.nn.Linear(hidden_dim, hidden_dim * 2),
                        torch.nn.ReLU(),
                        torch.nn.Linear(hidden_dim * 2, hidden_dim * 2),    # additional layer
                        torch.nn.ReLU(),                                    # activation function for the additional layer
                        torch.nn.Linear(hidden_dim * 2, hidden_dim),        # another additional layer
                        torch.nn.ReLU(),                                    # activation function for the additional layer
                        torch.nn.Linear(hidden_dim, action_dim)
                )
                self.criterion = torch.nn.MSELoss()    # use MSE as criterion
                self.optimizer = torch.optim.Adam(self.model.parameters(), lr=lr)    # use Adam optimizer

        def update(self, state, q):
                q_pred = self.model(torch.Tensor(state))    # predict q value
                loss = self.criterion(q_pred, torch.Tensor(q))    # calculate loss function
                self.optimizer.zero_grad()    # reset the gradient to 0
                loss.backward()    # backward process
                self.optimizer.step()    # update the optimizer

        def predict(self, state: np.ndarray) -> torch.Tensor:
                with torch.no_grad():
                        return self.model(torch.Tensor(state))    # prediction value should not be regarded as the "update value"
```

2分钟

```
[17] # result of DQN2
dqn2 = DQN2(state_dim=4, action_dim=2, hidden_dim = 64, lr=0.001)
DQN_process(env=env, model=dqn2, episodes=150, gamma=0.9, epsilon=0.5)
```



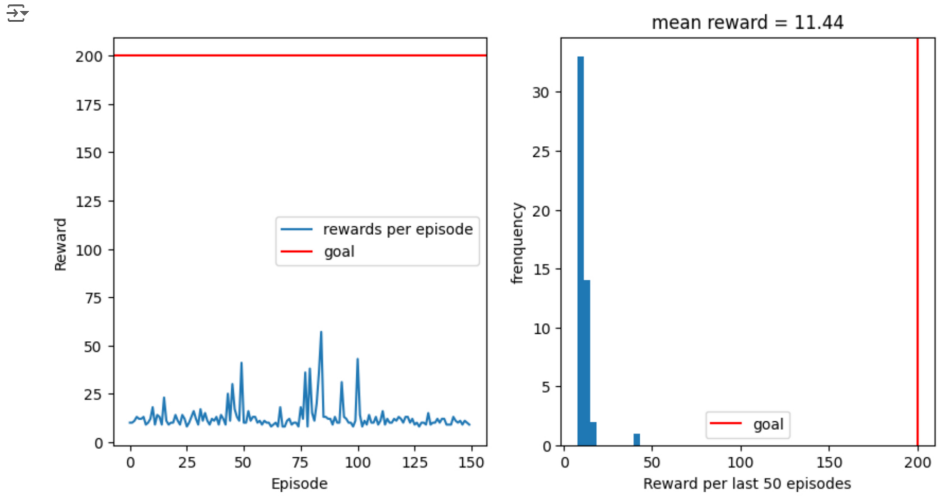
Interestingly, by adding two linear hidden layers, the mean reward dropped from 54.72 to 36.68, which seems not means that the more the hidden layers the better.

Then, in order to find the impact of activation function, we changed the ReLU to sigmoid function based on DQN1, and name it DQN1_sig.

0秒

```
[20] import torch
import numpy as np

class DQN1_sig:
        def __init__(self, state_dim, action_dim, hidden_dim=64, lr=0.001):
                """
                state_dim: input dimensions
                action_dim: output dimension(s)
                hidden_dim: set 64 hidden dimensions
                lr: learning rate
                """
                # define a Neural Network with two additional hidden layers using Sigmoid activation
                self.model = torch.nn.Sequential(
                        torch.nn.Linear(state_dim, hidden_dim),
                        torch.nn.Sigmoid(),
                        torch.nn.Linear(hidden_dim, hidden_dim * 2),
                        torch.nn.Sigmoid(),
                        torch.nn.Linear(hidden_dim*2, action_dim)
                )
                self.criterion = torch.nn.MSELoss()    # use MSE as criterion
                self.optimizer = torch.optim.Adam(self.model.parameters(), lr=lr)    # use Adam optimizer

        def update(self, state, q):
                q_pred = self.model(torch.Tensor(state))    # predict q value
                loss = self.criterion(q_pred, torch.Tensor(q))    # calculate loss function
                self.optimizer.zero_grad()    # reset the gradient to 0
                loss.backward()    # backward process
                self.optimizer.step()    # update the optimizer

        def predict(self, state: np.ndarray) -> torch.Tensor:
                with torch.no_grad():
                        return self.model(torch.Tensor(state))    # prediction value should not be regarded as the "update value"
```

```
[21] # result of DQN1_sig
     dqn1_sig = DQN1_sig(state_dim=4, action_dim=2, hidden_dim = 64, lr=0.001)
     DQN_process(env=env, model=dqn1_sig, episodes=150, gamma=0.9, epsilon=0.5)
```

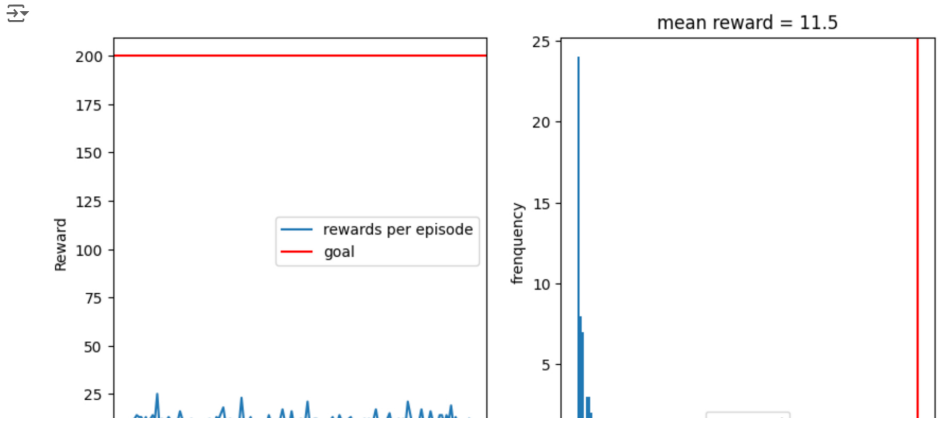


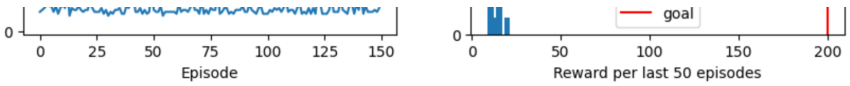

```
[ ] 开始借助 AI 编写或生成代码。
```

Finally, we changed the ReLU to sigmoid function based on DQN2, and name it DQN2_sig.

```
import torch
import numpy as np

class DQN2_sig:
    def __init__(self, state_dim, action_dim, hidden_dim=64, lr=0.001):
        """
        state_dim: input dimensions
        action_dim: output dimension(s)
        hidden_dim: set 64 hidden dimensions
        lr: learning rate
        """
        # define a Neural Network with two additional hidden layers using Sigmoid activation
        self.model = torch.nn.Sequential(
            torch.nn.Linear(state_dim, hidden_dim),
            torch.nn.Sigmoid(),
            torch.nn.Linear(hidden_dim, hidden_dim * 2),
            torch.nn.Sigmoid(),
            torch.nn.Linear(hidden_dim * 2, hidden_dim * 2),    # additional layer
            torch.nn.Sigmoid(),                                 # activation function for the additional layer
            torch.nn.Linear(hidden_dim * 2, hidden_dim),        # another additional layer
            torch.nn.Sigmoid(),                                 # activation function for the additional layer
            torch.nn.Linear(hidden_dim, action_dim)
        )
        self.criterion = torch.nn.MSELoss()    # use MSE as criterion
        self.optimizer = torch.optim.Adam(self.model.parameters(), lr=lr)    # use Adam optimizer

    def update(self, state, q):
        q_pred = self.model(torch.Tensor(state))    # predict q value
        loss = self.criterion(q_pred, torch.Tensor(q))    # calculate loss function
        self.optimizer.zero_grad()    # reset the gradient to 0
        loss.backward()    # backward process
        self.optimizer.step()    # update the optimizer

    def predict(self, state: np.ndarray) -> torch.Tensor:
        with torch.no_grad():
            return self.model(torch.Tensor(state))    # prediction value should not be regarded as the "update value"
```

```
# result of DQN2_sig
dqn2_sig = DQN2_sig(state_dim=4, action_dim=2, hidden_dim = 64, lr=0.001)
DQN_process(env=env, model=dqn2_sig, episodes=150, gamma=0.9, epsilon=0.5)
```





From the 4 models, it shows that DQN2 has the best reward per last 50 episods, and its' reward rises sharply as the episode increasing after 100.