

▼ Lab 13: Reinforcement Learning (RL)

I bring lab 13 and 14 of RL in RTML class (Up to DQN, remove self-environment creation). Please design exercise at below :-)

Today we'll have a gentle introduction to reinforcement learning. The material in today's lab comes from these references:

- Pytorch 1.x Reinforcement Learning Cookbook (Packtpub)
- Hands-On Reinforcement Learning for Games (Packtpub)
- <https://www.kdnuggets.com/2018/03/5-things-reinforcement-learning.html>
- Reinforcement Learning: An Introduction (Sutton et al.)
- <https://github.com/werner-duvaud/muzero-general> (simulator code)

Reinforcement learning

Reinforcement Learning (RL) is a machine learning technique that enables an agent to learn in an interactive environment by trial and error using feedback on its actions and experiences. RL uses rewards and punishment as signals for "good" and "bad" behavior.

Generally, at each step, the agent outputs an action, which is input to the environment. The environment evolves according to its dynamics, the agent observes the new state of the environment and (optionally) a reward, and the process continues until hopefully the agent learns what behavior maximizes its reward.

 Introduction

Markov decision process (MDP)

Markov decision process (MDP) is a discrete-time stochastic control process. It provides a mathematical framework for modeling decision making in situations where outcomes are partly random and partly under the control of a decision maker. MDPs are useful for studying optimization problems solved via dynamic programming. They are used in many disciplines, including robotics, automatic control, economics and manufacturing.

The MDP model is based on the idea of an environment that evolves as a Markov chain.

Markov chain

A Markov chain is a model of the dynamics of a discrete time system that obeys the (first order) "Markov property," meaning that the state s^{t+1} at time $t + 1$ is conditionally independent of the state at times $0, \dots, t - 1$ given the state at time t , i.e.,

$$p(s^{t+1} \mid s^t, s^{t-1}, \dots, s^0) = p(s^{t+1} \mid s^t).$$

Informally, we might say that the current state is all you need to know to predict the next state.

A Markov chain is defined by a set of possible states $S = s_0, s_1, \dots, s_n$ and a transition matrix $T(s, s')$ containing the probabilities of state s (current state) transitioning to state s' (next state).

Here is a visualization of a simple Markov chain:



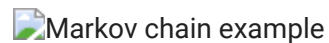
Markov chain

You might be interested in [this Markov chain simulator](#) and the [full screen diagram](#).

Now, the dynamics of the environment in a MDP are slightly different from that of a simple Markov chain. We have to consider how the agent's actions affect the system's dynamics. At each time step, rather than just transitioning randomly to the next state, we add the agent's action as an external input or disturbance $a \in A$, so (assuming a small number of discrete states and actions) the transition probabilities become a 3D tensor of size $|S| \times |A| \times |S|$ mapping each state/action pair to a probability distribution over the states.

A simple MDP

Assume we have **Three states (A, B, C)**, the Markov chain is:



Markov chain example

The transition matrix is:

$$T = \begin{bmatrix} 0.3 & 0.5 & 0.2 \\ 0.8 & 0.1 & 0.1 \\ 0.1 & 0.3 & 0.6 \end{bmatrix}$$

Each row means current state, and each column means the next state to go. If you consider 0.5 in the matrix (row 0, and column 1), you can say that "now you are in state **A** and the possible to go to next state of **B** is 0.5"

```
import torch

T = torch.tensor([[0.3, 0.5, 0.2],
                  [0.8, 0.1, 0.1],
                  [0.1, 0.3, 0.6]])

# show the matrix
print(T)
```

```
tensor([[0.3000, 0.5000, 0.2000],
        [0.8000, 0.1000, 0.1000],
        [0.1000, 0.3000, 0.6000]])
```

▼ Probability after k steps

The probability after k steps equation is:

$$T_n = T^n$$

Thus, if we want to find the T state at step 2, 5, 10, 20, we can do as:

```
T_2 = torch.matrix_power(T,2)
print('state transition at k=2:', T_2)
T_5 = torch.matrix_power(T,5)
print('state transition at k=5:', T_5)
T_10 = torch.matrix_power(T,10)
print('state transition at k=10:', T_10)
T_20 = torch.matrix_power(T,20)
print('state transition at k=20:', T_20)
```

```
state transition at k=2: tensor([[0.5100, 0.2600, 0.2300],
                                [0.3300, 0.4400, 0.2300],
                                [0.3300, 0.2600, 0.4100]])
state transition at k=5: tensor([[0.3991, 0.3230, 0.2779],
                                [0.4153, 0.3100, 0.2746],
                                [0.3926, 0.3165, 0.2908]])
state transition at k=10: tensor([[0.4026, 0.3170, 0.2804],
                                 [0.4024, 0.3172, 0.2804],
                                 [0.4024, 0.3170, 0.2806]])
state transition at k=20: tensor([[0.4024, 0.3171, 0.2805],
                                 [0.4024, 0.3171, 0.2805],
                                 [0.4024, 0.3171, 0.2805]])
```

The state transition after step 10 to 20 steps are going to converges. This means that, no matter what state the process is in, it has the same probability of transitioning to A (40.24%), B (31.71%), and C (28.05%).

▼ Initial Distribution

Assume we have the initial distribution of three state is:

$$V = \begin{bmatrix} 0.2 & 0.5 & 0.3 \end{bmatrix}$$

The initial distribution means you can go to any states at the first time, but it depends on the probability. In this matrix, we can say that there are possible to go to state A more than other states.

```
V_0 = torch.tensor([[0.2, 0.5, 0.3]])

print(V_0)

tensor([[0.2000, 0.5000, 0.3000]])
```

Try to calculate the state distribution after state 1, 2, 5, 10, and 20

```
V_1 = torch.mm(V_0, T)
print("Distribution of states after 1 step:\n{}\n".format(V_1))
V_2 = torch.mm(V_0, T_2)
print("Distribution of states after 2 step:\n{}\n".format(V_2))
V_5 = torch.mm(V_0, T_5)
print("Distribution of states after 5 step:\n{}\n".format(V_5))
V_10 = torch.mm(V_0, T_10)
print("Distribution of states after 10 step:\n{}\n".format(V_10))
V_20 = torch.mm(V_0, T_20)
print("Distribution of states after 20 step:\n{}\n".format(V_20))
```

```
Distribution of states after 1 step:
tensor([[0.4900, 0.2400, 0.2700]])
```

```
Distribution of states after 2 step:
tensor([[0.3660, 0.3500, 0.2840]])
```

```
Distribution of states after 5 step:
tensor([[0.4053, 0.3146, 0.2801]])
```

```
Distribution of states after 10 step:
tensor([[0.4024, 0.3171, 0.2805]])
```

```
Distribution of states after 20 step:
tensor([[0.4024, 0.3171, 0.2805]])
```

We can see that, after 10 steps, the state distribution converges. The probability of being in A (40.24%), B (31.71%) and the probability of being in s1 (28.05%) remain unchanged in the long run.

Starting with $[0.2, 0.5, 0.3]$, the state distribution after one iteration becomes $[0.4024, 0.3171, 0.2805]$. Details of its calculation are illustrated in the following diagram:



Come back to MDP

MDP is slightly different from a simple Markov chain because MDP has to consider agent's actions which affect to the system's dynamics. Thus, not only probability of transition of nextstate, but also agent's action need to be calculate. We assign \mathcal{A} as all possible actions and a as the action which agent selects (at this situation, a must be in \mathcal{A} or $a \in \mathcal{A}$).

The transition probabilities become a **3D tensor** of size $|\mathcal{S}| \times |\mathcal{A}| \times |\mathcal{S}|$ mapping each state/action pair to a probability distribution over the states.

▼ A simple MDP

Suppose we have **three states (s0, s1, s2) and two actions (a0, a1)** and that the state/action transition tensor is as follows:

$$T = \left\{ \begin{array}{l} \begin{bmatrix} 0.8 & 0.1 & 0.1 \end{bmatrix} \\ \begin{bmatrix} 0.1 & 0.6 & 0.3 \end{bmatrix} \\ \begin{bmatrix} 0.7 & 0.2 & 0.1 \end{bmatrix} \\ \begin{bmatrix} 0.1 & 0.8 & 0.1 \end{bmatrix} \\ \begin{bmatrix} 0.6 & 0.2 & 0.2 \end{bmatrix} \\ \begin{bmatrix} 0.1 & 0.4 & 0.5 \end{bmatrix} \end{array} \right.$$

The first matrix block is mean current state $s0$, second block is $s1$, and the third block is $s2$.

Consider into the matrix block, the rows of matrix block mean actions (a0 is 1st row, and a1 is 2nd row). And the columns of matix block is the same as Markov Chain: next state s'

```
# State transition function
T = torch.tensor([[[0.8, 0.1, 0.1],
                  [0.1, 0.6, 0.3]],
                 [[0.7, 0.2, 0.1],
                  [0.1, 0.8, 0.1]],
                 [[0.6, 0.2, 0.2],
                  [0.1, 0.4, 0.5]]])
```

```
[0.1, 0.4, 0.5]]])
```

```
# show the matrix  
print(T)
```

```
tensor([[[[0.8000, 0.1000, 0.1000],  
          [0.1000, 0.6000, 0.3000]],  
  
        [[0.7000, 0.2000, 0.1000],  
          [0.1000, 0.8000, 0.1000]],  
  
        [[0.6000, 0.2000, 0.2000],  
          [0.1000, 0.4000, 0.5000]]]])
```

Assume we want to observe at state 2, and see the probability to change to state 1 with playing action 0, we can write in code with

```
T[2,0,1]
```

```
tensor(0.2000)
```

To complete our simple MDP, we need a *reward function* R and a *discount factor* γ .

reward function R is a set of rewards that depend on the state and the action taken.

discount factor γ is how important future rewards are to the current state. Discount factor is a value between 0 and 1. A reward R that occurs n steps in the future from the current state, is multiplied by γ^n to describe its importance to the current state (Thus, current reward is $\gamma^t R$, where t is a number step.

Suppose $R = [-1, 0.1, 0.9]$ and $\gamma = 0.5$. Let's define our MDP in Python with PyTorch tensors:

```
# Reward function  
R = torch.tensor([-1.,0.1,0.9])  
  
# Discount factor  
gamma = 0.5
```

► The agent's goal

Once the MDP is defined, the agent's goal is to **maximize its expected reward**.

If we start in state s^0 and perform a series of actions a^0, a^1, \dots, a^{T-1} placing us in state s^1, s^2, \dots, s^T , we obtain the total reward R_F

$$R_F = \sum_{t=0}^T \gamma^t R(s^t)$$

The agent's goal is to behave so as to maximize the expected total reward. To do so, it should come up with a policy $\pi : S \times A \rightarrow \mathbb{R}$ giving a probability distribution over actions that can be executed in each state, then when in state s , sample action a according to that distribution $\pi(s, \cdot)$, and repeat.

Now the agent's goal can be clearly specified as finding an optimal policy

$$\pi^* = \operatorname{argmax}_{\pi} \mathbb{E}_{a^t \sim \pi(s^t), s^t \sim T(s^{t-1}, a^{t-1})} \left[\sum_{t=0}^T \gamma^t R(s^t) \right]$$

Under a particular policy π , then, the *value* of state s is the expected reward we obtain by following π from state s :

$$V^{\pi}(s) = \mathbb{E}_{a^t \sim \pi(s^t), s^t \sim T(s^{t-1}, a^{t-1}) | s^0 = s} \left[R(s) + \sum_{t=1}^T \gamma^t R(s^t) \right]$$

The value function clearly obeys the *Bellman equations*

$$V^{\pi}(s) = R(s) + \gamma \sum_{s', a'} \pi(s, a') T(s, a', s') V^{\pi}(s').$$

Too difficult?

OK, lets see the easier version of equation

The equation of V relate at time t is:

$$V_{t+1} = R + \gamma * T * V_t$$

When the value converges, which mean $V_{t+1} = V_t$, so we can derive the value V as:

$$\begin{aligned} V &= R + \gamma * T * V \\ V &= (I - \gamma * T)^{-1} * R. \end{aligned}$$

The function can be writed as:

[] ↳ 7 cells hidden

► Policy evaluation

To determine how good a particular policy is, we use policy evaluation. Policy evaluation is an iterative algorithm. It starts with arbitrary values for each state and then iteratively updates the values based on the Bellman equations until the values converge. Assume π is the value of a policy, the update equation is:

$$V(s) = \sum_a \pi(s, a) \left[R(s, a) + \gamma \sum_{s'} T(s, a', s') V(s') \right]$$

$\pi(s, a)$: the probability of taking action a in stat s under policy π $R(s, a)$: the reward received in state s by taking action a

You can see this algorithm's pseudocode in Sutton's book on page 75.

Because Policy evaluation is a loop processing, you need to stop by conditions. There are 2 ways to terminate the iterative updating process.

- Fix number of iterations
- Set some specific threshold and terminating the process when the values of all states change to lower than the threshold.

Here we compute the value of the three states in our MDP assuming the agent always peforms the first action. The function process are:

1. Initializes the policy values as all zeros.
2. Updates the values based on the Bellman expectation equation.
3. Computes the maximal change of the values across all states.
4. If the maximal change is greater than the threshold, it keeps updating the values. Otherwise, it terminates the evaluation process and returns the latest values.

[] ↳ 8 cells hidden

Value iteration algorithm

The idea behind value iteration is quite similar to that of policy evaluation. It is also an iterative algorithm. It starts with arbitrary policy values and then iteratively updates the values based on the Bellman optimality equation until they converge. So in each iteration, instead of taking the expectation (average) of values across all actions, it picks the action that achieves the maximal policy values:

$$V^*(s) = \max_a \left[R(s, a) + \gamma \sum_{s'} T(s, a', s') V^*(s') \right]$$

$V^*(s)$ denotes the optimal value, which is the value of the optimal policy

Once the optimal values are computed, we obtain the optimal policy:

$$\pi^*(s) = \operatorname{argmax}_a \sum_{s'} T(s, a, s') [R(s, a) + \gamma V^*(s')]$$

Apply it to gym environment


In this step, we try to use *FrozenLake* environment to see how to implement the value iteration.

FrozenLake is a typical Gym environment with a discrete state space. It is about moving an agent from the starting location to the goal location in a grid world, and at the same time avoiding traps. Currently, We use four by four grid (<https://gym.openai.com/envs/FrozenLake-v0/>).

► OpenAI Gym

One of the popular simulation environment for RL is OpenAI Gym.

[OpenAI](#) is a research company trying to develop systems exhibiting *artificial general intelligence* (AGI). They developed Gym to support the development of RL algorithms. Gym provides many reinforcement learning simulations and tasks. Visit [the Gym website](#) for a full list of environments.

 Gym example

Install Gym

From your local machine, you can follow from [this step](#). However, it is a little bit old (and the gym is update every 2 month!), so try to solve those problems yourself. :-)

```
[ ] ↳ 62 cells hidden
```

Using the environments to do reinforcement learning

Monte Carlo (MC) Method

Monte Carlo method is a model-free which have no require any prior knowledge of the environment. MC method is more scalable than MDP. MC control is used for finding the optimal policy when a policy is not given. There are 2 basically of MC control: on-policy and off-policy. On-policy

method learns about the optimal policy by executing the policy and evaluating and improving it, while Off-policy method learns about the optimal policy using data generated by another policy.

▼ Monte Carlo Method Concept

MC method is any method that uses randomness to solve problems. The algorithm repeats suitable **random sampling** and observes the fraction of samples that obey particular properties in order to make numerical estimations.

Find π using MC method

π ? Yes the π which is used for calculate circle. We assume that if there are some random points which are in the square area. We can calculate π by

$$A_{rect} = (2r)^2 = 4r^2$$
$$A_{circle} = \pi r^2$$

Hence,

$$\frac{A_{circle}}{A_{rect}} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4}$$

Let's try to code:

```
import torch
import numpy
import math
import matplotlib.pyplot as plt
```

Random 2,000 points in the square $x \in [-1, 1]$ and $y \in [-1, 1]$

```
n_point = 2000
points = torch.rand((n_point, 2)) * 2 - 1
```

Find the points which are inside the circle, and keep them (this is for plot in the graph only)

```

points_circle_x = []
points_circle_y = []
n_point_circle = 0

pi_iteration = []
i = 1
for point in points:
    r = torch.sqrt(point[0] ** 2 + point[1] ** 2)
    if r <= 1:
        points_circle_x.append(point[0])
        points_circle_y.append(point[1])
        n_point_circle += 1
    pi_iteration.append(4 * (n_point_circle / i))
    i += 1

points_circle_x = numpy.asarray(points_circle_x)
points_circle_y = numpy.asarray(points_circle_y)

```

Find the estimate π

```

pi_estimated = 4 * n_point_circle / n_point
print('Estimated value of pi is:', pi_estimated)

```

```

Estimated value of pi is: 3.136

```

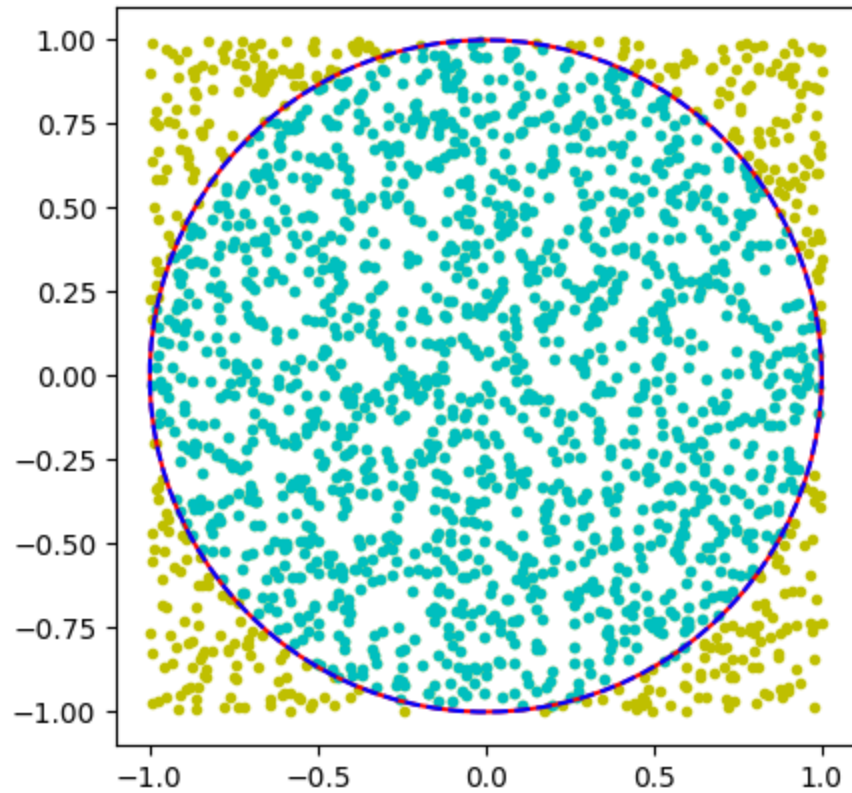
Plot to show the points and circle

```

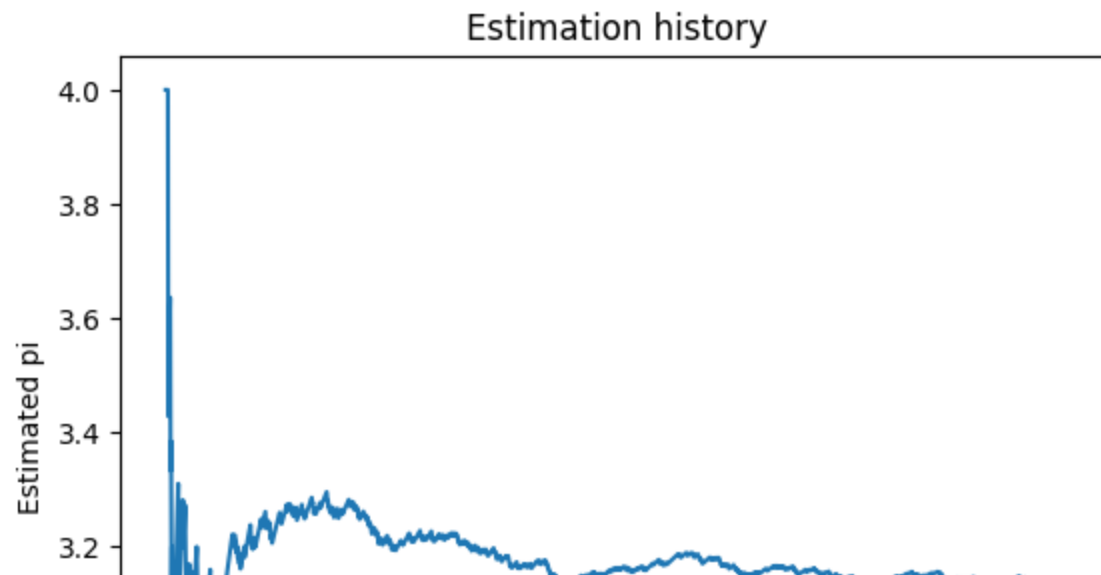
plt.plot(points[:, 0].numpy(), points[:, 1].numpy(), 'y.')
plt.plot(points_circle_x, points_circle_y, 'c.')
# plot real circle
i = torch.linspace(0, 2 * math.pi)
plt.plot(torch.cos(i).numpy(), torch.sin(i).numpy(), 'r-')
# plot circle from estimate pi
i = torch.linspace(0, 2 * pi_estimated)
plt.plot(torch.cos(i).numpy(), torch.sin(i).numpy(), 'b--')
plt.axes().set_aspect('equal')
plt.show()

```

```
/tmp/ipykernel_1037299/192663526.py:4: UserWarning: Not providing a value for linspace's steps is deprecated and will throw a run
  i = torch.linspace(0, 2 * math.pi)
/tmp/ipykernel_1037299/192663526.py:9: MatplotlibDeprecationWarning: Adding an axes using the same arguments as a previous axes c
plt.axes().set_aspect('equal')
```



```
plt.plot(pi_iteration)
plt.plot([math.pi] * n_point, '--')
plt.xlabel('Iteration')
plt.ylabel('Estimated pi')
plt.title('Estimation history')
plt.show()
```



Monte Carlo Policy Evaluation

Last time, we applied **DP** algorithm into policy evaluation. The things we need are:

- Full state-value transition matrix
- Reward matrix

Those are fully known environment. However, in real-life situation, we don't know before we reach the situation. This is the **limitation of model-base algorithm**. To address the problem, the **model-free algorithm** is created for no pre-required prior knowledge both transition and rewards matrices.

We need to recall the returns of **Giving Policy (G)** from a process, which are the total rewards over the long run,

$$G_t = \sum_k \gamma^k R_{t+k+1}$$

MC policy evaluation uses **empirical mean return** instead of **expected return (as in DP)** to estimate the value function. There are two ways to perform MC policy evaluation.

- First-visit MC prediction -- averages the returns only for the first occurrence of a state, s, in an episode.
- Every-visit MC prediction -- averages the returns for every occurrence of a state, s, in an episode.

First-visit MC prediction has fewer calculations than the every-visit version, so it is more frequently used.

Incremental means

To update the MC method, the incremental means have been used for update the policy. An incremental or running mean allows us to keep an average for a list of numbers without having to remember the list. We hold one value that we incrementally update using the following equation:

$$V(S_t) = V(S_t) + \alpha(G_t + V(S_t))$$

When $V(S_t)$ is the current state value for the policy, α is a discount rate, and G_t .

▼ FrozenLake and First-visit MC policy evaluation

Let's start in FrozenLake environment using first-visit MC prediction.

```
import torch
import gym

env = gym.make("FrozenLake-v1")
```

Create a function which simulate for each episode given a policy and returns the reward and state for each step

```
def run_episode(env, policy):
    state = env.reset()
    # this is the difference between MC and DP, find rewards and states
    rewards = []
    states = [state]
    is_done = False
    while not is_done:
        action = policy[state].item()
        state, reward, is_done, info = env.step(action)
        # keep all states and reward
        states.append(state)
        rewards.append(reward)
        if is_done:
            break
    # convert to torch
    states = torch.tensor(states)
    rewards = torch.tensor(rewards)
    return states, rewards
```

Create MC prediction in first-visit.

```
def mc_prediction_first_visit(env, policy, gamma, n_episode):
    n_state = policy.shape[0]
    V = torch.zeros(n_state)
    N = torch.zeros(n_state)
    for episode in range(n_episode):
        # run 1 episode until end of the episode
        states_t, rewards_t = run_episode(env, policy)
        return_t = 0
        first_visit = torch.zeros(n_state)
        G = torch.zeros(n_state)
        # take a look the state and reward from the last to first start
        # calculate given policy
        for state_t, reward_t in zip(reversed(states_t)[1:], reversed(rewards_t)):
            # calculate rewards
            # because the reward at the last can be only 0 or 1, otherwise are 0
            # so the reward at first start will be smallest
            return_t = gamma * return_t + reward_t
            # put the reward for the state into given policy
            # as you can see, if we come in the same state, it will be replaced to the early time when visit
            # That's why we call first-visit
            G[state_t] = return_t
            first_visit[state_t] = 1
        # at the end of given policy calculation
        # we need to update the state transition by summation them (prepare to average)
        for state in range(n_state):
            if first_visit[state] > 0:
                V[state] += G[state]
                N[state] += 1
        # average state transition here
        for state in range(n_state):
            if N[state] > 0:
                V[state] = V[state] / N[state]
    return V
```

Setup discount rate γ as 1 and simulate 10,000 episodes

We tried to used `optimal_policy` from lab 02 (don't be worry, we just bring the data), and feed it into the first-visit MC function, and see the results.

```

gamma = 1
n_episode = 10000

optimal_policy = torch.tensor([0., 3., 3., 3., 0., 3., 2., 3., 3., 1., 0., 3., 3., 2., 1., 3.])
value = mc_prediction_first_visit(env, optimal_policy, gamma, n_episode)
print('The value function calculated by first-visit MC prediction:\n', value)

```

```

The value function calculated by first-visit MC prediction:
tensor([0.7439, 0.5081, 0.5014, 0.4586, 0.7439, 0.0000, 0.3887, 0.0000, 0.7439,
        0.7450, 0.6733, 0.0000, 0.0000, 0.8043, 0.8929, 0.0000])

```

As you can see, in MC-based prediction, it is not necessary to know about the full model of the environment. In most real-world cases, the transition matrix and reward matrix are not known beforehand, or are extremely difficult to obtain. Imagine how many possible states there are playing *chess* or *Go* and the number of possible actions; it is almost impossible to work out the transition matrix and reward matrix. Model-free reinforcement learning is about learning from experience by interacting with the environment.

We only considered what could be observed, which included the new state and reward in each step, and made predictions using the Monte Carlo method.

Note that the more episodes we simulate, the more accurate predictions we can obtain. If you plot the value updated after each episode, you will see how it converges over time, which is similar to what we saw when estimating the value of π .

▼ How about every-visit MC?

Take a look at every-visit MC function

```

def mc_prediction_every_visit(env, policy, gamma, n_episode):
    n_state = policy.shape[0]
    V = torch.zeros(n_state)
    N = torch.zeros(n_state)
    G = torch.zeros(n_state)
    for episode in range(n_episode):
        states_t, rewards_t = run_episode(env, policy)
        return_t = 0
        for state_t, reward_t in zip(reversed(states_t)[1:], reversed(rewards_t)):
            return_t = gamma * return_t + reward_t
            # Here is the difference, instead of discard the other visit
            # we calculate all visiting in the target state
            # That's why call every-visit

```



```

        G[state_t] += return_t
        N[state_t] += 1
    for state in range(n_state):
        if N[state] > 0:
            V[state] = G[state] / N[state]
    return V

```

```

e_value = mc_prediction_every_visit(env, optimal_policy, gamma, n_episode)
print('The value function calculated by every-visit MC prediction:\n', e_value)

```

```

The value function calculated by every-visit MC prediction:
tensor([0.6208, 0.4301, 0.3951, 0.3610, 0.6190, 0.0000, 0.3649, 0.0000, 0.6371,
        0.6759, 0.6373, 0.0000, 0.0000, 0.7638, 0.8804, 0.0000])

```

Actually, the every-visit is more accurate than the first-visit, but it takes a lot of calculation, so it is not popular.

On-policy Monte Carlo control

On-policy Monte Carlo works look-a-like to policy iteration which has 2 phases: evaluation and improvement.

- Evaluation phase: it evaluates the **action-values** (called **Q-function** $Q(s, a)$) instead of evaluates the value function.
- Improvement phase: the policy is updated by assigning the optimal action to each stage: $\pi(s) = \operatorname{argmax}_a Q(s, a)$

Developing MC control with epsilon-greedy policy

The optimal policy using MC control can find the action with the highest state-action value was selected. However, the best choice available in early episodes does not guarantee an optimal solution. If we just focus on what is temporarily the best option and ignore the overall problem, we will be stuck in local optima instead of reaching the global optimal. In order to address the problem, we use another algorithm: epsilon-greedy policy.

In MC control with epsilon-greedy policy, we no longer exploit the best action all the time, but choose an action randomly under certain probabilities. As the name implies, the algorithm has two folds:

$$\pi(s, a) = \frac{\epsilon}{|A|}$$

when $|A|$ is the number of possible actions

Greedy

Greedy is the action with the highest state-action value is favored, and its probability of being chosen is increased by $1 - \epsilon$:

$$\pi(s, a) = 1 - \epsilon + \frac{\epsilon}{|A|}$$

Epsilon-greedy policy exploits the best action most of the time and also keeps exploring different actions from time to time.

▼ Modify MC control with epsilon-greedy policy with FrozenLake

From the code above, let's modify as below:

```
import torch
import gym

env = gym.make("FrozenLake-v1")
```

Modify run_episode to return states, actions, rewards

```
def run_episode(env, Q, epsilon, n_action):
    """
    Run a episode and performs epsilon-greedy policy
    @param env: OpenAI Gym environment
    @param Q: Q-function
    @param epsilon: the trade-off between exploration and exploitation
    @param n_action: action space
    @return: resulting states, actions and rewards for the entire episode
    """
    state = env.reset()
    rewards = []
    actions = []
    states = []
    is_done = False
    while not is_done:
        probs = torch.ones(n_action) * epsilon / n_action
        best_action = torch.argmax(Q[state]).item()
        probs[best_action] += 1.0 - epsilon
        action = torch.multinomial(probs, 1).item()
        actions.append(action)
```

```

states.append(state)
state, reward, is_done, info = env.step(action)
rewards.append(reward)
if is_done:
    break
return states, actions, rewards

```

Create MC_control using epsilon greedy

```

from collections import defaultdict

def mc_control_epsilon_greedy(env, gamma, n_episode, epsilon):
    """
    Obtain the optimal policy with on-policy MC control with epsilon_greedy
    @param env: OpenAI Gym environment
    @param gamma: discount factor
    @param n_episode: number of episodes
    @param epsilon: the trade-off between exploration and exploitation
    @return: the optimal Q-function, and the optimal policy
    """

    n_action = env.action_space.n
    G_sum = defaultdict(float)
    N = defaultdict(int)
    Q = defaultdict(lambda: torch.empty(n_action))
    for episode in range(n_episode):
        if (episode + 1) % 1000 == 0:
            print("Training episode {}".format(episode+1))
        states_t, actions_t, rewards_t = run_episode(env, Q, epsilon, n_action)
        return_t = 0
        G = {}
        for state_t, action_t, reward_t in zip(states_t[:-1], actions_t[:-1], rewards_t[:-1]):
            return_t = gamma * return_t + reward_t
            G[(state_t, action_t)] = return_t
        for state_action, return_t in G.items():
            state, action = state_action

            G_sum[state_action] += return_t
            N[state_action] += 1
            Q[state][action] = G_sum[state_action] / N[state_action]
    policy = {}

```

```
for state, actions in Q.items():
    policy[state] = torch.argmax(actions).item()
return Q, policy
```

Find optimal policy

```
gamma = 1

n_episode = 100000
epsilon = 0.1

optimal_Q, optimal_policy = mc_control_epsilon_greedy(env, gamma, n_episode, epsilon)
```

```
Training episode 42000
Training episode 43000
Training episode 44000
Training episode 45000
Training episode 46000
Training episode 47000
Training episode 48000
Training episode 49000
Training episode 50000
Training episode 51000
Training episode 52000
Training episode 53000
Training episode 54000
Training episode 55000
Training episode 56000
Training episode 57000
Training episode 58000
Training episode 59000
Training episode 60000
Training episode 61000
Training episode 62000
Training episode 63000
Training episode 64000
Training episode 65000
Training episode 66000
Training episode 67000
Training episode 68000
Training episode 69000
Training episode 70000
Training episode 71000
Training episode 72000
Training episode 73000
```

```
Training episode 73000
Training episode 74000
Training episode 75000
Training episode 76000
Training episode 77000
Training episode 78000
Training episode 79000
Training episode 80000
Training episode 81000
Training episode 82000
Training episode 83000
Training episode 84000
Training episode 85000
Training episode 86000
Training episode 87000
Training episode 88000
Training episode 89000
Training episode 90000
Training episode 91000
Training episode 92000
Training episode 93000
Training episode 94000
Training episode 95000
Training episode 96000
Training episode 97000
Training episode 98000
Training episode 99000
Training episode 100000
```

To test the optimal policy, we need to run simulation using policy.

Let's create `simulate_episode` function:

```
def simulate_episode(env, policy):
    state = env.reset()
    is_done = False
    while not is_done:
        action = policy[state]
        state, reward, is_done, info = env.step(action)
        if is_done:
            return reward
```

Run the optimal policy

```
n_episode = 50000
n_win_optimal = 0
n_lose_optimal = 0

for episode in range(n_episode):
    if (episode + 1) % 1000 == 0:
        print("Testing episode {}".format(episode+1))
    reward = simulate_episode(env, optimal_policy)
    if reward == 1:
        n_win_optimal += 1
    elif reward == -1:
        n_lose_optimal += 1
```

```
Testing episode 1000
Testing episode 2000
Testing episode 3000
Testing episode 4000
Testing episode 5000
Testing episode 6000
Testing episode 7000
Testing episode 8000
Testing episode 9000
Testing episode 10000
Testing episode 11000
Testing episode 12000
Testing episode 13000
Testing episode 14000
Testing episode 15000
Testing episode 16000
Testing episode 17000
Testing episode 18000
Testing episode 19000
Testing episode 20000
Testing episode 21000
Testing episode 22000
Testing episode 23000
Testing episode 24000
Testing episode 25000
Testing episode 26000
Testing episode 27000
Testing episode 28000
Testing episode 29000
Testing episode 30000
Testing episode 31000
Testing episode 32000
```

```
Testing episode 33000
Testing episode 34000
Testing episode 35000
Testing episode 36000
Testing episode 37000
Testing episode 38000
Testing episode 39000
Testing episode 40000
Testing episode 41000
Testing episode 42000
Testing episode 43000
Testing episode 44000
Testing episode 45000
Testing episode 46000
Testing episode 47000
Testing episode 48000
Testing episode 49000
Testing episode 50000
```

```
print('Winning probability under the optimal policy: {}'.format(n_win_optimal/n_episode))

print('Losing probability under the optimal policy: {}'.format(n_lose_optimal/n_episode))
```

```
Winning probability under the optimal policy: 0.25438
Losing probability under the optimal policy: 0.0
```

▼ Implement other environments: LunarLander

Let's see the environment:

```
import gym
import numpy as np
import matplotlib.pyplot as plt
from IPython import display as ipythondisplay

from pyvirtualdisplay import Display
display = Display(visible=0, size=(400, 300))
display.start()

env = gym.make('LunarLander-v2')
```

```

env.reset()
prev_screen = env.render(mode='rgb_array')
plt.imshow(prev_screen)

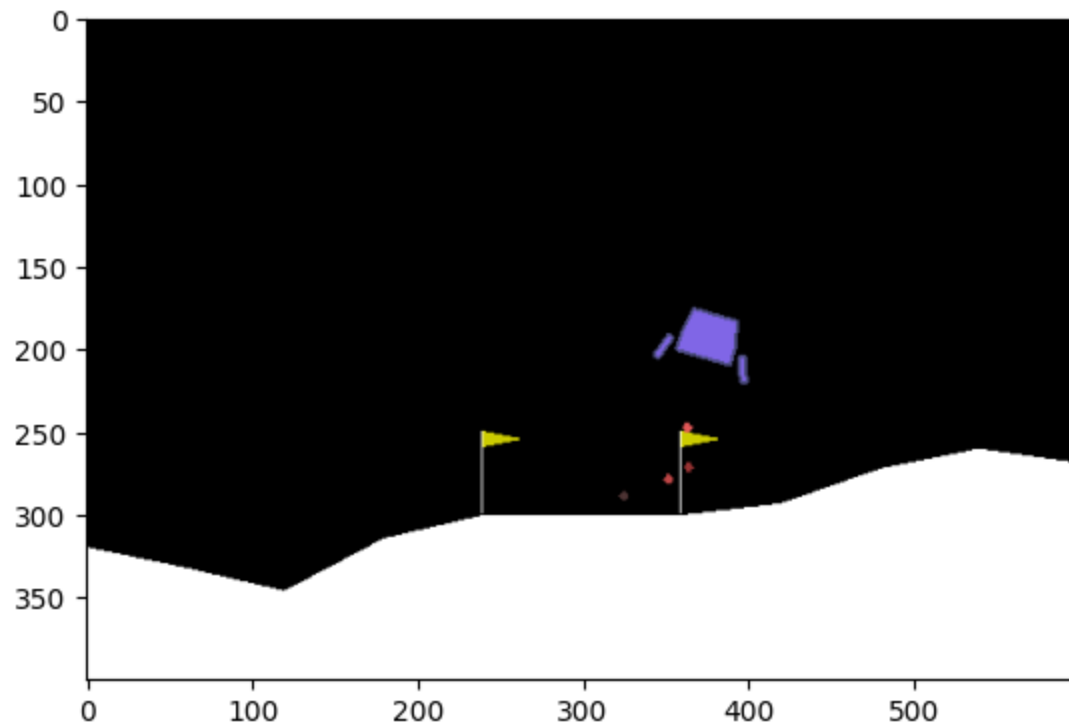
for i in range(50):
    action = env.action_space.sample()
    obs, reward, done, info = env.step(action)
    screen = env.render(mode='rgb_array')

    plt.imshow(screen)
    ipythondisplay.clear_output(wait=True)
    ipythondisplay.display(plt.gcf())

    if done:
        break

ipythondisplay.clear_output(wait=True)
env.close()

```



Checking state in the environment.


```
state = env.reset()
print(state)
```

```
[ 0.00296688  1.408531    0.3004886  -0.10618901 -0.00343099 -0.06806513
  0.          0.          ]
```

Because the state is floating array, It is not good for creating the Q table. We try to change the state to integer array and convert the state array to tuple

```
state = tuple((state * 10).astype(int))
print(state)
```

```
(0, 14, 3, -1, 0, 0, 0, 0)
```

Modify run_episode function for supporting the state.

```
import numpy as np
import torch

def run_episode2(env, Q, epsilon, n_action):
    """
    Run a episode and performs epsilon-greedy policy
    @param env: OpenAI Gym environment
    @param Q: Q-function
    @param epsilon: the trade-off between exploration and exploitation
    @param n_action: action space
    @return: resulting states, actions and rewards for the entire episode
    """
    state = env.reset()
    state = tuple((state * 10).astype(int))
    rewards = []
    actions = []
    states = []
    is_done = False
    while not is_done:
        probs = torch.ones(n_action) * epsilon / n_action
        #print(Q)
        #print(state)
```

```

best_action = torch.argmax(Q[state]).item()
probs[best_action] += 1.0 - epsilon
action = torch.multinomial(probs, 1).item()
actions.append(action)
states.append(state)
state, reward, is_done, info = env.step(action)
state = tuple((state * 10).astype(int))
rewards.append(reward)
if is_done:
    break
return states, actions, rewards

```

```

from collections import defaultdict

```

```

def mc_control_epsilon_greedy2(env, gamma, n_episode, epsilon):
    """
    Obtain the optimal policy with on-policy MC control with epsilon_greedy
    @param env: OpenAI Gym environment
    @param gamma: discount factor
    @param n_episode: number of episodes
    @param epsilon: the trade-off between exploration and exploitation
    @return: the optimal Q-function, and the optimal policy
    """

    n_action = env.action_space.n
    G_sum = defaultdict(float)
    N = defaultdict(int)
    Q = defaultdict(lambda: torch.empty(n_action))
    for episode in range(n_episode):
        if (episode + 1) % 500 == 0:
            print("Training episode {}".format(episode+1))
        states_t, actions_t, rewards_t = run_episode2(env, Q, epsilon, n_action)
        return_t = 0
        G = {}
        for state_t, action_t, reward_t in zip(states_t[::-1], actions_t[::-1], rewards_t[::-1]):
            return_t = gamma * return_t + reward_t
            G[(state_t, action_t)] = return_t
        for state_action, return_t in G.items():
            state, action = state_action

            G_sum[state_action] += return_t
            N[state_action] += 1

```

```

        Q[state][action] = G_sum[state_action] / N[state_action]
    policy = {}
    for state, actions in Q.items():
        policy[state] = torch.argmax(actions).item()
    return Q, policy

```

```
gamma = 0.5
```

```
n_episode = 5000
```

```
epsilon = 0.1
```

```
optimal_Q, optimal_policy = mc_control_epsilon_greedy2(env, gamma, n_episode, epsilon)
```

```

Training episode 500
Training episode 1000
Training episode 1500
Training episode 2000
Training episode 2500
Training episode 3000
Training episode 3500
Training episode 4000
Training episode 4500
Training episode 5000

```

```
len(optimal_policy)
```

```
188934
```

```

def simulate_episode_render(env, policy):
    state = env.reset()
    state = tuple((state * 10).astype(int))
    is_done = False
    while not is_done:
        try:
            action = policy[state]
            print("get action")
        except:
            action = 0
            print("no action")
        print(action)
        state, reward, is_done, info = env.step(action)

```

```
state = tuple((state * 10).astype(int))
screen = env.render(mode='rgb_array')

plt.imshow(screen)
ipythondisplay.clear_output(wait=True)
ipythondisplay.display(plt.gcf())
if is_done:
    return reward
```

```
from gym.wrappers import Monitor
```

```
vdo_path = 'video_rl/'
env = Monitor(gym.make('LunarLander-v2'), vdo_path, force=True)
env.reset()
prev_screen = env.render(mode='rgb_array')
plt.imshow(prev_screen)
```

```
simulate_episode_render(env, optimal_policy)
```

```
ipythondisplay.clear_output(wait=True)
env.close()
```

▼ Another example: Blackjack

```
import torch
import gym
from collections import defaultdict

env = gym.make('Blackjack-v1')

def run_episode(env, Q, epsilon, n_action):
    state = env.reset()
    rewards = []
    actions = []
    states = []
    is_done = False
    # without epsilon-greedy
    # action = torch.randint(0, n_action, [1]).item()
    #####
    while not is_done:
        # with epsilon-greedy
        probs = torch.ones(n_action) * epsilon / n_action
        best_action = torch.argmax(Q[state]).item()
        probs[best_action] += 1.0 - epsilon
        action = torch.multinomial(probs, 1).item()
        #####
        actions.append(action)
        states.append(state)
        state, reward, is_done, info = env.step(action)
        rewards.append(reward)
    return states, actions, rewards

def mc_control_on_policy(env, gamma, n_episode, epsilon):
    n_action = env.action_space.n
    G_sum = defaultdict(float)
    N = defaultdict(int)
    Q = defaultdict(lambda: torch.empty(env.action_space.n))
    for episode in range(n_episode):
        states_t, actions_t, rewards_t = run_episode(env, Q, epsilon, n_action)
        return_t = 0
        G = {}
```

```

    for state_t, action_t, reward_t in zip(states_t[:-1], actions_t[:-1], rewards_t[:-1]):
        return_t = gamma * return_t + reward_t
        G[(state_t, action_t)] = return_t
        for state_action, return_t in G.items():
            state, action = state_action
            if state[0] <= 21:
                G_sum[state_action] += return_t
                N[state_action] += 1
                Q[state][action] = G_sum[state_action] / N[state_action]

    policy = {}
    for state, actions in Q.items():
        policy[state] = torch.argmax(actions).item()
    return Q, policy

gamma = 1
n_episode = 500000
epsilon = 0.1
optimal_Q, optimal_policy = mc_control_on_policy(env, gamma, n_episode, epsilon)
# print(optimal_policy)
# print(optimal_Q)

def simulate_episode(env, policy):
    state = env.reset()
    is_done= False
    while not is_done:
        action = policy[state]
        state, reward, is_done, info = env.step(action)
        if is_done:
            return reward

n_episode = 100
n_win_optimal = 0
n_lose_optimal = 0
for _ in range(n_episode):
    reward = simulate_episode(env, optimal_policy)
    if reward == 1:
        n_win_optimal += 1
    elif reward == -1:
        n_lose_optimal += 1
print('after episode 100, win ', n_win_optimal, ' lose ', n_lose_optimal)

```

after episode 100, win 47 lose 47

▼ Off-policy Monte Carlo control

The Off-policy method optimizes the **target policy** (π) using data generated by another policy (**behavior policy** (b)).

- Target policy: exploitation purposes, greedy with respect to its current Q-function.
- Behavior policy: exploration purposes, generate behavior which the target policy used for learning. The behavior policy can be anything to confirm that it can explore all possibilities, then all actions and all states can be chosen with non-zero probabilities.

The weight important for state-action pair is calculated as:

$$w_t = \sum_{k=t} [\pi(a_k | s_k) / b(a_k | s_k)]$$

- $\pi(a_k | s_k)$: probabilities of taking action a_k in state s_k
- $b(a_k | s_k)$: probabilities under the behavior policy.

```
import torch
import gym
from collections import defaultdict

env = gym.make('Blackjack-v1')

def gen_random_policy(n_action):
    probs = torch.ones(n_action) / n_action
    def policy_function(state):
        return probs
    return policy_function

random_policy = gen_random_policy(env.action_space.n)

def run_episode(env, behavior_policy):
    state = env.reset()
    rewards = []
    actions = []
    states = []
    is_done = False
    while not is_done:
        probs = behavior_policy(state)
        action = torch.multinomial(probs, 1).item()
        actions.append(action)
        states.append(state)
```

```

        state, reward, is_done, info = env.step(action)
        rewards.append(reward)
    if is_done:
        break
    return states, actions, rewards

def mc_control_off_policy(env, gamma, n_episode, behavior_policy):
    n_action = env.action_space.n
    G_sum = defaultdict(float)
    N = defaultdict(int)
    Q = defaultdict(lambda: torch.empty(n_action))
    for episode in range(n_episode):
        W = {}
        w = 1
        states_t, actions_t, rewards_t = run_episode(env, behavior_policy)
        return_t = 0
        G = {}
        for state_t, action_t, reward_t in zip(states_t[::-1], actions_t[::-1], rewards_t[::-1]):
            return_t = gamma * return_t + reward_t
            G[(state_t, action_t)] = return_t
            w *= 1. / behavior_policy(state_t)[action_t]
            W[(state_t, action_t)] = w
            if action_t != torch.argmax(Q[state_t]).item():
                break

        for state_action, return_t in G.items():
            state, action = state_action
            if state[0] <= 21:
                G_sum[state_action] += return_t * W[state_action]
                N[state_action] += 1
                Q[state][action] = G_sum[state_action] / N[state_action]
    policy = {}
    for state, actions in Q.items():
        policy[state] = torch.argmax(actions).item()
    return Q, policy

gamma = 1
n_episode = 500000
optimal_Q, optimal_policy = mc_control_off_policy(env, gamma, n_episode, random_policy)
# print(optimal_policy)
# print(optimal_Q)

```



```

def simulate_episode(env, policy):
    state = env.reset()
    is_done= False
    while not is_done:
        action = policy[state]
        state, reward, is_done, info = env.step(action)
        if is_done:
            return reward

n_episode = 100
n_win_optimal = 0
n_lose_optimal = 0
for _ in range(n_episode):
    reward = simulate_episode(env, optimal_policy)
    if reward == 1:
        n_win_optimal += 1
    elif reward == -1:
        n_lose_optimal += 1
print('after episode 100, win ', n_win_optimal, ' lose ', n_lose_optimal)

```

```

after episode 100, win  30  lose  58

```

Temporal Difference (TD)

Makov Decision Process (MDP) using **Monte Carlo method** which it a *model-free* approach. It does not require the prior knowledge environment. However, all algorithm that we learned, need to play until the end of episode, then it can update its knowledge. This is slow due to waiting some process to success or fail to terminate.

For **Temporal Difference(TD)**, it is another method that can learn while it's running each step because learning every time step in an episode, TD increased learning efficiency significantly.

Temporal Difference (TD) learning is a model-free learning algorithm like MC learning. In MC learning, Q-function is called and updated at the end of the entire episode, but TD learning update Q-function every step of an episode. One of the TD learning algorithm is Q-learning

Algorithms in TD method

The TD method has 2 main algorithms family:

1. Q-Learning
2. SARSA

Q-Learning

Q-Learning is a very popular TD method. It is an off-policy learning algorithm. The Q-function updates the Q policy based on the equation:

$$Q(s, a) = Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a))$$

where

- s: current state
- s': next state after taking action
- a: action in the current state
- a': action will take when go to next state
- α : learning rate
- γ : discount factor

$\max_{a'} Q(s', a')$ means the behavior policy is greedy, where the highest Q-value among those in state s' is selected to generate learning data.

In Q-learning, actions are taken according to the epsilon-greedy policy.

▼ Q-Learning example: SpaceInvaders

Now, let's do the Q-Learning algorithm using SpaceInvaders environment.

```
import torch
import gym
from collections import defaultdict
import numpy

env_id = "SpaceInvaders-v0"
env = gym.make(env_id)

env.reset()
```

```
[Powered by Stella]
array([[[ 0,  0,  0],
        [ 0,  0,  0],
        [ 0,  0,  0],
        ...,
        [ 0,  0,  0],
        [ 0,  0,  0],
        [ 0,  0,  0]],

       [[ 0,  0,  0],
        [ 0,  0,  0],
        [ 0,  0,  0],
        ...,
        [ 0,  0,  0],
        [ 0,  0,  0],
        [ 0,  0,  0]],

       [[ 0,  0,  0],
        [ 0,  0,  0],
        [ 0,  0,  0],
        ...,
        [ 0,  0,  0],
        [ 0,  0,  0],
        [ 0,  0,  0]],

       ...,

       [[80, 89, 22],
        [80, 89, 22],
        [80, 89, 22],
        ...,
        [80, 89, 22],
        [80, 89, 22],
        [80, 89, 22]],

       [[80, 89, 22],
        [80, 89, 22],
        [80, 89, 22],
        ...,
        [80, 89, 22],
        [80, 89, 22],
        [80, 89, 22]],

       [[80, 89, 22],
        [80, 89, 22],
        [80, 89, 22],
        ...,
```

```
[80, 89, 22],  
[80, 89, 22],  
[80, 89, 22]]], dtype=uint8)
```

▼ Define Epsilon greedy policy

As last time the epsilon greedy policy equation can be written as:

$$\pi(s, a) = \frac{\epsilon}{|A|}$$

when $|A|$ is the number of possible actions, and

$$\pi(s, a) = 1 - \epsilon + \frac{\epsilon}{|A|}$$

```
def gen_epsilon_greedy_policy(n_action, epsilon):  
    def policy_function(state, Q, available_actions):  
        probs = torch.ones(n_action) * epsilon / n_action  
        best_action = torch.argmax(Q[state]).item()  
        if not(best_action in available_actions):  
            best_action = -1  
            Q_max = -800000000  
            for i in range(n_action):  
                if i in available_actions and Q_max < Q[state][i]:  
                    Q_max = Q[state][i]  
                    best_action = i  
        probs[best_action] += 1.0 - epsilon  
        action = torch.multinomial(probs, 1).item()  
        return action  
    return policy_function
```

▼ Define Q-learning function

We perform Q-learning in the tasks:

1. Initialize the Q-table with all zeros.
2. In each episode, we let the agent follow the epsilon-greedy policy to choose what action to take. And we update the Q function for each step.
3. Run `n_episodes` episodes

4. Obtain the optimal policy based on the optimal Q function.

```
def q_learning(env, gamma, n_episode, alpha, player):
    """
    Obtain the optimal policy with off-policy Q-learning method
    @param env: OpenAI Gym environment
    @param gamma: discount factor
    @param n_episode: number of episodes
    @return: the optimal Q-function, and the optimal policy
    """

    n_action = 9
    Q = defaultdict(lambda: torch.zeros(n_action))
    print('start learning')
    for episode in range(n_episode):
        print("episode: ", episode + 1)
        state = env.reset()
        state = hash(tuple(state.reshape(-1)))

        is_done = False
        while not is_done:
            action = epsilon_greedy_policy(state, Q, available_actions)
            next_state, reward, is_done, info = env.step(action)
            next_state = hash(tuple(next_state.reshape(-1)))
            td_delta = reward + gamma * torch.max(Q[next_state]) - Q[state][action]
            Q[state][action] += alpha * td_delta

            length_episode[episode] += 1
            total_reward_episode[episode] += reward

            if is_done:
                break
            state = next_state

    policy = {}
    for state, actions in Q.items():
        policy[state] = torch.argmax(actions).item()
    return Q, policy
```

We specify the $\gamma = 1$, $\alpha = 0.4$, and $\epsilon = 0.1$ with 500 episodes

```
gamma = 1
alpha = 0.4
epsilon = 0.1

n_episode = 500
```

▼ Create an instance of the epsilon-greedy policy function

```
available_actions = numpy.arange(env.action_space.n)
available_actions = torch.from_numpy(available_actions)
epsilon_greedy_policy = gen_epsilon_greedy_policy(env.action_space.n, epsilon)

print(epsilon_greedy_policy)

<function gen_epsilon_greedy_policy.<locals>.policy_function at 0x7f985ba611f0>
```

▼ Train it!

```
length_episode = [0] * n_episode
total_reward_episode = [0] * n_episode

# agent play first
optimal_Q, optimal_policy = q_learning(env, gamma, n_episode, alpha, 1)

print('The optimal policy:\n', optimal_policy)
```

```
episode: 267
episode: 268
episode: 269
episode: 270
episode: 271
episode: 272
episode: 273
episode: 274
episode: 275
```

episode: 276
episode: 277
episode: 278
episode: 279
episode: 280
episode: 281
episode: 282
episode: 283
episode: 284
episode: 285
episode: 286
episode: 287
episode: 288
episode: 289
episode: 290
episode: 291
episode: 292
episode: 293
episode: 294
episode: 295
episode: 296
episode: 297
episode: 298
episode: 299
episode: 300
episode: 301
episode: 302
episode: 303
episode: 304
episode: 305
episode: 306
episode: 307
episode: 308
episode: 309
episode: 310
episode: 311
episode: 312
episode: 313
episode: 314
episode: 315
episode: 316
episode: 317
episode: 318
episode: 319
episode: 320
episode: 321
episode: 322

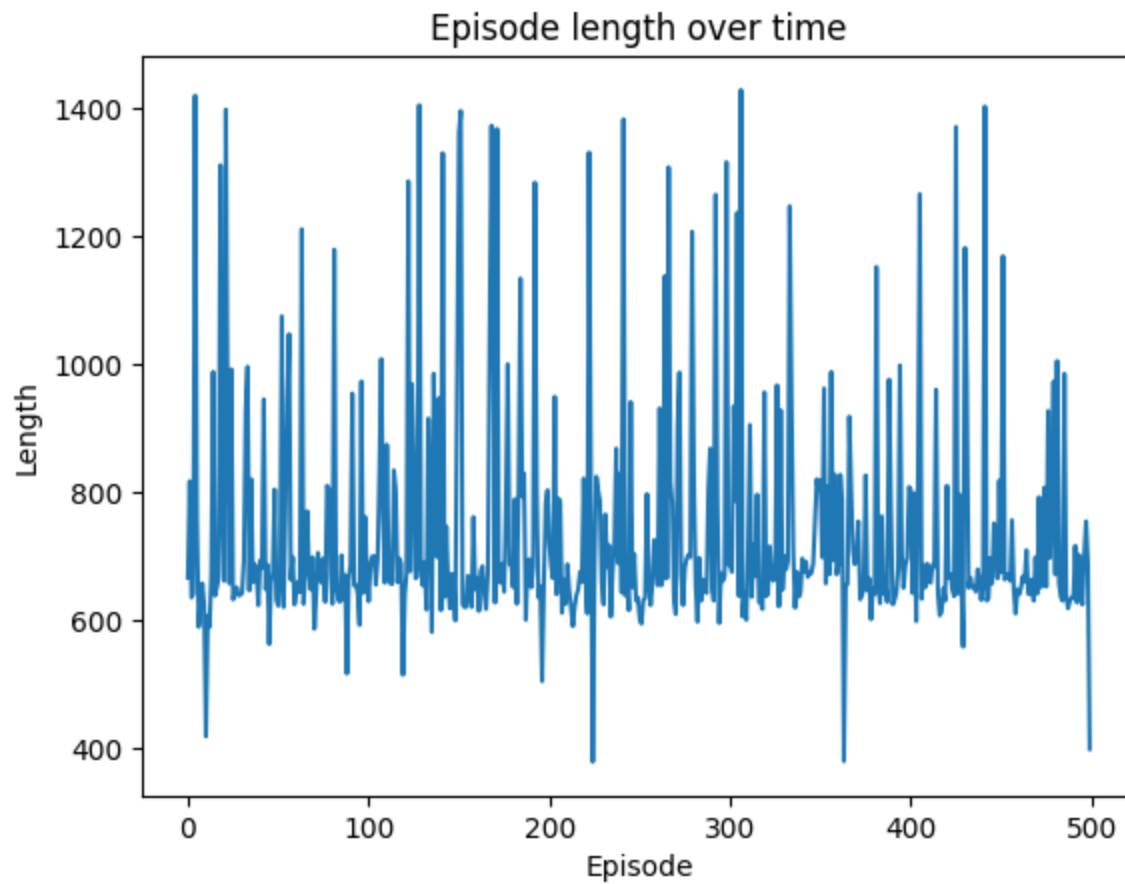
```
episode: 321  
episode: 323  
episode: 324  
episode: 325
```

▼ Display the plot of episode

```
import matplotlib.pyplot as plt

plt.plot(length_episode)
plt.title('Episode length over time')
plt.xlabel('Episode')
plt.ylabel('Length')
plt.show()

plt.plot(total_reward_episode)
print(total_reward_episode[-100:])
plt.title('Episode reward over time')
plt.xlabel('Episode')
plt.ylabel('Total reward')
plt.show()
```

[120.0, 60.0, 120.0, 30.0, 75.0, 300.0, 30.0, 20.0, 75.0, 105.0, 50.0, 55.0, 50.0, 105.0, 30.0, 50.0, 30.0, 15.0, 15.0, 15.0, 35.0]



▼ Simulate the optimal policy

```
import time
from IPython import display
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline

def play_game(Q, available_actions):
```

```
done = False
plt.figure(figsize=(9,9))
img = plt.imshow(env.render(mode='rgb_array')) # only call this once
state = env.reset()
state = hash(tuple(state.reshape(-1)))
while(not done):
    action = epsilon_greedy_policy(state, Q, available_actions)
    next_state, reward, done, _ = env.step(action)
    # env.render() No use!
    img.set_data(env.render(mode='rgb_array')) # just update the data
    display.display(plt.gcf())
    display.clear_output(wait=True)
    time.sleep(0.03)
    state = next_state
    state = hash(tuple(state.reshape(-1)))

play_game(optimal_Q, available_actions)
env.close()
```



```
!pip install gym pyvirtualdisplay > /dev/null 2>&1
!pip install -U gym==0.21.0
!pip install -U gym[atari,accept-rom-license]
!pip install PyVirtualDisplay
!sudo apt-get install xvfb
!pip install pygame
```

Successfully installed gym-0.21.0

Looking in indexes: <https://pypi.org/simple>, <https://us-python.pkg.dev/colab-wheels/public/simple/>

Requirement already satisfied: gym[accept-rom-license,atari] in /usr/local/lib/python3.7/dist-packages (0.21.0)

Requirement already satisfied: importlib-metadata>=4.8.1 in /usr/local/lib/python3.7/dist-packages (from gym[accept-rom-license,atari]) (4.8.1)

Requirement already satisfied: cloudpickle>=1.2.0 in /usr/local/lib/python3.7/dist-packages (from gym[accept-rom-license,atari]) (1.6.0)

Requirement already satisfied: numpy>=1.18.0 in /usr/local/lib/python3.7/dist-packages (from gym[accept-rom-license,atari]) (1.19.5)

Collecting ale-py~0.7.1

Downloading ale_py-0.7.5-cp37-cp37m-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (1.6 MB)

1.6 MB 12.8 MB/s

Collecting autorom[accept-rom-license]~0.4.2

Downloading AutoROM-0.4.2-py3-none-any.whl (16 kB)

Requirement already satisfied: importlib-resources in /usr/local/lib/python3.7/dist-packages (from ale-py~0.7.1->gym[accept-rom-license,atari]) (5.10.0)

Requirement already satisfied: tqdm in /usr/local/lib/python3.7/dist-packages (from autorom[accept-rom-license]~0.4.2->gym[accept-rom-license,atari]) (4.64.0)

Requirement already satisfied: click in /usr/local/lib/python3.7/dist-packages (from autorom[accept-rom-license]~0.4.2->gym[accept-rom-license,atari]) (8.0.4)

Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-packages (from autorom[accept-rom-license]~0.4.2->gym[accept-rom-license,atari]) (2.27.0)

Collecting AutoROM.accept-rom-license

Downloading AutoROM.accept-rom-license-0.4.2.tar.gz (9.8 kB)

Installing build dependencies ... done

Getting requirements to build wheel ... done

Preparing wheel metadata ... done

Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.7/dist-packages (from importlib-resources->ale-py~0.7.1->gym[accept-rom-license,atari]) (3.6.0)

```

Requirement already satisfied: typing-extensions>=3.6.4 in /usr/local/lib/python3.7/dist-packages (from importlib-metadata>=4.1
Requirement already satisfied: urllib3!=1.25.0,!1.25.1,<1.26,>=1.21.1 in /usr/local/lib/python3.7/dist-packages (from request
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7/dist-packages (from requests->autorom[accept-rom-
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-packages (from requests->autorom[accept-rom-licen
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.7/dist-packages (from requests->autorom[accept-rom-
Building wheels for collected packages: AutoROM.accept-rom-license
  Building wheel for AutoROM.accept-rom-license (PEP 517) ... done
  Created wheel for AutoROM.accept-rom-license: filename=AutoROM.accept_rom_license-0.4.2-py3-none-any.whl size=441028 sha256=6
  Stored in directory: /root/.cache/pip/wheels/87/67/2e/6147e7912fe37f5408b80d07527dab807c1d25f5c403a9538a
Successfully built AutoROM.accept-rom-license
Installing collected packages: AutoROM.accept-rom-license, autorom, ale-py
Successfully installed AutoROM.accept-rom-license-0.4.2 ale-py-0.7.5 autorom-0.4.2
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: PyVirtualDisplay in /usr/local/lib/python3.7/dist-packages (3.0)
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following package was automatically installed and is no longer required:
  libnvidia-common-460
Use 'sudo apt autoremove' to remove it.
The following NEW packages will be installed:
  xvfb
0 upgraded, 1 newly installed, 0 to remove and 5 not upgraded.
Need to get 785 kB of archives.
After this operation, 2,271 kB of additional disk space will be used.
Ign:1 http://archive.ubuntu.com/ubuntu bionic-updates/universe amd64 xvfb amd64 2:1.19.6-1ubuntu4.11
Err:1 http://security.ubuntu.com/ubuntu bionic-updates/universe amd64 xvfb amd64 2:1.19.6-1ubuntu4.11
404 Not Found [IP: 185.125.190.39 80]
E: Failed to fetch http://security.ubuntu.com/ubuntu/pool/universe/x/xorg-server/xvfb\_1.19.6-1ubuntu4.11\_amd64.deb 404 Not F
E: Unable to fetch some archives, maybe run apt-get update or try with --fix-missing?
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting pygame
  Downloading pygame-2.0.0-py3-none-any.whl (966 kB)
    |████████████████████████████████████████| 966 kB 15.4 MB/s
Installing collected packages: pygame
Successfully installed pygame-2.0.0

```

▼ Deep Q-Learning

A deep Q-Network (DQN) is similar to a supervised regression model F_{θ} , but it more specifically maps states to action values directly instead of using a set of features.

A DQN is trained to output $Q(s, a)$ values for each action given the input state s . In operation, in state s , the action a is chosen greedily based on $Q(s, a)$ or stochastically following an epsilon-greedy policy.

In tabular Q learning, the update rule is an off-policy TD learning rule. When we take action a in state s receiving reward r , we update $Q(s, a)$ as

$$Q(s, a) = Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a)),$$

where

- s' is the resulting state after taking action a in state s
- $\max_{a'} Q(s', a')$ is value of the action a' we would take in state s' according to a greedy behavior policy.

A DQN does the same thing using backpropagation, minimizing inconsistencies in its Q estimates. At each step, the difference between the estimated value and the observed data from the subsequent step should be minimized, giving us a kind of regression problem, for which a squared error loss function is appropriate, giving us a delta for the a th output of

$$\delta_a = r + \gamma \max_{a'} Q(s')_{a'} - Q(s)_a.$$

With an appropriate exploration strategy and learning rate, DQN should find the optimal network model best approximating the state-value function $Q(s, a)$ for each possible state and action.

DQN Example: Cartpole

Let's develop a sample DQN application step by step. First, some imports we'll need.

Note: From now, the coding style have changed for supporting more advance coding.

```
import math, random

import torch
import torch.nn as nn
import torch.optim as optim
import torch.autograd as autograd
import torch.nn.functional as F

import matplotlib.pyplot as plt

import gym
import numpy as np
```

```
from collections import deque
from tqdm import trange

# Select GPU or CPU as device

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

▼ ϵ decay schedule

ϵ is how much the model use the policy action as random. At first, we want ϵ high as possible to explore the new path, and the ϵ will decreased to exploit the policy at the old state that we have explored.

Recall that some of the theoretical results on TD learning assume ϵ -greedy exploration with ϵ decaying slowly to 0 over time. Let's define an exponential decay schedule for ϵ . First, an example:

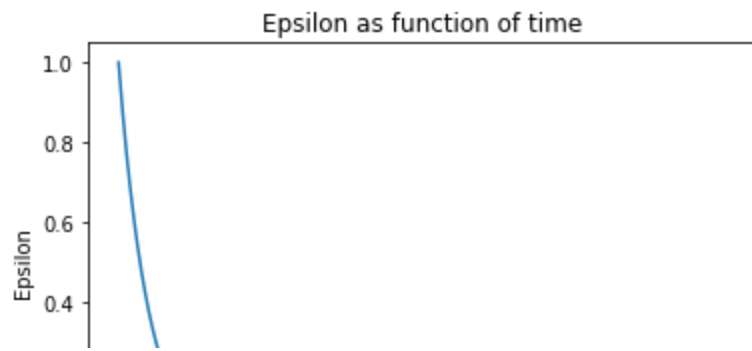
```
epsilon_start = 1.0
epsilon_final = 0.01
epsilon_decay = 500

# Define epsilon as a function of time (episode index)

eps_by_episode = lambda episode: epsilon_final + (epsilon_start - epsilon_final) * math.exp(-1. * episode / epsilon_decay)

# Note that the above lambda expression is equivalent to explicitly defining a function:
# def epsilon_episode(episode):
#     return epsilon_final + (epsilon_start - epsilon_final) * math.exp(-1. * episode / epsilon_decay)

plt.plot([eps_by_episode(i) for i in range(10000)])
plt.title('Epsilon as function of time')
plt.xlabel('Time (episode index)')
plt.ylabel('Epsilon')
plt.show()
```



Here's a reusable function to generate an annealing schedule function according to given parameters:

```
# Epsilon annealing schedule generator

def gen_eps_by_episode(epsilon_start, epsilon_final, epsilon_decay):
    eps_by_episode = lambda episode: epsilon_final + (epsilon_start - epsilon_final) * math.exp(-1. * episode / epsilon_decay)
    return eps_by_episode

epsilon_start = 1.0
epsilon_final = 0.01
epsilon_decay = 500
eps_by_episode = gen_eps_by_episode(epsilon_start, epsilon_final, epsilon_decay)
```

▼ Replay buffer

We know that deep learning methods learn faster when training samples are combined into batches. This speeds up learning and also makes it more stable by averaging updates over multiple samples.

RL algorithms also benefit from batched training. However, we see that the standard Q learning rule always updates Q estimates using the most recent experience. If we always trained on batches consisting of samples of the most recent behavior, correlations between successive state action pairs will make learning less effective. So we would also like to select random training samples to make them look more like the i.i.d. sampling that supervised learning performs well under.

In RL, the standard way of doing this is to create a large buffer of past state action pairs then form training batches by sampling from that replay buffer. Our replay buffer will store tuples consisting of an observed state, an action, the next_state, the reward, and the termination signal obtained by the agent at that point in time:

```
class ReplayBuffer(object):
```

```

def __init__(self, capacity):
    self.buffer = deque(maxlen=capacity)

def push(self, state, action, reward, next_state, done):
    # Add batch index dimension to state representations
    state = np.expand_dims(state, 0)
    next_state = np.expand_dims(next_state, 0)
    self.buffer.append((state, action, reward, next_state, done))

def sample(self, batch_size):
    state, action, reward, next_state, done = zip(*random.sample(self.buffer, batch_size))
    return np.concatenate(state), action, reward, np.concatenate(next_state), done

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

```

▼ Basic DQN

Next, a basic DQN class. We just create a neural network that takes as input a state and returns an output vector indicating the value of each possible action $Q(s, a)$.

The steps we take during learning will be as follows:

To implement the policy, besides the usual `forward()` method, we add one additional method `act()`, which samples an ϵ -greedy action for state s using the current estimate $Q(s, a)$. `act()` will be used to implement step 1 in the pseudocode above.

```

class DQN(nn.Module):

    def __init__(self, n_state, n_action):
        super(DQN, self).__init__()
        self.layers = nn.Sequential(
            nn.Linear(n_state, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, n_action)
        )

```



```

def forward(self, x):
    return self.layers(x)

def act(self, state, epsilon):
    # Get an epsilon greedy action for given state
    if random.random() > epsilon: # Use argmax_a Q(s,a)
        state = autograd.Variable(torch.FloatTensor(state).unsqueeze(0), volatile=True).to(device)
        q_value = self.forward(state)
        q_value = q_value.cpu()
        action = q_value.max(1)[1].item()
    else: # get random action
        action = random.randrange(env.action_space.n)
    return action

```

▼ Create gym environment, prepare DQN for training

Next we set up a gym environment for the cartpole simulation, create a DQN model with Adam optimization, and create a replay buffer of length 1000.

```

env_id = "CartPole-v0"
env = gym.make(env_id)

model = DQN(env.observation_space.shape[0], env.action_space.n).to(device)

optimizer = optim.Adam(model.parameters())

replay_buffer = ReplayBuffer(3000)

```

▼ Training step

In the training step, we sample a batch from the replay buffer, calculate $Q(s, a)$ and $\max_{a'} Q(s', a')$, calculate the target Q value $r + \gamma \max_{a'} Q(s', a')$, the mean squared loss between the predicted and target Q values, and then backpropagate.

```

def compute_td_loss(model, batch_size, gamma=0.99):

    # Get batch from replay buffer
    state, action, reward, next_state, done = replay_buffer.sample(batch_size)

```

```

# Convert to tensors. Creating Variables is not necessary with more recent PyTorch versions.
state          = autograd.Variable(torch.FloatTensor(np.float32(state))).to(device)
next_state     = autograd.Variable(torch.FloatTensor(np.float32(next_state))), volatile=True).to(device)
action         = autograd.Variable(torch.LongTensor(action)).to(device)
reward         = autograd.Variable(torch.FloatTensor(reward)).to(device)
done           = autograd.Variable(torch.FloatTensor(done)).to(device)

# Calculate Q(s) and Q(s')
q_values       = model(state)
# print(q_values.shape)
next_q_values  = model(next_state)

# Get Q(s,a) and max_a' Q(s',a')
#print("action.unsqueeze(1)", action.unsqueeze(1))
# print(action.unsqueeze(1).shape)
q_value        = q_values.gather(1, action.unsqueeze(1)).squeeze(1)

next_q_value    = next_q_values.max(1)[0]
#print("next_q_value", next_q_value)
# Calculate target for Q(s,a): r + gamma max_a' Q(s',a')
# Note that the done signal is used to terminate recursion at end of episode.
expected_q_value = reward + gamma * next_q_value * (1 - done)

# Calculate MSE loss. Variables are not needed in recent PyTorch versions.
loss = (q_value - autograd.Variable(expected_q_value.data)).pow(2).mean()

optimizer.zero_grad()
loss.backward()
optimizer.step()

return loss

```

▼ Plot rewards and losses

Here's a little function to plot relevant details for us:

```

def plot(episode, rewards, losses):
    # clear_output(True)
    plt.figure(figsize=(20,5))
    plt.subplot(131)
    plt.title('episode %s. reward: %s' % (episode, np.mean(rewards[-10:])))

```

```
plt.plot(rewards)
plt.subplot(132)
plt.title('loss')
plt.plot(losses)
plt.show()
```

▼ Training loop

The training loop lets the agent play the game until the end of the episode. Each step is appended to the replay buffer. We don't do any learning until the buffer's length reaches the batch_size.

```
def train(env, model, eps_by_episode, optimizer, replay_buffer, episodes = 100, batch_size=32, gamma = 0.99):
    losses = []
    all_rewards = []
    episode_reward = 0
    tot_reward = 0
    tr = trange(episodes+1, desc='Agent training', leave=True)

    # Get initial state input
    state = env.reset()

    # Execute episodes iterations
    for episode in tr:
        tr.set_description("Agent training (episode{}) Avg Reward {}".format(episode+1,tot_reward/(episode+1)))
        tr.refresh()

        # Get initial epsilon greedy action
        epsilon = eps_by_episode(episode)
        action = model.act(state, epsilon)

        # Take a step
        next_state, reward, done, _ = env.step(action)

        # Append experience to replay buffer
        replay_buffer.push(state, action, reward, next_state, done)

        tot_reward += reward
        episode_reward += reward

        state = next_state
```

```
# Start a new episode if done signal is received
if done:
    state = env.reset()
    all_rewards.append(episode_reward)
    episode_reward = 0

# Train on a batch if we've got enough experience
# print(len(replay_buffer))
if len(replay_buffer) > batch_size:
    loss = compute_td_loss(model, batch_size, gamma)
    losses.append(loss.item())

plot(episode, all_rewards, losses)
return model, all_rewards, losses
```

▼ Train!

Let's train our DQN model for 10,000 steps in the cartpole simulation:

```
model, all_rewards, losses = train(env, model, eps_by_episode, optimizer, replay_buffer, episodes = 100, batch_size=32, gamma = 0.99)
```

Agent training (episode19) Avg Reward 0.9473684210526315:	14%	14/101 [00:00<00:01, 63.80it/s]	/usr/local/lib/python3
Agent training (episode33) Avg Reward 0.9696969696969697:	30%	30/101 [00:00<00:01, 59.17it/s]	/usr/local/lib/python3

▼ Play in the simulation

You can run your simulation in Jupyter if you have OpenGL installed, but after the simulation is finished, you must close the simulator with `env.close()` (this closes the simulator, not the environment).

```
import time
from IPython import display
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline

def play_game(model):
    done = False
    plt.figure(figsize=(9,9))
    img = plt.imshow(env.render(mode='rgb_array')) # only call this once
    state = env.reset()
    while(not done):
        action = model.act(state, epsilon_final)
        next_state, reward, done, _ = env.step(action)
        # env.render() No use!
        img.set_data(env.render(mode='rgb_array')) # just update the data
        display.display(plt.gcf())
        display.clear_output(wait=True)
        time.sleep(0.03)
        state = next_state

play_game(model)
env.close()
```

▼ In class exercise (50 points)

```
!pip install box2d-py
!pip install gym[Box_2D]
```

Looking in indexes: <https://pypi.org/simple>, <https://us-python.pkg.dev/colab-wheels/public/simple/>

```
Collecting box2d-py
  Downloading box2d_py-2.3.8-cp37-cp37m-manylinux1_x86_64.whl (448 kB)
    |████████████████████████████████████████| 448 kB 13.4 MB/s
Installing collected packages: box2d-py
Successfully installed box2d-py-2.3.8
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: gym[Box_2D] in /usr/local/lib/python3.7/dist-packages (0.21.0)
WARNING: gym 0.21.0 does not provide the extra 'box_2d'
Requirement already satisfied: numpy>=1.18.0 in /usr/local/lib/python3.7/dist-packages (from gym[Box_2D]) (1.21.6)
Requirement already satisfied: cloudpickle>=1.2.0 in /usr/local/lib/python3.7/dist-packages (from gym[Box_2D]) (1.5.0)
Requirement already satisfied: importlib-metadata>=4.8.1 in /usr/local/lib/python3.7/dist-packages (from gym[Box_2D]) (4.13.0)
Requirement already satisfied: typing-extensions>=3.6.4 in /usr/local/lib/python3.7/dist-packages (from importlib-metadata>=4.8.1)
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.7/dist-packages (from importlib-metadata>=4.8.1->gym[Box_2D])
```

```
env_id = 'PongDeterministic-v4'
env = gym.make(env_id)
```

```
print(env.unwrapped.get_action_meanings())
print(env.observation_space.shape)
```

```
['NOOP', 'FIRE', 'RIGHT', 'LEFT', 'RIGHTFIRE', 'LEFTFIRE']
(210, 160, 3)
```

```
import torchvision.transforms as T
from PIL import Image
image_size = 84

transform = T.Compose([T.ToPILImage(), # from tensors to image data
                      T.Grayscale(num_output_channels=1), # convert to grayscale with 1 channel
                      T.Resize((image_size, image_size), interpolation=Image.CUBIC), # resize to 84*84 by using Cubic interpolation
                      T.ToTensor()]) # convert back to tensor

def get_state(observation):
    # Numpy: Use transpose(a, argsort(axes)) to invert the transposition of tensors when using the axes keyword argument.
    # Example: x = np.ones((1, 2, 3))
    # np.transpose(x, (1, 0, 2)).shape --> (2, 1, 3)
    state = observation.transpose((2,0,1))
    state = torch.from_numpy(state)
    state = transform(state)
    return state
```

```
/usr/local/lib/python3.7/dist-packages/torchvision/transforms/transforms.py:333: UserWarning: Argument 'interpolation' of type ir
"Argument 'interpolation' of type int is deprecated since 0.13 and will be removed in 0.15. "
```

```
model = DQN(image_size * image_size, env.action_space.n).to(device)
```

```
optimizer = optim.Adam(model.parameters())
```

```
replay_buffer = ReplayBuffer(1000)
```

```
model
```

```
DQN(
  (layers): Sequential(
    (0): Linear(in_features=7056, out_features=128, bias=True)
    (1): ReLU()
    (2): Linear(in_features=128, out_features=128, bias=True)
    (3): ReLU()
    (4): Linear(in_features=128, out_features=6, bias=True)
  )
)
```

```
def train2(env, model, eps_by_episode, optimizer, replay_buffer, episodes = 100, batch_size=32, gamma = 0.99):
    losses = []
    all_rewards = []
    episode_reward = 0

    obs = env.reset()
    # convert observation state to image state, and reshape as 1D size
    state = get_state(obs).view(image_size * image_size)
    #####
    tot_reward = 0
    tr = trange(episodes+1, desc='Agent training', leave=True)
    for episode in tr:
        tr.set_description("Agent training (episode{}) Avg Reward {}".format(episode+1,tot_reward/(episode+1)))
        tr.refresh()

        # get action with q-values
        epsilon = eps_by_episode(episode)
        action = model.act(state, epsilon)

        # input action into state
```

```

next_obs, reward, done, _ = env.step(action)
# convert observation state to image state, and reshape as 1D size
next_state = get_state(next_obs).view(image_size * image_size)
#####
# save data into buffer
replay_buffer.push(state, action, reward, next_state, done)

tot_reward += reward

state = next_state
obs = next_obs
episode_reward += reward

if done:
    obs = env.reset()
    # convert observation state to image state, and reshape as 1D size
    state = get_state(obs).view(image_size * image_size)
    #####
    all_rewards.append(episode_reward)
    episode_reward = 0

if len(replay_buffer) > batch_size:
    loss = compute_td_loss(model, batch_size, gamma)
    losses.append(loss.item())

plot(episode, all_rewards, losses)
return model, all_rewards, losses

```

```

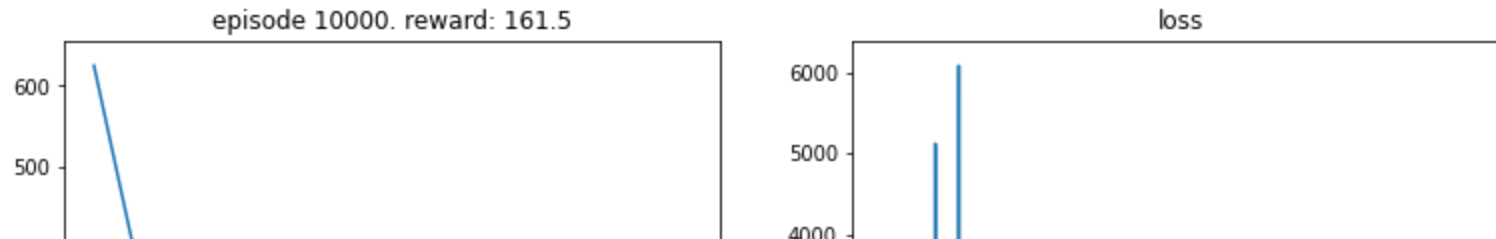
model, all_rewards, losses = train2(env, model, eps_by_episode, optimizer, replay_buffer, episodes = 10000, batch_size=32, gamma = 0.9

```


Agent training (episode1) Avg Reward 0.0: 0% | 0/10001 [00:00<?, ?it/s]/usr/local/lib/python3.7/dist-packages/ipykerr

Agent training (episode31) Avg Reward 0.0: 0% | 30/10001 [00:01<05:47, 28.67it/s]/usr/local/lib/python3.7/dist-packag

Agent training (episode10001) Avg Reward 0.27047295270472954: 100% | ██████████ | 10001/10001 [05:54<00:00, 28.18it/s]



```
def play_game2(model):
    done = False
    obs = env.reset()
    # convert observation state to image state, and reshape as 1D size
    state = get_state(obs).view(image_size * image_size)
    #####
    while(not done):
        action = model.act(state, epsilon_final)
        next_obs, reward, done, _ = env.step(action)
        # convert observation state to image state, and reshape as 1D size
        next_state = get_state(next_obs).view(image_size * image_size)
        #####
        screen = env.render(mode='rgb_array')
        plt.imshow(screen)

        #ipythondisplay.clear_output(wait=True)
        #ipythondisplay.display(plt.gcf())
        state = next_state
```

```
!pip install IPython
```

Looking in indexes: <https://pypi.org/simple>, <https://us-python.pkg.dev/colab-wheels/public/simple/>

Requirement already satisfied: IPython in /usr/local/lib/python3.7/dist-packages (7.9.0)

Requirement already satisfied: decorator in /usr/local/lib/python3.7/dist-packages (from IPython) (4.4.2)

Requirement already satisfied: prompt-toolkit<2.1.0,>=2.0.0 in /usr/local/lib/python3.7/dist-packages (from IPython) (2.0.10)

Requirement already satisfied: pygments in /usr/local/lib/python3.7/dist-packages (from IPython) (2.6.1)

Requirement already satisfied: jedi>=0.10 in /usr/local/lib/python3.7/dist-packages (from IPython) (0.18.2)

Requirement already satisfied: pexpect in /usr/local/lib/python3.7/dist-packages (from IPython) (4.8.0)

Requirement already satisfied: traitlets>=4.2 in /usr/local/lib/python3.7/dist-packages (from IPython) (5.1.1)

Requirement already satisfied: backcall in /usr/local/lib/python3.7/dist-packages (from IPython) (0.2.0)

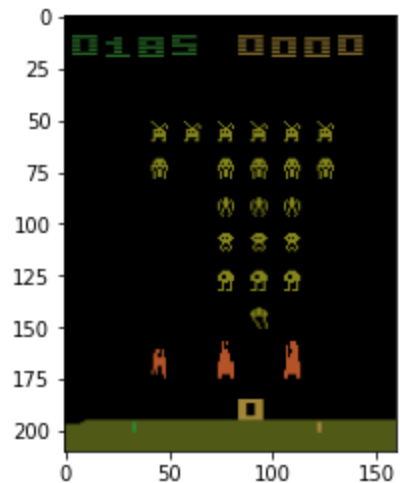
Requirement already satisfied: pickleshare in /usr/local/lib/python3.7/dist-packages (from IPython) (0.7.5)

Requirement already satisfied: setuptools>=18.5 in /usr/local/lib/python3.7/dist-packages (from IPython) (57.4.0)

```
Requirement already satisfied: parso<0.9.0,>=0.8.0 in /usr/local/lib/python3.7/dist-packages (from jedi>=0.10->IPython) (0.8.3)
Requirement already satisfied: wcwidth in /usr/local/lib/python3.7/dist-packages (from prompt-toolkit<2.1.0,>=2.0.0->IPython) (0.
Requirement already satisfied: six>=1.9.0 in /usr/local/lib/python3.7/dist-packages (from prompt-toolkit<2.1.0,>=2.0.0->IPython)
Requirement already satisfied: ptyprocess>=0.5 in /usr/local/lib/python3.7/dist-packages (from pexpect->IPython) (0.7.0)
```

```
from IPython.display import display
play_game2(model)
#ipythondisplay.clear_output(wait=True)
env.close()
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:19: UserWarning: volatile was removed and now has no effect. Use `wi
```



▼ Take home exercise (50 points)

```
env_id = 'SpaceInvaders-v0'
env = gym.make(env_id)

print(env.unwrapped.get_action_meanings())
print(env.observation_space.shape)

model = DQN(image_size * image_size, env.action_space.n).to(device)

optimizer = optim.Adam(model.parameters())

replay_buffer = ReplayBuffer(1000)
```

```
model, all_rewards, losses = train2(env, model, eps_by_episode, optimizer, replay_buffer, episodes = 20000, batch_size=32, gamma = 0.9
```

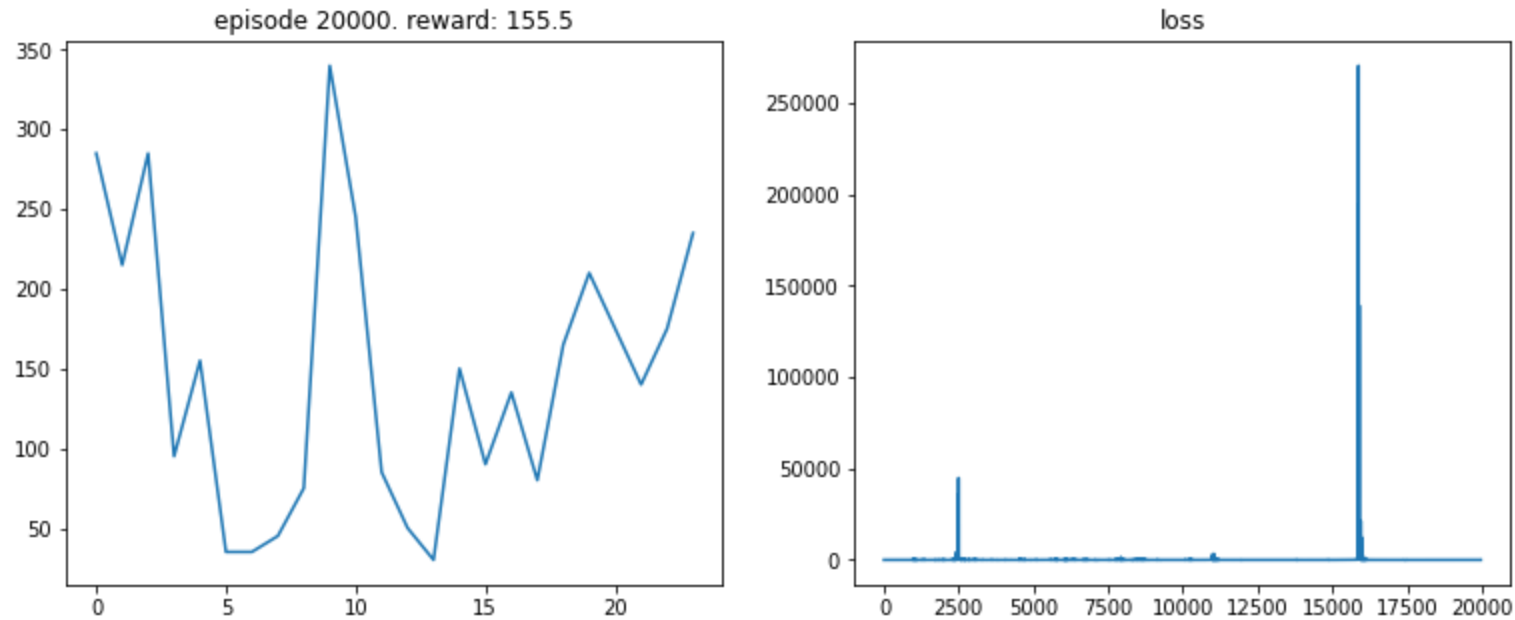
```
['NOOP', 'FIRE', 'RIGHT', 'LEFT', 'RIGHTFIRE', 'LEFTFIRE']
```

```
(210, 160, 3)
```

```
Agent training (episode29) Avg Reward 0.0: 0%| | 27/20001 [00:00<04:37, 71.92it/s]/usr/local/lib/python3.7/dist-packag
```

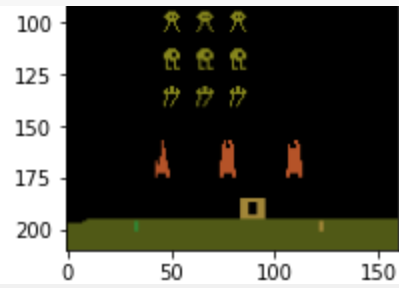
```
Agent training (episode33) Avg Reward 0.0: 0%| | 27/20001 [00:00<04:37, 71.92it/s]/usr/local/lib/python3.7/dist-packag
```

```
Agent training (episode20001) Avg Reward 0.17649117544122794: 100%|██████████| 20001/20001 [11:49<00:00, 28.19it/s]
```



```
play_game2(model)
#ipythondisplay.clear_output(wait=True)
env.close()
```

/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:19: UserWarning: volatile was removed and now has no effect. Use `wi



[Colab paid products](#) - [Cancel contracts here](#)

✓ 0s completed at 10:58 AM

