

File Edit View Run Kernel Tabs Settings Help

616assessment4\_xiwen\_z X MDS607\_assgn1.ipynb +

Notebook Python 3 (ipykernel)

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

**Author:** Xiuwen ZHENG  
**Group members:** Xiuwen ZHENG, Linman ZHANG, Liang LIU, En SHI Date: 12/05/2024  
**Course:** MDS616 - Reinforcement Learning  
**Tutor:** Anjin Liu

```
[1]: 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 gymnasium environment and check the data

```
[2]: # use gymnasium CartPole-v1 environment, render mode is rgb_array
env = gym.make("CartPole-v1")

[3]: # reset the environment
env.reset()

[3]: array([-0.03905344, -0.03762776, -0.02049117, -0.01770797], dtype=float32)

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

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

**CartPole Env Example**

Part 2: Baseline Model and Deep Q learning

1. Baseline Model —— Random Policy

```
[6]: # define a function 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)

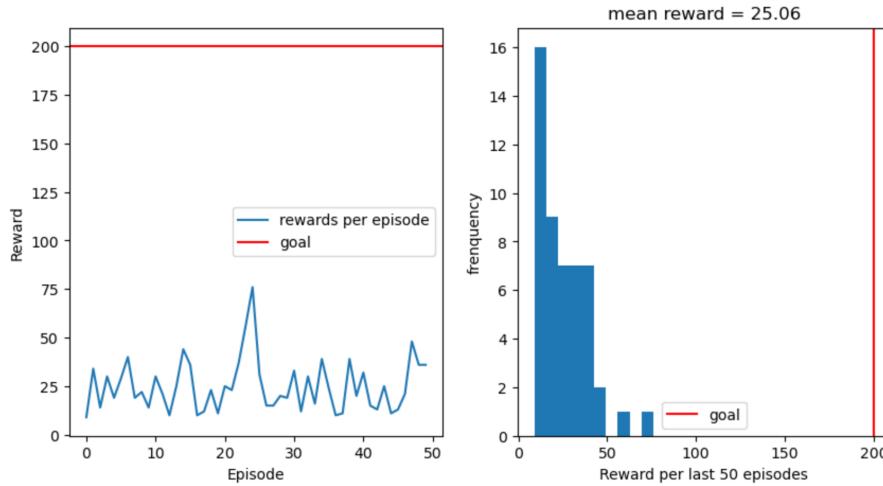
    # plot 1 -- rewards per episode
    ax[0].plot(values, label="rewards per episode")
    ax[0].axhline(200, c='red', label='goal')
    ax[0].set_xlabel('Episode')
    ax[0].set_ylabel('Reward')
    ax[0].legend()

    # plot 2 -- mean reward of the last 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') # we set a goal of 200 reward, but it is just a recommendation, you can try 150 or 250
    ax[1].set_xlabel('Reward per last 50 episodes')
    ax[1].set_ylabel('frequency')
    ax[1].legend()

    plt.show()
```

```
[8]: # 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()
```

```
[9]: # firstly, run 50 episodes to see the performance
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.

```
[10]: print(torch.__path__)
['D:\\anaconda3\\Lib\\site-packages\\torch']
```

We defined a simple DQN1 with 3 linear layers.

```
[11]: 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.

```
[12]: 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 predict value
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

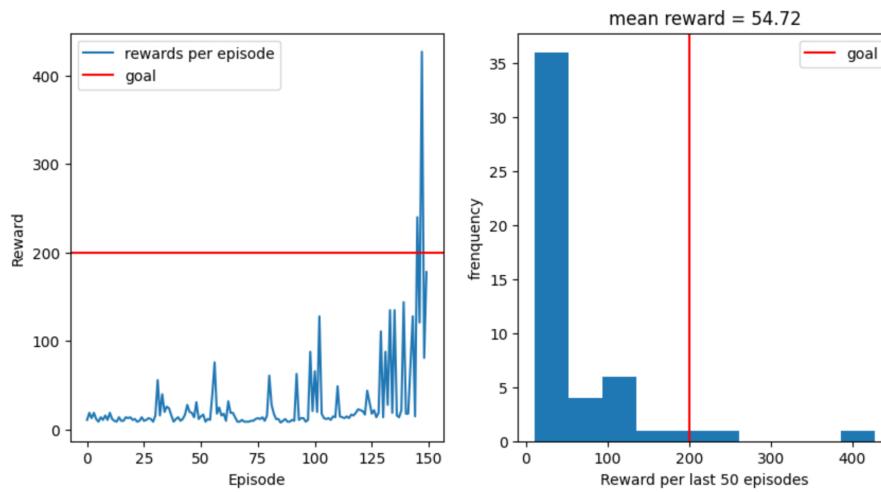
q_values = model.predict(state)
# q value of the current step = former reward + gamma * the best prediction
# tell the agent the episode if it is end
q_values[action] = (reward + (0 if done else gamma * torch.max(model.predict(next_state)).item()))
model.update(state, q_values) # update the model base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

```

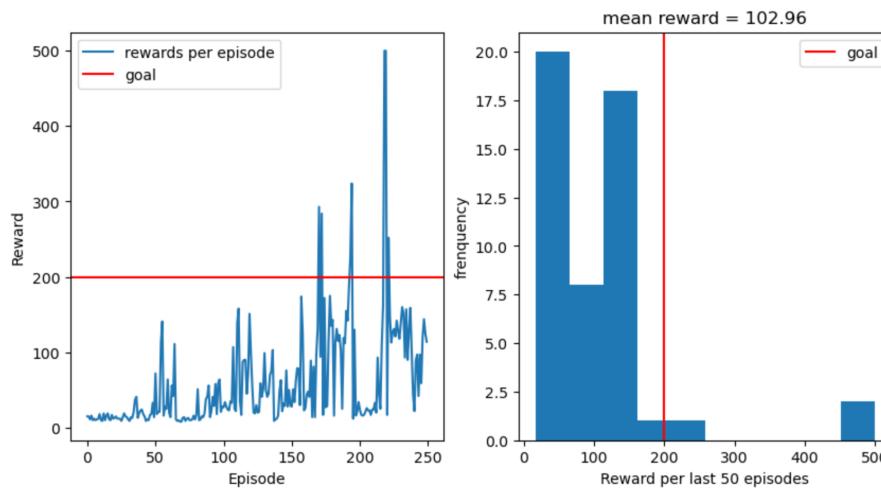
### 1. result of DQN1 - for 150 episodes

```
[14]: # result of DQN1 - for 150 episodes
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)
```



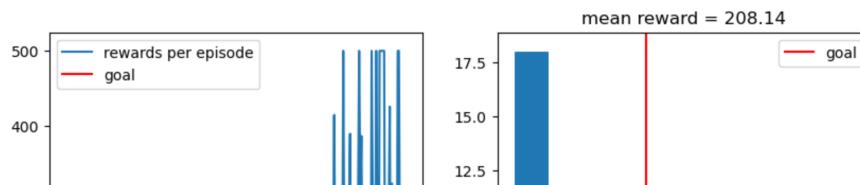
### 2. result of DQN1 - for 250 episodes

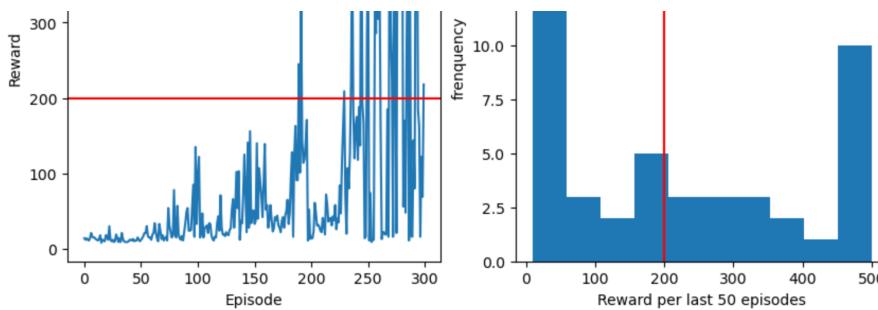
```
[22]: # result of DQN1 - for 250 episodes
dqn = DQN1(state_dim=4, action_dim=2, hidden_dim = 64, lr=0.001)
DQN_process(env=env, model=dqn, episodes=250, gamma=0.9, epsilon=0.5)
```



### 3. result of DQN1 - for 300 episodes

```
[19]: # result of DQN1 - for 300 episodes
dqn = DQN1(state_dim=4, action_dim=2, hidden_dim = 64, lr=0.001)
DQN_process(env=env, model=dqn, episodes=300, gamma=0.9, epsilon=0.5)
```





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

```
[14]: 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
        """

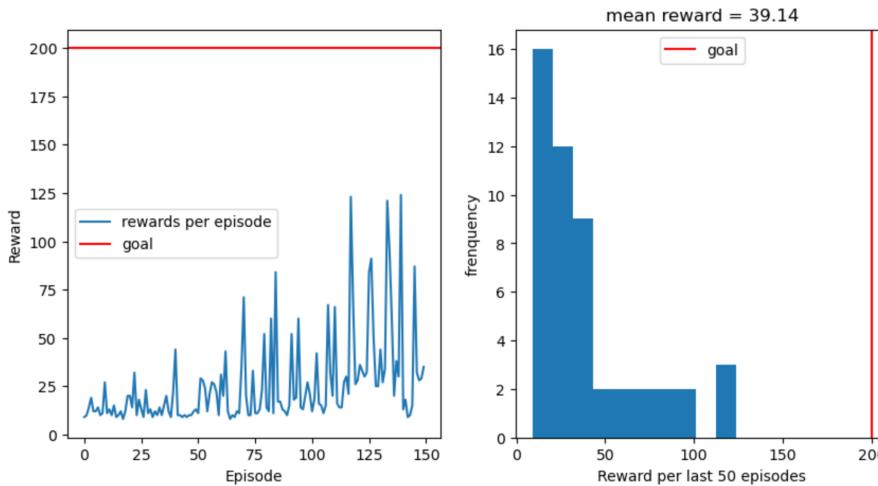
        # define a Neural Network with two additional hidden Layers
        self.model = torch.nn.Sequential(
            torch.nn.Linear(state_dim, hidden_dim),
            torch.nn.ReLU(),
            torch.nn.Linear(hidden_dim, hidden_dim * 2),
            torch.nn.ReLU(),
            torch.nn.Linear(hidden_dim * 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"
```

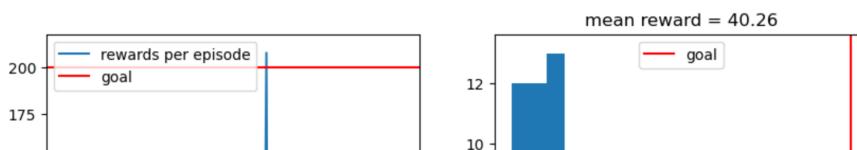
### 1. result of DQN2 - for 150 episodes

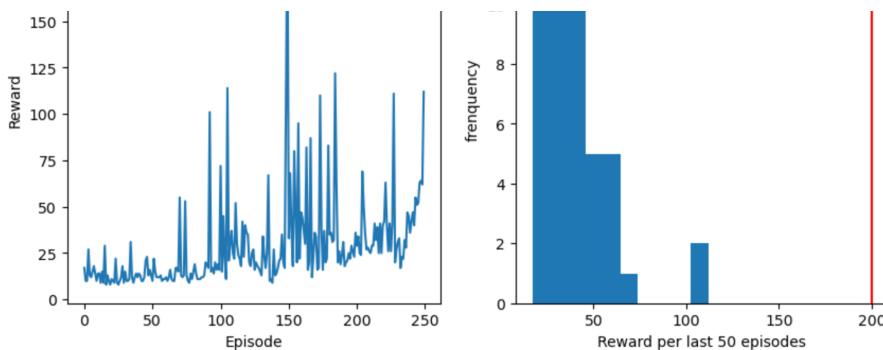
```
[15]: # result of DQN2
dqn2 = DQN2(state_dim=4, action_dim=2, hidden_dim = 64, lr=0.001)
DQN_process(env=env, model=dqn2, episodes=150, gamma=0.9, epsilon=0.5)
```



### 2. result of DQN2 - for 200 episodes

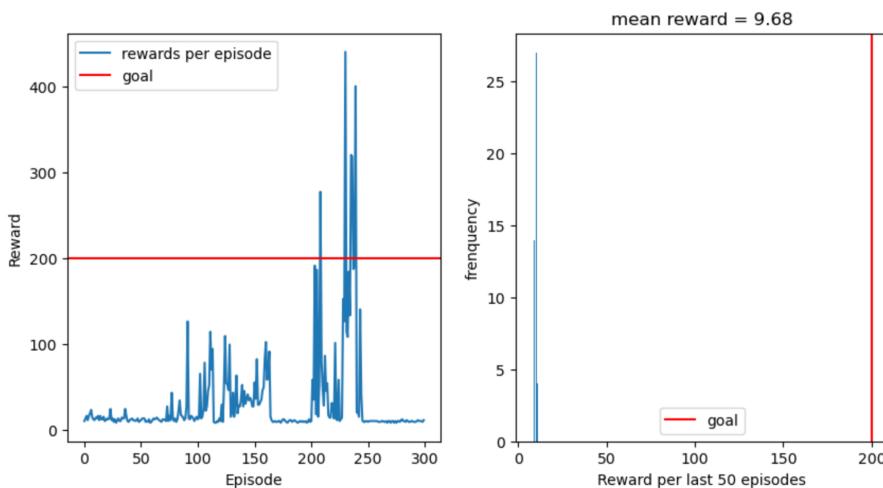
```
[21]: # result of DQN2 - for 250 episodes
dqn2 = DQN2(state_dim=4, action_dim=2, hidden_dim = 64, lr=0.001)
DQN_process(env=env, model=dqn2, episodes=250, gamma=0.9, epsilon=0.5)
```





### 3. result of DQN2 - for 300 episodes

```
[20]: # result of DQN2 - for 300 episodes
dqn2 = DQN2(state_dim=4, action_dim=2, hidden_dim = 64, lr=0.001)
DQN_process(env=env, model=dqn2, episodes=300, 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.

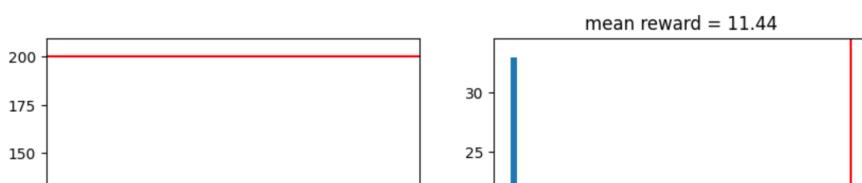
```
[20]: import torch
import numpy as np

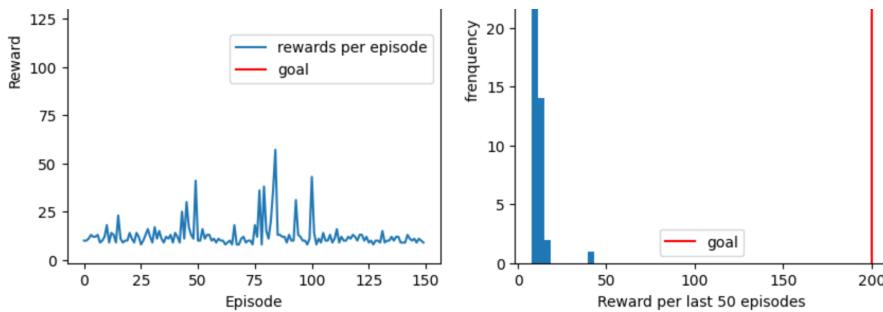
class DQN1_sig:
    def __init__(self, state_dim, action_dim, hidden_dim=64, lr=0.001):
        """
        state_dim: input dimensions
        action_dim: output dimension(s)
        hidden_dim: set 64 hidden dimensions
        lr: learning rate
        """
        # define a Neural Network with two 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)
```





[ ]:

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

```
[24]: import torch
import numpy as np

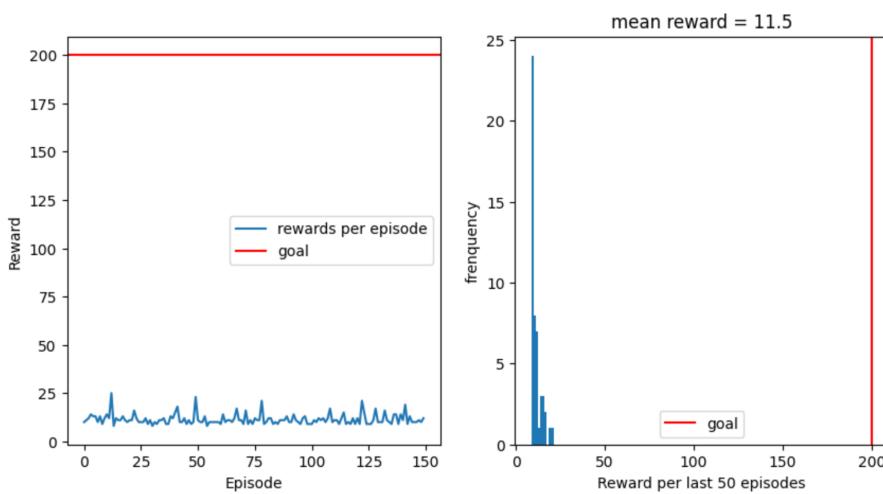
class DQN2_sig:
    def __init__(self, state_dim, action_dim, hidden_dim=64, lr=0.001):
        """
        state_dim: input dimensions
        action_dim: output dimension(s)
        hidden_dim: set 64 hidden dimensions
        lr: learning rate
        """

        # 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"
```

```
[25]: # 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.

Simple 0 10 Python 3 (ipykernel) | Idle

Mode: Command Ln 3, Col 26 616assessment4\_xiuwen\_zheng.ipynb 1