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 :-)

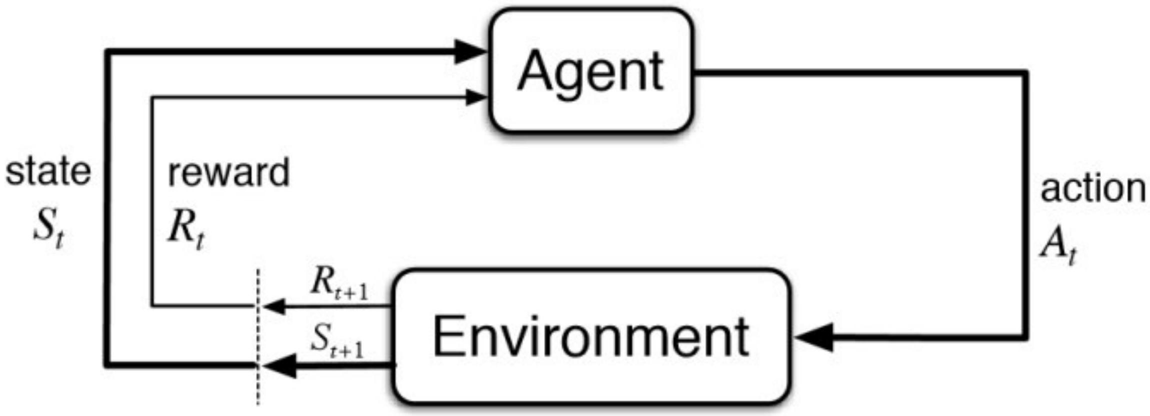
Today we'll have a gentle introduction to reinforcement learning. The material in today's lab comes from these references:

- 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 (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 (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.



Markov decision process (MDP)

A MDP is a discrete-time stochastic control process. The MDP model is based on the idea of an environment that evolves as a Markov chain.

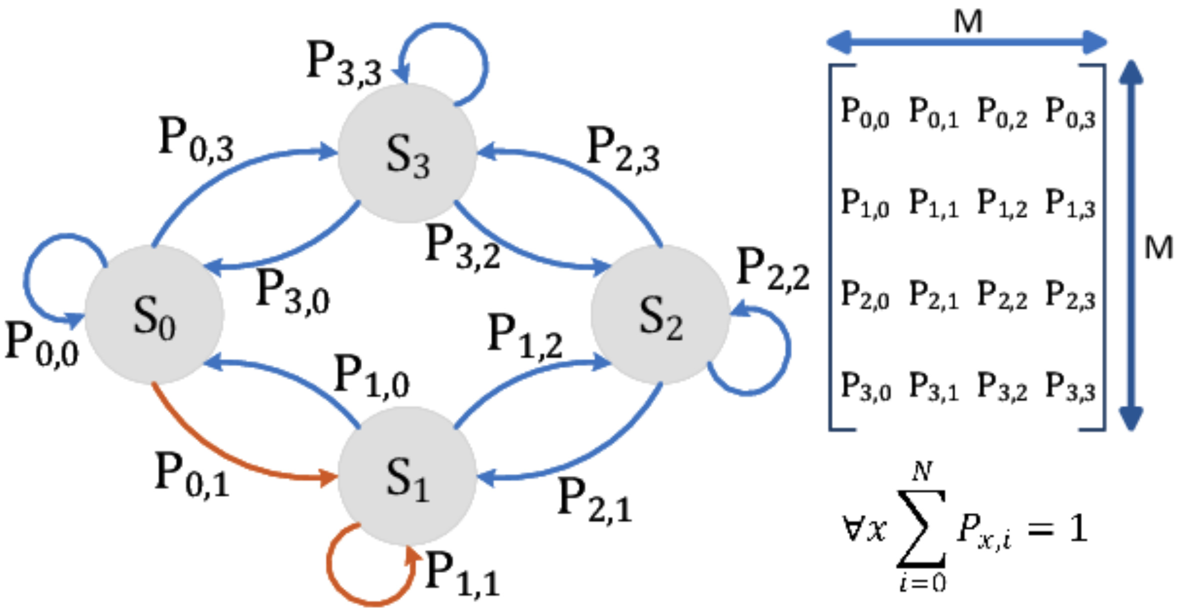
Markov chain

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 $0, \ldots, t-1$ given the state at time t, i.e.,

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

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

More precisely, a Markov chain is defined by a set of possible states $S = s_0, s_1, \ldots, s_n$ and a transition matrix $T(s, s')$ containing the propbabilities of state s transitioning to state s'. Here is a visualization of a simple Markov chain:



You might be interested in [this Markov chain simulator (https://setosa.io/ev/markov-chains/)](https://setosa.io/ev/markov-chains/).

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

Suppose we have three states and two actions and that the state/action transition tensor is as follows:

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

To complete our simple MDP, we need a *reward function* R and a *discount factor* γ. Suppose $R = [1, 0, -1]$ and $\gamma = 0.5$. Let's define our MDP in Python with PyTorch tensors:

In [1]:
```
import torch

# 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]]])

# Reward function

R = torch.tensor([1.,0.,-1.])

# 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 $a^0, a^1, \ldots a^{T-1}$ placing us in state $s^1, s^2, \ldots s^T$, we obtain the total reward

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

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

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

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*

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

How good is a policy? 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.

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

Here we compute the value of the three states in our MDP assuming the agent always peforms the first action:

In [2]:
```
def policy_evaluation(policy, trans_matrix, rewards, gamma, thres
hold):
    n_state = policy.shape[0]
    V = torch.zeros(n_state)
    while True:
        V_temp = torch.zeros(n_state)
        for state, actions in enumerate(policy):
            for action, action_prob in enumerate(actions):
                V_temp[state] += action_prob * (rewards[state] +
gamma * torch.dot(trans_matrix[state, action], V))
        max_delta = torch.max(torch.abs(V-V_temp))
        V = V_temp.clone()
        if max_delta <= threshold:
            break
    return V

threshold = 0.0001
policy_optimal = torch.tensor([[1.0, 0.0],
                               [1.0, 0.0],
                               [1.0, 0.0]])
V = policy_evaluation(policy_optimal, T, R, gamma, threshold)
print(V)
```

```
tensor([ 1.6786,  0.6260, -0.4821])
```

Policy iteration

Policy iteration starts with a random policy then uses policy evaluation and the resulting values to iteratively improve the policy until an optimal policy is obtained. It is, however, slow, due to the policy evaluation loop within the policy iteration loop.

Here's an implementation using a slightly different formulation fo the reward as a function $R(s, a, s')$ of the current state s, action taken a, and resulting state s':

In [5]:
```
def policy_evaluation(policy, trans_matrix, rewards, gamma, thres
hold):
    n_state = policy.shape[0]
    V = torch.zeros(n_state)
    while True:
        V_temp = torch.zeros(n_state)
        for state in range(n_state):
            action = int(policy[state].item())
            for new_state in range(n_state):
                trans_prop = trans_matrix[state, action, new_stat
e]
                reward = rewards[state, action, new_state]
                V_temp[state] += trans_prop * (reward + gamma * V
[new_state])
        max_delta = torch.max(torch.abs(V-V_temp))
        V = V_temp.clone()
        if max_delta <= threshold:
            break
    return V

def policy_improvement(trans_matrix, rewards, gamma):
    n_state = trans_matrix.shape[0]
    n_action = trans_matrix.shape[1]
    policy = torch.zeros(n_state)

    for state in range(n_state):
        v_actions = torch.zeros(n_action)
        for action in range(n_action):
            for new_state in range(n_state):
                trans_prop = trans_matrix[state, action, new_stat
e]
                reward = rewards[state, action, new_state]
                v_actions[action] += trans_prop * (reward + gamma
* V[new_state])
        policy[state] = torch.argmax(v_actions)
    return policy

def policy_iteration(trans_matrix, rewards, gamma, threshold):
    n_state = trans_matrix.shape[0]
    n_action = trans_matrix.shape[1]
    policy = torch.randint(high=n_action, size=(n_state,)).float
()
    while True:
        V = policy_evaluation(policy, trans_matrix, rewards, gamm
a, threshold)
        policy_improved = policy_improvement(trans_matrix, reward
s, gamma)
        if torch.equal(policy_improved, policy):
            return V, policy_improved
        policy = policy_improved

# Reward R(s,a,a') example

R2 = torch.tensor([[[0.1,0.,-0.2],
                   [0.2,0.,-0.1]],
                  [[0.3,0.,-0.5],
                   [0.1,0.,-0.2]],
```

```
                  [[0.2,0.,-0.1],
                   [1.,0.,-1.]]]])

V_optimal, optimal_policy = policy_iteration(T, R2, gamma, thresh
old)
print(V_optimal)
print(optimal_policy)
```

```
tensor([0.1352, 0.2405, 0.1829])
tensor([0., 0., 0.])
```

Value iteration

Value iteration is a much more efficient way to obtain the optimial policy. It calculates the value of each state on the assumption that the agent will deterministically select the action a in state s that maximizes the expected reward after that. Once this value converges, the optimal policy is just to select the best action according to that value function:

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

$$\pi^*(s) = \mathrm{argmax}_a \sum_{s'} T(s,a,s') V^*(s').$$

Instead of taking the expectation (average) of values across all actions, we pick the action that achieves the maximal policy values. As above, we use the slightly different formulation in which the reward $R(s,a,s')$ is given based on initial state, action, and resulting state:

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

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

Here's sample code:

In [6]:
```
def value_iteration(trans_matrix, rewards, gamma, threshold):
    n_state = trans_matrix.shape[0]
    n_action = trans_matrix.shape[1]
    V = torch.zeros(n_state)
    while True:
        V_temp = torch.zeros(n_state)
        for state in range(n_state):
            v_actions = torch.zeros(n_action)
            for action in range(n_action):
                for new_state in range(n_state):
                    trans_prop = trans_matrix[state, action, new_
state]
                    reward = rewards[state, action, new_state]
                    v_actions[action] += trans_prop * (reward + g
amma * V[new_state])
            V_temp[state] = torch.max(v_actions)
        max_delta = torch.max(torch.abs(V-V_temp))
        V = V_temp.clone()
        if max_delta <= threshold:
            break
    return V

V = value_iteration(T, R2, gamma, threshold)
print(V)
```

```
tensor([0.1352, 0.2405, 0.1829])
```

Exercise

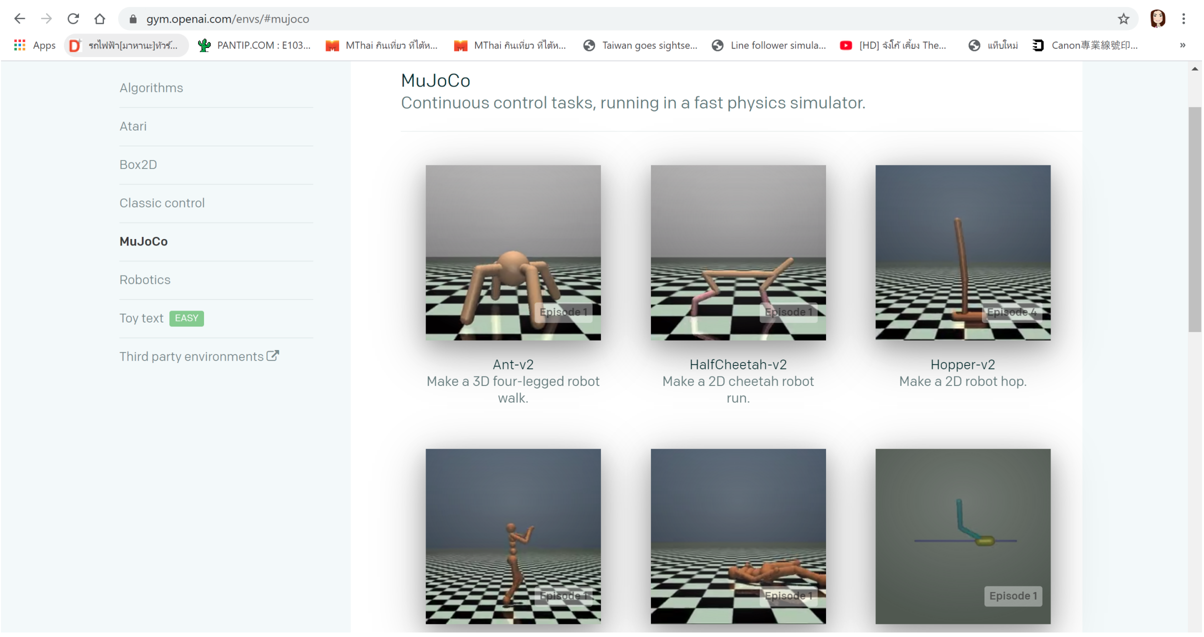
Write some code now to find the optimal policy based on the values you got from value iteration above. Do you get the same result as for policy iteration?

OpenAI Gym

Looking at policies in matrix form is a bit difficult.

A popular simulation environment for RL is OpenAI Gym.

OpenAI (https://openai.com) 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 (https://gym.openai.com) for a full list of environments.



Installing Gym

Install Gym:

In [7]:
```
!pip install gym
# or
# !git clone https://github.com/openai/gym
# !cd gym
# pip install -a
```

```
Looking in indexes: https://pypi.org/simple, https://pypi.ngc.nvi
dia.com
Collecting gym
  Downloading gym-0.18.0.tar.gz (1.6 MB)
     |████████████████████████████████| 1.6 MB 1.9 MB/s eta 0:00:
01
Requirement already satisfied: scipy in /opt/conda/lib/python3.8/
site-packages (from gym) (1.4.1)
Requirement already satisfied: numpy>=1.10.4 in /opt/conda/lib/py
thon3.8/site-packages (from gym) (1.19.2)
Collecting pyglet<=1.5.0,>=1.4.0
  Downloading pyglet-1.5.0-py2.py3-none-any.whl (1.0 MB)
     |████████████████████████████████| 1.0 MB 5.4 MB/s eta 0:00:
01
Collecting Pillow<=7.2.0
  Downloading Pillow-7.2.0-cp38-cp38-manylinux1_x86_64.whl (2.2 M
B)
     |████████████████████████████████| 2.2 MB 3.7 MB/s eta 0:00:
01
Requirement already satisfied: cloudpickle<1.7.0,>=1.2.0 in /opt/
conda/lib/python3.8/site-packages (from gym) (1.4.1)
Requirement already satisfied: future in /opt/conda/lib/python3.8
/site-packages (from pyglet<=1.5.0,>=1.4.0->gym) (0.18.2)
Building wheels for collected packages: gym
  Building wheel for gym (setup.py) ... done
  Created wheel for gym: filename=gym-0.18.0-py3-none-any.whl siz
e=1656446 sha256=fb2ae60177e83c5aa194a0aec2ae1758fa5fd083eb1c8af7
09a28fc816838ccd
  Stored in directory: /tmp/pip-ephem-wheel-cache-neht8alu/wheels
/d8/e7/68/a3f0f1b5831c9321d7523f6fd4e0d3f83f2705a1cbd5daaa79
Successfully built gym
Installing collected packages: pyglet, Pillow, gym
  Attempting uninstall: Pillow
    Found existing installation: Pillow 8.0.1
    Uninstalling Pillow-8.0.1:
      Successfully uninstalled Pillow-8.0.1
Successfully installed Pillow-7.2.0 gym-0.18.0 pyglet-1.5.0
```

To see visualizations easily, you should run the simulator locally, not in Jupyter.

Atari games environment

Atari games includes video games such as Alien, Pong, and Space Race. Here is the example of using the Space Invaders game. First, install the atari simulation environment:

In [8]:
```
!pip install gym[atari]
```

```
Looking in indexes: https://pypi.org/simple, https://pypi.ngc.nvi
dia.com
Requirement already satisfied: gym[atari] in /opt/conda/lib/pytho
n3.8/site-packages (0.18.0)
Requirement already satisfied: cloudpickle<1.7.0,>=1.2.0 in /opt/
conda/lib/python3.8/site-packages (from gym[atari]) (1.4.1)
Requirement already satisfied: numpy>=1.10.4 in /opt/conda/lib/py
thon3.8/site-packages (from gym[atari]) (1.19.2)
Requirement already satisfied: scipy in /opt/conda/lib/python3.8/
site-packages (from gym[atari]) (1.4.1)
Requirement already satisfied: Pillow<=7.2.0 in /opt/conda/lib/py
thon3.8/site-packages (from gym[atari]) (7.2.0)
Requirement already satisfied: pyglet<=1.5.0,>=1.4.0 in /opt/cond
a/lib/python3.8/site-packages (from gym[atari]) (1.5.0)
Requirement already satisfied: opencv-python>=3.; extra == "atar
i" in /opt/conda/lib/python3.8/site-packages (from gym[atari])
(4.4.0.46)
Collecting atari-py~=0.2.0; extra == "atari"
  Downloading atari-py-0.2.6.tar.gz (790 kB)
     |████████████████████████████████| 790 kB 1.2 MB/s eta 0:00:
01
Requirement already satisfied: future in /opt/conda/lib/python3.8
/site-packages (from pyglet<=1.5.0,>=1.4.0->gym[atari]) (0.18.2)
Requirement already satisfied: six in /opt/conda/lib/python3.8/si
te-packages (from atari-py~=0.2.0; extra == "atari"->gym[atari])
(1.15.0)
Building wheels for collected packages: atari-py
  Building wheel for atari-py (setup.py) ... done
  Created wheel for atari-py: filename=atari_py-0.2.6-cp38-cp38-l
inux_x86_64.whl size=2812183 sha256=fe160ffca5c1ff7901860c96a1914
be06d10d9076ef2d7cf8d8ac7bdf38d143a
  Stored in directory: /tmp/pip-ephem-wheel-cache-4s0peg1q/wheels
/7f/5e/27/2e90b9887063d82ee2f9f8b2f8db76bb2290aa281dc40449c8
Successfully built atari-py
Installing collected packages: atari-py
Successfully installed atari-py-0.2.6
```

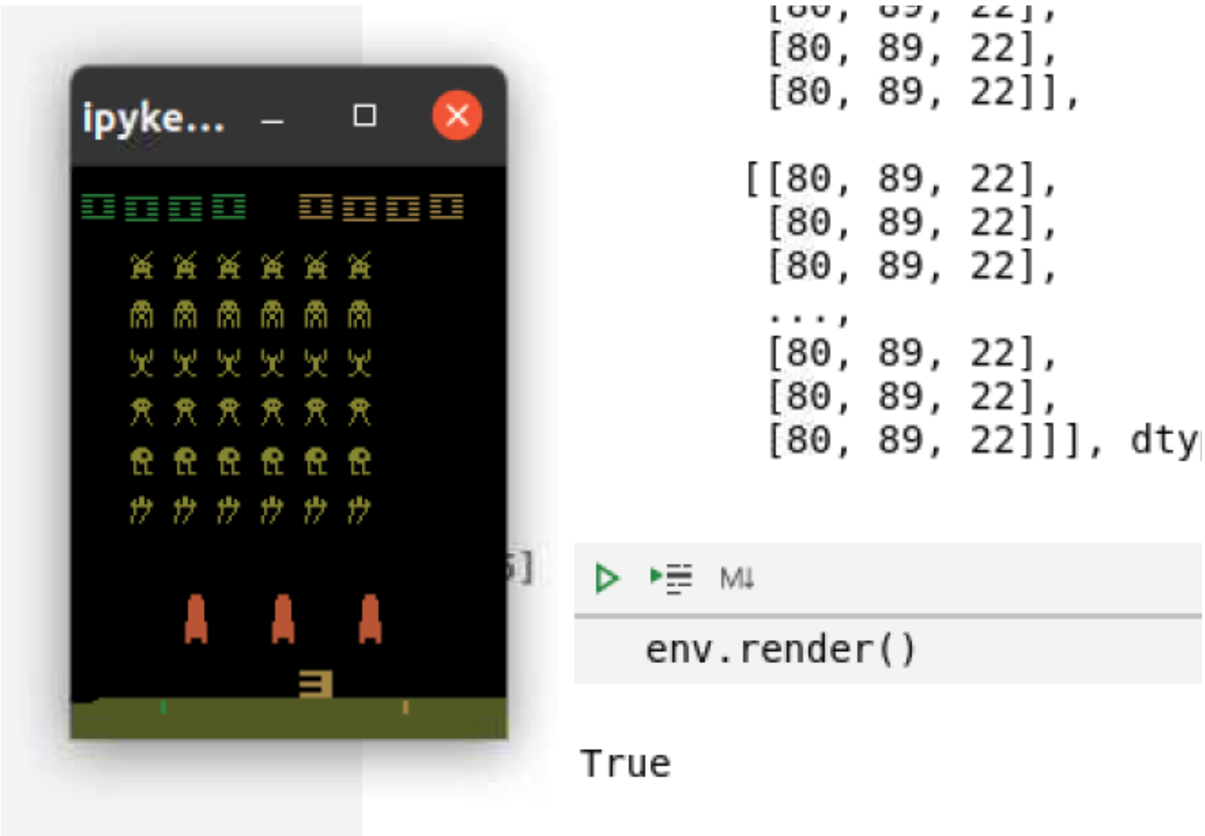
Then create a Space Invaders environment:

In [11]:
```
import gym
# create environment
env = gym.make('SpaceInvaders-v0')

# reset environments
env.reset()
```

Out[11]:
```
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)
```

Render the environment locally. (You cannot render in Jupyter, or let us know if you find a way!)

Locally, you should get output like this:



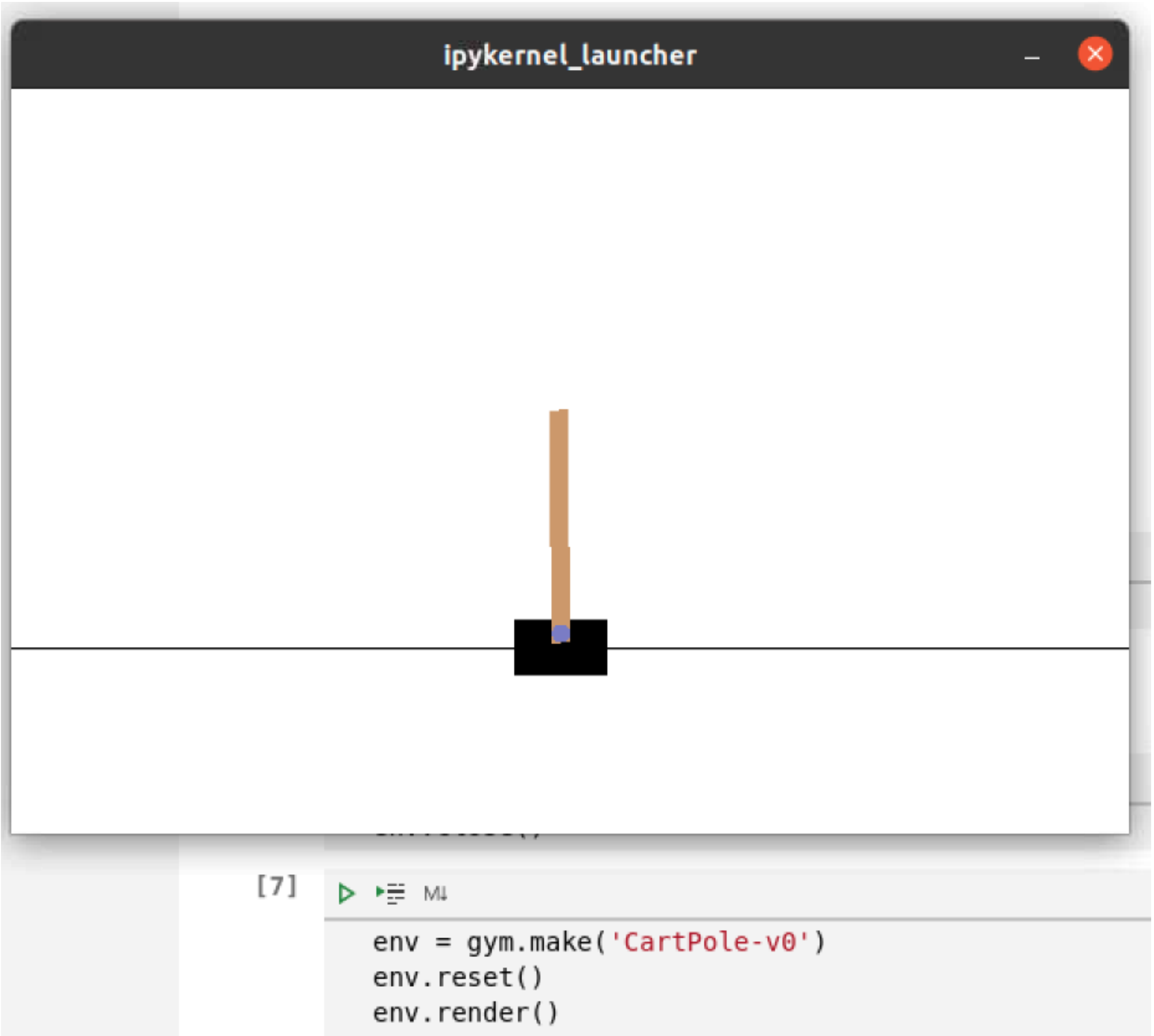
After rendering, you'll want to close the environment.

WARNING

In JUPYTER NOTEBOOK or JUPYTER LAB, you need to use `env.close()` after `env.render()` , unless the program will stuck.

In [ ]:
```
# Example 1

# import time
# env.render()
# time.sleep(5)
#close the environment
# env.close()
```

If you cannot use Atari games due to errors occurring in Windows, you can try another simulator such as CartPole (inverted pendulum):




```
In [ ]: # Example 2

        #import gym
        #import time
        # create environment
        #env = gym.make('CartPole-v0')

        # reset environments
        #env.reset()

        # render the environment
        #env.render()
        #time.sleep(5)

        #env.close()
```

Before try the example above, I have another solution that you can use gym run in Jupyter notebook. One is MP4 and another is real-time showing. [In this link](https://kyso.io/eoin/openai-gym-jupyter#code=both)

Way1: MP4 Running

Save the simulation as an mp4 and save it at the end. This method makes the simulation faster since there is no visualising happening every step. In this simulation we just make the space ship take a random move each step.

In [2]:
```
import gym
from gym import wrappers

env = gym.make('SpaceInvaders-v0')
env = wrappers.Monitor(env, "./gym-results", force=True)
env.reset()
for _ in range(1000):
    action = env.action_space.sample()
    observation, reward, done, info = env.step(action)
    if done: break
env.close()
```

And show MP4 vdo by this code

In [3]:
```
import io
import base64
from IPython.display import HTML

video = io.open('./gym-results/openaigym.video.%s.video000000.mp4
' % env.file_infix, 'r+b').read()
encoded = base64.b64encode(video)
HTML(data='''
    <video width="360" height="auto" alt="test" controls><source
src="data:video/mp4;base64,{0}" type="video/mp4" /></video>'''
.format(encoded.decode('ascii')))
```

Out[3]:

No video with supported format and MIME type found.

Way2: Show as gif in realtime

This method is useful for look at the simulation in realtime, but it does make the simulation take longer. It does not save the gif, but if you run this cell you will the image change as the simulation progresses.

In [7]:
```
import gym
from IPython import display
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline

env = gym.make('CarRacing-v0')
env.reset()
plt.figure(figsize=(9,9))
#img = plt.imshow(env.render(mode='rgb_array')) # only call this once
for _ in range(100):
    #img.set_data(env.render(mode='rgb_array')) # just update the data
    #display.display(plt.gcf())
    #display.clear_output(wait=True)
    env.render()
    action = env.action_space.sample()
    env.step(action)
env.close()
```

```
/home/alisa/.local/lib/python3.8/site-packages/gym/logger.py:30:
UserWarning: WARN: Box bound precision lowered by casting to floa
t32
  warnings.warn(colorize('%s: %s'%('WARN', msg % args), 'yellow
'))

Track generation: 1161..1460 -> 299-tiles track

<Figure size 648x648 with 0 Axes>
```

In this lab, I recommend you to use the way2 for smoother working ;-)

Take a look at the action space available, and try to get a sample from the action space:

In [5]:
```
print(env.action_space)
action = env.action_space.sample()
print(action)
```

```
Discrete(6)
4
```

Execute an action using `step()` . The `step` method returns the next state after the action is taken.

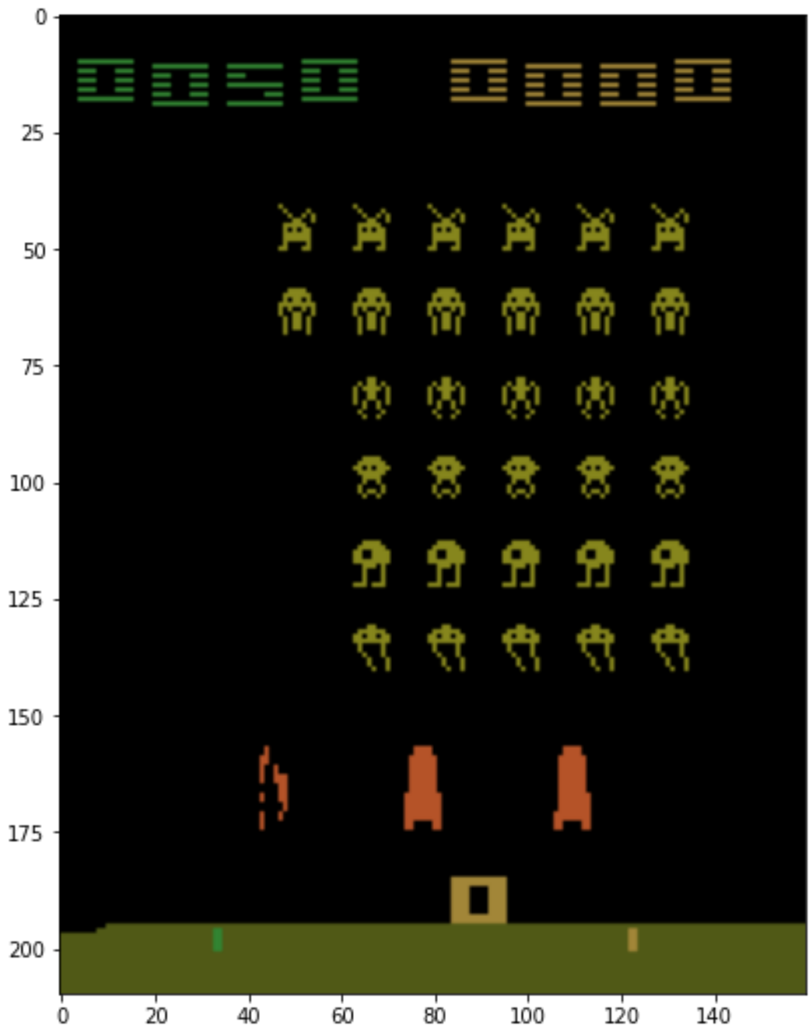
- **new_state**: The new observation
- **reward**: The reward associated with that action in that state.
- **is_done**: A flag to tell the game end (True).
- **info**: extra information

In [6]:
```
new_state, reward, is_done, info = env.step(action)
print(is_done)
print(reward)
print(info)
```

```
False
0.0
{'ale.lives': 3}
```

Let's make a *while* loop with a random agent:

In [7]:
```
import time

is_done = False
env = gym.make('SpaceInvaders-v0')
env.reset()
plt.figure(figsize=(9,9))
img = plt.imshow(env.render(mode='rgb_array')) # only call this once
while not is_done:
    action = env.action_space.sample()
    new_state, reward, is_done, info = env.step(action)
    print(info)
    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)
env.close()
```



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.

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: $\pi(s) = argmax_a Q(s,a)$

In this code below, we add **epsilon-greedy** policy which it will not exploit the best action all the time. The equations are:

- Epsilon (ϵ):

  $\pi(s,a) = \frac{\epsilon}{|A|}$ When $|A|$ is the number of all possible actions.
- Greedy:

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

In [4]:
```
import torch
import gym
from collections import defaultdict

env = gym.make('Blackjack-v0')

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, epsi
lon, n_action)
        return_t = 0
        G = {}
        for state_t, action_t, reward_t in zip(states_t[::-1], ac
tions_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[st
ate_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_ep
isode, 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  46  lose  45
```

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 importand 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.

In [13]:
```
import torch
import gym
from collections import defaultdict

env = gym.make('Blackjack-v0')

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_polic
y):
    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, behavio
r_policy)
        return_t = 0
        G = {}
        for state_t, action_t, reward_t in zip(states_t[::-1], ac
tions_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_e
pisode, 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  41  lose  51
```

Temporal Difference (TD)

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

Q-Learning

Q-leaning is an off-policy learning algorithm. The Q-function is based on the equation:

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

Where

- α: learning rate
- γ: discount factor
- $\max_{a'} Q(s',a')$: greedy behavior policy, the highest Q-value among those in state s' is selected to generate learning data.

We know from class that Q-learning finds the optimal greedy policy while running an epsilon-greedy policy.

In [24]:
```
import importlib
from collections import defaultdict
import torch
import numpy

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

env.reset()

# defining epsilon-greedy policy
def gen_epsilon_greedy_policy(n_action, epsilon):
    def policy_function(state, Q, available_actions):
        probs = torch.ones(n_action) * epsilon / n_action
        # print(probs)
        # print(state)
        # print(Q[state])
        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[stat
e][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

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_ac
tions)
            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
```

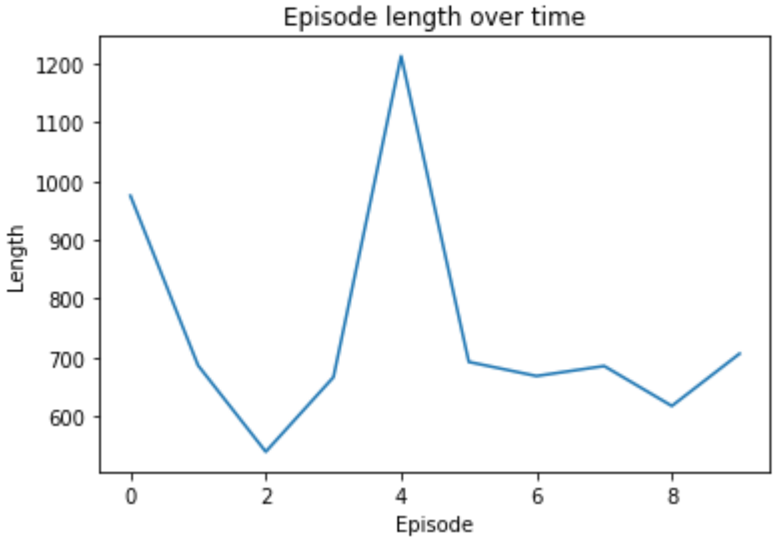
```
start learning
episode:  1
episode:  2
episode:  3
episode:  4
episode:  5
episode:  6
episode:  7
episode:  8
episode:  9
episode:  10
```

Train it!

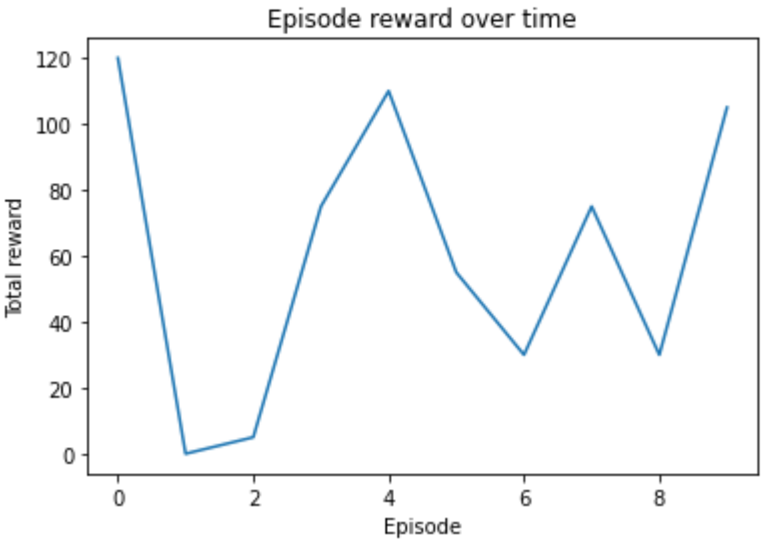
In [ ]:

```
gamma = 1

# the more episodes have been learned, the cleverer will be get.
n_episode = 10

alpha = 0.4

epsilon = 0.1

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

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

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


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

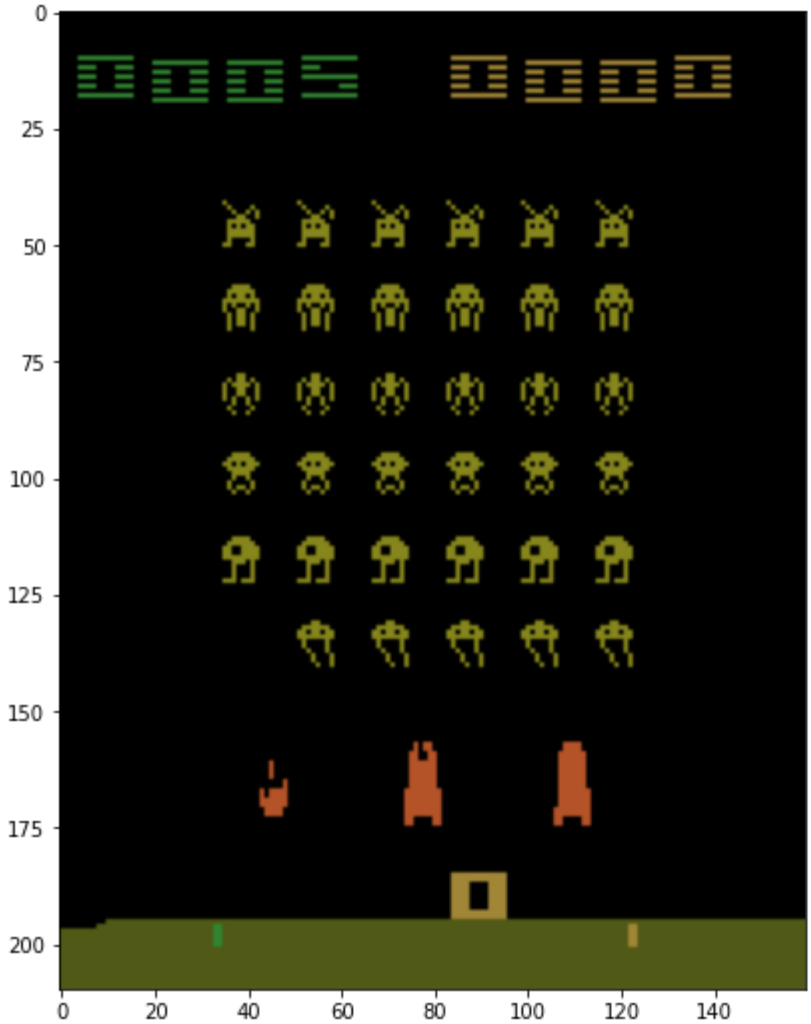
In [21]:
```
import matplotlib.pyplot as plt

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, 0.0, 5.0, 75.0, 110.0, 55.0, 30.0, 75.0, 30.0, 105.0]
```



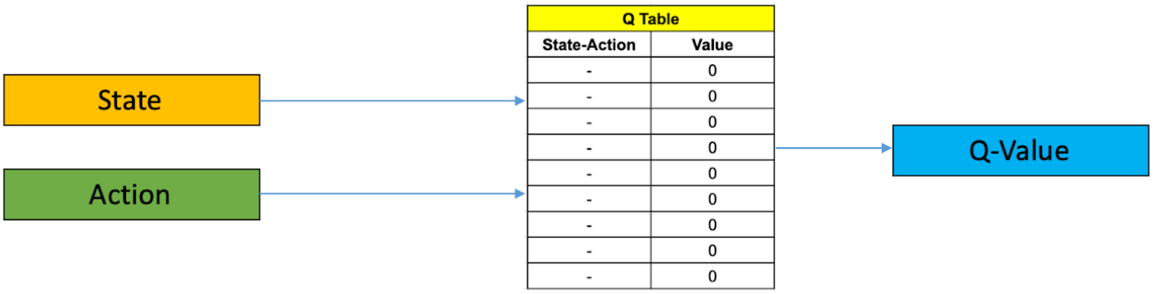
In [25]:
```
import time
from IPython import display
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline

def play_game(Q, available_actions):
    done = False
    plt.figure(figsize=(9,9))
    img = plt.imshow(env.render(mode='rgb_array')) # only call th
is once
    state = env.reset()
    state = hash(tuple(state.reshape(-1)))
    while(not done):
        action = epsilon_greedy_policy(state, Q, available_action
s)
        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()
```


0
25
50
75
100
125
150
175
200
0
20
40
60
80
100
120
140

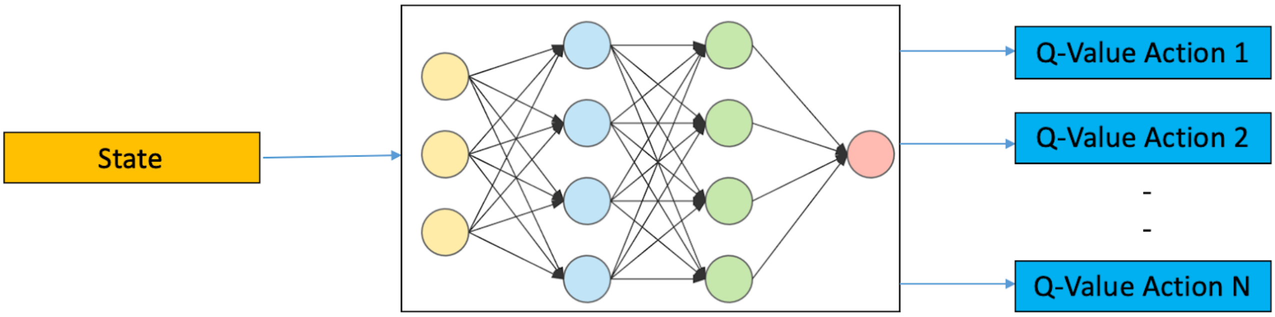
Deep Q-Learning

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.



Q Learning



Deep Q Learning

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 appopriate, giving us a delta for the ath 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. We'll use the `gym`

In [ ]:
```
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

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:

In [ ]:
```
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 explicit
ly defining a function:
# def epsilon_episode(episode):
#     return epsilon_final + (epsilon_start - epsilon_final) * ma
th.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:

In [ ]:
```
# Epsilon annealing schedule generator

def gen_eps_by_episode(epsilon_start, epsilon_final, epsilon_deca
y):
    eps_by_episode = lambda episode: epsilon_final + (epsilon_sta
rt - 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 samaples.

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:

In [ ]:
```
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, do
ne))

    def sample(self, batch_size):
        state, action, reward, next_state, done = zip(*random.sam
ple(self.buffer, batch_size))
        return np.concatenate(state), action, reward, np.concaten
ate(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 indiciating the value of each possible action $Q(s, a)$.

The steps we take during learning will be as follows:

1. take some action $\mathbf{a}_i$ and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$
2. $\mathbf{y}_i = r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$
3. $\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i)$

Explore with epsilon-greedy policy

$$\pi(\mathbf{a}_t|\mathbf{a}_t) = \begin{cases} 1 - \epsilon \text{ if } \mathbf{a}_t = \arg\max_{\mathbf{a}_t} Q_\phi(\mathbf{s}_t, \mathbf{a}_t) \\ \epsilon/(|\mathcal{A}| - 1) \text{ otherwise} \end{cases}$$

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.

In [ ]:
```
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).un
squeeze(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.

In [ ]:
```
env_id = "CartPole-v0"
env = gym.make(env_id)

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

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

replay_buffer = ReplayBuffer(1000)
```

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.

In [ ]:
```
def compute_td_loss(model, batch_size, gamma=0.99):

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

    # Convert to tensors. Creating Variables is not necessary wit
h more recent PyTorch versions.
    state      = autograd.Variable(torch.FloatTensor(np.float32(s
tate))).to(device)
    next_state = autograd.Variable(torch.FloatTensor(np.float32(n
ext_state)), volatile=True).to(device)
    action     = autograd.Variable(torch.LongTensor(action)).to(d
evice)
    reward     = autograd.Variable(torch.FloatTensor(reward)).to
(device)
    done       = autograd.Variable(torch.FloatTensor(done)).to(de
vice)

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

    # Get Q(s,a) and max_a' Q(s',a')
    q_value          = q_values.gather(1, action.unsqueeze(1)).sq
ueeze(1)
    next_q_value     = next_q_values.max(1)[0]
    # 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 PyTo
rch versions.
    loss = (q_value - autograd.Variable(expected_q_value.data)).p
ow(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:

In [ ]:
```
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(reward
s[-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.

In [ ]:
```
def train(env, model, eps_by_episode, optimizer, replay_buffer, e
pisodes = 10000, 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, don
e)

        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
        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:

In [ ]:
```
model,all_rewards, losses = train(env, model, eps_by_episode, opt
imizer, replay_buffer, episodes = 10000, batch_size=32, gamma =
0.99)
```

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).

In [ ]:
```
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 th
is 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)

Dr. Matt, please design

In [ ]:
```
```

Take home exercise (50 points)

Dr. Matt, please design

In [ ]:
```
```