IERG 5350 Assignment 3: Value Function Approximation in RL

2020-2021 Term 1, IERG 5350: Reinforcement Learning. Department of Information Engineering, The Chinese University of Hong Kong. Course Instructor: Professor ZHOU Bolei. Assignment author: PENG Zhenghao, SUN Hao, ZHAN Xiaohang.

Student Name	Student ID
BAI Fan	1155149235

Welecome to the assignment 3 of our RL course.

You need to go through this self-contained notebook, which contains many TODOs in part of the cells and has special `[TODO]` signs. You need to finish all TODOs. Some of them may be easy such as uncommenting a line, some of them may be difficult such as implementing a function. You can find them by searching the `[TODO]` symbol. However, we suggest you to go through the notebook step by step, which would give you a better understanding of the content.

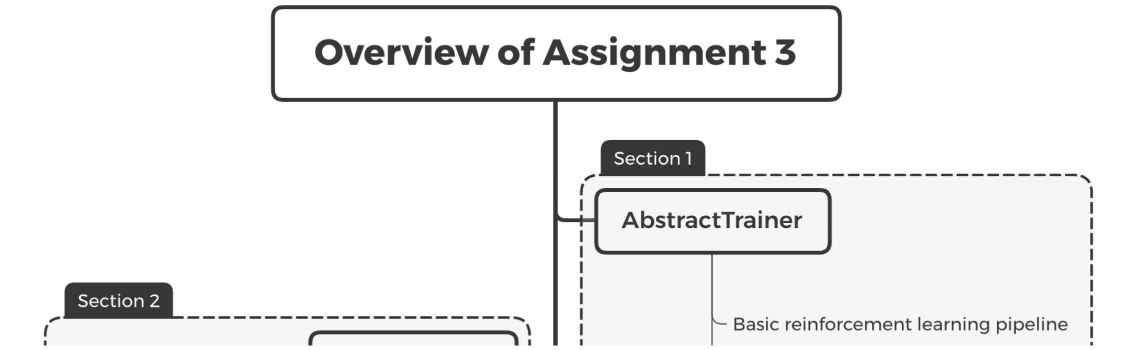
You are encouraged to add more code on extra cells at the end of the each section to investigate the problems you think interesting. At the end of the file, we left a place for you to optionaly write comments (Yes, please give us rewards so we can keep improving the assignment!).

Please report any code bugs to us via github issue.

We will cover the following knowledege in this assignment:

1. The n-step TD control algorithm
2. Linear function as value approximator
3. Feature construction
4. Neural network based function approximation
5. The basic usage of Pytorch

In the first section of notebook, we build a basic RL pipeline. In the second section, we implement the linear function as approximator and also introduces feature construction technique. In the third section, we implement a simple neural network simply using Numpy package.



Section 1: Basic Reinforcement Learning Pipeline

(5 / 100 points)

In this section, we will prepare several functions for evaulation, training RL algorithms. We will also build an `AbstractTrainer` class used as a general framework which left blanks for different function approximation methods.

In [1]:
```
import gym
import numpy as np
import torch
from utils import *
import torch
import torch.nn as nn
```

In [2]:
```
# Run this cell without modification

def evaluate(policy, num_episodes=1, seed=0, env_name='FrozenLake8x8-v0',
             render=False):
    """This function evaluate the given policy and return the mean episode
    reward.
    :param policy: a function whose input is the observation
    :param num_episodes: number of episodes you wish to run
    :param seed: the random seed
    :param env_name: the name of the environment
    :param render: a boolean flag indicating whether to render policy
    :return: the averaged episode reward of the given policy.
    """
    env = gym.make(env_name)
    env.seed(seed)
    rewards = []
    if render: num_episodes = 1
    for i in range(num_episodes):
        obs = env.reset()
        act = policy(obs)
        ep_reward = 0
        while True:
            obs, reward, done, info = env.step(act)
            act = policy(obs)
            ep_reward += reward
            if render:
                env.render()
                wait(sleep=0.05)
            if done:
                break
        rewards.append(ep_reward)
    if render:
        env.close()
    return np.mean(rewards)
```

In [3]:
```
# Run this cell without modification

def run(trainer_cls, config=None, reward_threshold=None):
    """Run the trainer and report progress, agnostic to the class of trainer
    :param trainer_cls: A trainer class
    :param config: A dict
    :param reward_threshold: the reward threshold to break the training
    :return: The trained trainer and a dataframe containing learning progress
    """
    assert inspect.isclass(trainer_cls)
    if config is None:
        config = {}
    trainer = trainer_cls(config)
    config = trainer.config
    start = now = time.time()
    stats = []
    for i in range(config['max_iteration'] + 1):
        stat = trainer.train()
        stats.append(stat or {})
        if i % config['evaluate_interval'] == 0 or \
                i == config["max_iteration"]:
            reward = trainer.evaluate(config.get("evaluate_num_episodes", 50))
            print("({:.1f}s,+{:.1f}s)\tIteration {}, current mean episode "
                  "reward is {}. {}".format(
                time.time() - start, time.time() - now, i, reward,
```

```
                {k: round(np.mean(v), 4) for k, v in
                 stat.items()} if stat else ""))
            now = time.time()
        if reward_threshold is not None and reward > reward_threshold:
            print("In {} iteration, current mean episode reward {:.3f} is "
                  "greater than reward threshold {}. Congratulation! Now we "
                  "exit the training process.".format(
                i, reward, reward_threshold))
            break
    return trainer, stats
```

In [4]:

```
# Solve TODOs and remove "pass"

default_config = dict(
    env_name="CartPole-v0",
    max_iteration=1000,
    max_episode_length=1000,
    evaluate_interval=100,
    gamma=0.99,
    eps=0.3,
    seed=0
)


class AbstractTrainer:
    """This is the abstract class for value-based RL trainer. We will inherent
    the specify algorithm's trainer from this abstract class, so that we can
    reuse the codes.
    """

    def __init__(self, config):
        self.config = merge_config(config, default_config)

        # Create the environment
        self.env_name = self.config['env_name']
        self.env = gym.make(self.env_name)
        if self.env_name == "Pong-ram-v0":
            self.env = wrap_deepmind_ram(self.env)

        # Apply the random seed
        self.seed = self.config["seed"]
        np.random.seed(self.seed)
        self.env.seed(self.seed)

        # We set self.obs_dim to the number of possible observation
        # if observation space is discrete, otherwise the number
        # of observation's dimensions. The same to self.act_dim.
        if isinstance(self.env.observation_space, gym.spaces.box.Box):
            assert len(self.env.observation_space.shape) == 1
            self.obs_dim = self.env.observation_space.shape[0]
            self.discrete_obs = False
        elif isinstance(self.env.observation_space,
                        gym.spaces.discrete.Discrete):
            self.obs_dim = self.env.observation_space.n
            self.discrete_obs = True
        else:
            raise ValueError("Wrong observation space!")

        if isinstance(self.env.action_space, gym.spaces.box.Box):
            assert len(self.env.action_space.shape) == 1
            self.act_dim = self.env.action_space.shape[0]
        elif isinstance(self.env.action_space, gym.spaces.discrete.Discrete):
            self.act_dim = self.env.action_space.n
        else:
            raise ValueError("Wrong action space!")

        self.eps = self.config['eps']

        # You need to setup the parameter for your function approximator.
        self.initialize_parameters()

    def initialize_parameters(self):
        self.parameters = None
        raise NotImplementedError(
            "You need to override the "
            "Trainer._initialize_parameters() function.")

    def process_state(self, state):
        """Preprocess the state (observation).

        If the environment provides discrete observation (state), transform
        it to one-hot form. For example, the environment FrozenLake-v0
        provides an integer in [0, ..., 15] denotes the 16 possible states.
        We transform it to one-hot style:

        original state 0 -> one-hot vector [1, 0, 0, 0, 0, 0, 0, 0, ...]
        original state 1 -> one-hot vector [0, 1, 0, 0, 0, 0, 0, 0, ...]
        original state 15 -> one-hot vector [0, ..., 0, 0, 0, 0, 0, 1]

        If the observation space is continuous, then you should do nothing.
        """
        if not self.discrete_obs:
            return state
        else:
            new_state = np.zeros((self.obs_dim,))
            new_state[state] = 1
        return new_state
```

```
    def compute_values(self, processed_state):
        """Approximate the state value of given state.
        This is a private function.
        Note that you should NOT preprocess the state here.
        """
        raise NotImplementedError("You need to override the "
                                  "Trainer.compute_values() function.")

    def compute_action(self, processed_state, eps=None):
        """Compute the action given the state. Note that the input
        is the processed state."""

        values = self.compute_values(processed_state)
        assert values.ndim == 1, values.shape

        if eps is None:
            eps = self.eps

        # [TODO] Implement the epsilon-greedy policy here. We have `eps`
        #  probability to choose a uniformly random action in action_space,
        #  otherwise choose action that maximizes the values.
        # Hint: Use the function of self.env.action_space to sample random
        # action.
        if np.random.uniform(0, 1)  <= eps:
            action = np.random.randint(self.env.action_space.n)
        else:
            action = np.argmax(values)

        return action

    def evaluate(self, num_episodes=50, *args, **kwargs):
        """Use the function you write to evaluate current policy.
        Return the mean episode reward of 50 episodes."""
        policy = lambda raw_state: self.compute_action(
            self.process_state(raw_state), eps=0.0)
        result = evaluate(policy, num_episodes, seed=self.seed,
                          env_name=self.env_name, *args, **kwargs)
        return result

    def compute_gradient(self, *args, **kwargs):
        """Compute the gradient."""
        raise NotImplementedError(
            "You need to override the Trainer.compute_gradient() function.")

    def apply_gradient(self, *args, **kwargs):
        """Compute the gradient"""
        raise NotImplementedError(
            "You need to override the Trainer.apply_gradient() function.")

    def train(self):
        """Conduct one iteration of learning."""
        raise NotImplementedError("You need to override the "
                                  "Trainer.train() function.")
```

In [5]:

```
# Run this cell without modification

class TestTrainer(AbstractTrainer):
    """This class is used for testing. We don't really train anything."""
    def compute_values(self, state):
        return np.random.random_sample(size=self.act_dim)
    def initialize_parameters(self):
        self.parameters = np.random.random_sample(size=(self.obs_dim, self.act_dim))

t = TestTrainer(dict(env_name="CartPole-v0"))
obs = t.env.observation_space.sample()
processed = t.process_state(obs)
assert processed.shape == (4, )
assert np.all(processed == obs)
# Test compute_action
values = t.compute_values(processed)
correct_act = np.argmax(values)
assert t.compute_action(processed, eps=0) == correct_act
print("Average episode reward for a random policy in 500 episodes in CartPole-v0: ",
      t.evaluate(num_episodes=500))
```

```
Average episode reward for a random policy in 500 episodes in CartPole-v0:  22.068
/home/baifan/anaconda3/envs/pt2/lib/python3.7/site-packages/gym/logger.py:30: UserWarning: WARN: Box bound precision lowe
red by casting to float32
  warnings.warn(colorize('%s: %s'%('WARN', msg % args), 'yellow'))
```

Section 2: Linear function approximation

In this section, we implement a simple linear function whose input is the state (or the processed state) and output is the state-action values.

First, we implement a `LinearTrainer` class which implements (1). Linear function approximation and (2). n-step semi-gradient method to update the linear function.

Then we further implement a `LinearTrainerWithFeatureConstruction` class which processs the input state and provide polynomial features which increase the utility of linear function approximation.

We refer the Chapter 9.4 (linear method), 9.5 (feature construction), and 10.2 (n-step semi-gradient method) of the RL textbook to you.

In this section, we leverage the n-step semi-gradient. What is the "correct value" of an action and state in one-step case? We consider it is $r_t + \gamma Q(s_{t+1}, a_{t+1})$ and thus lead to the TD error $TD = r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$. In n-step case, the target value of Q is:

$$Q(s_t, a_t) = \sum_{i=t}^{t+n-1} \gamma^{i-t} r_i + \gamma^n Q(s_{t+n}, a_{t+n})$$

We follow the pipeline depicted in Chapter 10.2 (page 247) of the textbook to implement this logic. Note that notation of the time step of reward is different in this assignment and in the textbook. In textbook, the reward R_{t+1} is the reward when apply action a_t to the environment at state s_t. In the equation above the r_t has exactly the same meaning. In the code below, we store the states, actions and rewards to a list during training. You need to make sure the indices of these list, like the `tau` in `actions[tau]` has the correct meaning.

After computing the target Q value, we need to derive the gradient to update the parameters. Consider a loss function, the Mean Square Error between the target Q value and the output Q value:

$$\text{loss} = \frac{1}{2}[\sum_{i=t}^{t+n-1} \gamma^{i-t} r_i + \gamma^n Q(s_{t+n}, a_{t+n}) - Q(s_t, a_t)]^2$$

Compute the gradient of Loss with respect to the Q function:

$$\frac{d\text{loss}}{dQ} = -(\sum_{i=t}^{t+n-1} \gamma^{i-t} r_i + \gamma^n Q(s_{t+n}, a_{t+n}) - Q(s_t, a_t))$$

According to the chain rule, the gradient of the loss w.r.t. the parameter (W) is:

$$\frac{d\text{loss}}{dW} = -(\sum_{i=t}^{t+n-1} \gamma^{i-t} r_i + \gamma^n Q(s_{t+n}, a_{t+n}) - Q(s_t, a_t))\frac{dQ}{dW}$$

To minimize the loss, we only need to descent the gradient:

$$W = W - lr\frac{d\text{loss}}{dW}$$

wherein lr is the learning rate. Therefore, in conclusion the update rule of parameters is:

$$W = W + lr(\sum_{i=t}^{t+n-1} \gamma^{i-t} r_i + \gamma^n Q(s_{t+n}, a_{t+n}) - Q(s_t, a_t))\frac{dQ}{dW}$$

Section 2.1: Basics

(30 / 100 points)

We want to approximate the state-action values. That is, the expected return when applying action a_t in state s_t. Linear methods approximate state-action value function by the inner product between a parameter matatrix W and the input state vector s:

$v(s, W) = W^T s$

Note that $W \in \mathbb{R}^{(O,A)}$ and $s \in \mathbb{R}^{(O,1)}$, wherein O is the observation (state) dimensions, namely the `self.obs_dim` in trainer and A is the action dimension, namely the `self.act_dim` in trainer. Each action corresponding to one state-action values $Q(s, a)$.

Note that you should finish this section **purely by Numpy without calling any other packages**.

In [6]:

```python
# Solve the TODOs and remove `pass`

# Build the algorithm-specify config.
linear_approximator_config = merge_config(dict(
    parameter_std=0.01,
    learning_rate=0.01,
    n=3,
), default_config)


class LinearTrainer(AbstractTrainer):
    def __init__(self, config):
        config = merge_config(config, linear_approximator_config)

        # Initialize the abstract class.
        super().__init__(config)

        self.max_episode_length = self.config["max_episode_length"]
        self.learning_rate = self.config["learning_rate"]
        self.gamma = self.config["gamma"]
        self.n = self.config["n"]

    def initialize_parameters(self):
        # [TODO] Initialize self.parameters, which is two dimensional matrix,
        #  and subjects to a normal distribution with scale
        #  config["parameter_std"].
        std = self.config["parameter_std"]
        self.parameters = np.random.normal(0, std, size=(self.obs_dim, self.act_dim))

        print("Initialize parameters with shape: {}.".format(
            self.parameters.shape))

    def compute_values(self, processed_state):
        # [TODO] Compute the value for each potential action. Note that you
        #  should NOT preprocess the state here."""
        assert processed_state.ndim == 1, processed_state.shape
```

```
        ret = np.dot(self.parameters.T,processed_state)

        return ret

    def train(self):
        """
        Please implement the n-step Sarsa algorithm presented in Chapter 10.2
        of the textbook. You algorithm should reduce the convention one-step
        Sarsa when n = 1. That is:
            TD = r_t + gamma * Q(s_t+1, a_t+1) - Q(s_t, a_t)
            Q(s_t, a_t) = Q(s_t, a_t) + learning_rate * TD
        """
        s = self.env.reset()
        processed_s = self.process_state(s)
        processed_states = [processed_s]
        rewards = [0.0]
        actions = [self.compute_action(processed_s)]
        T = float("inf")

        for t in range(self.max_episode_length):
            if t < T:
                # [TODO]  When the termination is not reach, apply action,
                #  process state, record state / reward / action to the
                #  lists defined above, and deal with termination.
                next_state, reward, done, _ = self.env.step(actions[t])

                processed_s = self.process_state(next_state)
                processed_states.append(processed_s)
                rewards.append(reward)
                if done:
                    T = t + 1

                else:
                    next_act = self.compute_action(processed_s)
                    actions.append(next_act)

            tau = t - self.n + 1
            if tau >= 0:
                gradient = self.compute_gradient(
                    processed_states, actions, rewards, tau, T
                )
                self.apply_gradient(gradient)
            if tau == T - 1:
                break

    def compute_gradient(self, processed_states, actions, rewards, tau, T):
        """Compute the gradient"""
        n = self.n

        # [TODO] Compute the approximation goal, the truth state action value
        #  G. It is a n-step discounted sum of rewards. Refer to Chapter 10.2
        #  of the textbook.
        # [HINT] G have two parts: the accumuted reward computed from step tau to
        #  step tau+n, and the possible state value at time step tau+n, if the episode
        #  is not terminated. Remember to apply the discounter factor (\gamma^n) to
        #  the second part of G if applicable.

        G = 0
        for i in range(tau,min(T,tau + n)):
            G += np.power(self.gamma,i - tau) * rewards[i+1]

        if tau + n < T:
            # [TODO] If at time step tau + n the episode is not terminated,
            # then we should add the state action value at tau + n
            # to the G.
            G += np.power(self.gamma,n) * self.compute_values(processed_states[tau + n])[actions[tau + n]]


        # Denote the state-action value function Q, then the loss of
        # prediction error w.r.t. the weights can be separated into two
        # parts (the chain rule):
        #     dLoss / dweight = (dLoss / dQ) * (dQ / dweight)
        # We call the first one loss_grad, and the latter one
        # value_grad. We consider the Mean Square Error between the target
        # value (G) and the predicted value (Q(s_t, a_t)) to be the loss.

        loss_grad = np.zeros((self.act_dim, 1))
        # [TODO] fill the propoer value of loss_grad, denoting the gradient
        # of the MSE w.r.t. the output of the linear function.
        loss_grad[actions[tau]] = -(G - self.compute_values(processed_states[tau]))[actions[tau]]

        # [TODO] compute the value of value_grad, denoting the gradient of
        # the output of the linear function w.r.t. the parameters.
        value_grad = processed_states[tau].reshape(self.obs_dim, 1)

        assert loss_grad.shape == (self.act_dim, 1)
        assert value_grad.shape == (self.obs_dim, 1)

        # [TODO] merge two gradients to get the gradient of loss w.r.t. the
        # parameters.
        gradient = np.dot(loss_grad, value_grad.T).T

        return gradient

    def apply_gradient(self, gradient):
        """Apply the gradient to the parameter."""
        assert gradient.shape == self.parameters.shape, (
            gradient.shape, self.parameters.shape)
```

```
        # [TODO] apply the gradient to self.parameters
        self.parameters -= self.learning_rate * gradient
```

In [7]:
```
# Run this cell without modification

# Build the test trainer.
test_trainer = LinearTrainer(dict(parameter_std=0.0))

# Test self.parameters.
assert test_trainer.parameters.std() == 0.0, \
    "Parameters should subjects to a normal distribution with standard " \
    "deviation config['parameter_std'], but you have {}." \
    "".format(test_trainer.parameters.std())
assert test_trainer.parameters.mean() == 0, \
    "Parameters should subjects to a normal distribution with mean 0. " \
    "But you have {}.".format(test_trainer.parameters.mean())

# Test compute_values
fake_state = test_trainer.env.observation_space.sample()
processed_state = test_trainer.process_state(fake_state)
assert processed_state.shape == (test_trainer.obs_dim, ), processed_state.shape
values = test_trainer.compute_values(fake_state)
assert values.shape == (test_trainer.act_dim, ), values.shape

# Test compute_gradient
tmp_gradient = test_trainer.compute_gradient(
    [processed_state]*10, [test_trainer.env.action_space.sample()]*10, [0.0]*10, 2, 5)
assert tmp_gradient.shape == test_trainer.parameters.shape

test_trainer.train()
print("Now your codes should be bug-free.")
```

```
Initialize parameters with shape: (4, 2).
Now your codes should be bug-free.
```

In [8]:
```
# Run this cell without modification

linear_trainer, _ = run(LinearTrainer, dict(
    max_iteration=10000,
    evaluate_interval=1000,
    parameter_std=0.01,
    learning_rate=0.01,
    n=3,
    env_name="CartPole-v0"
))

# It's OK to see bad performance
```

```
Initialize parameters with shape: (4, 2).
(0.0s,+0.0s)    Iteration 0, current mean episode reward is 9.28.
(0.5s,+0.5s)    Iteration 1000, current mean episode reward is 9.6.
(1.0s,+0.5s)    Iteration 2000, current mean episode reward is 9.72.
(1.6s,+0.5s)    Iteration 3000, current mean episode reward is 9.78.
(2.1s,+0.5s)    Iteration 4000, current mean episode reward is 9.8.
(2.6s,+0.5s)    Iteration 5000, current mean episode reward is 9.84.
(3.2s,+0.6s)    Iteration 6000, current mean episode reward is 9.84.
(3.8s,+0.6s)    Iteration 7000, current mean episode reward is 9.84.
(4.3s,+0.6s)    Iteration 8000, current mean episode reward is 9.84.
(4.9s,+0.6s)    Iteration 9000, current mean episode reward is 9.82.
(5.5s,+0.6s)    Iteration 10000, current mean episode reward is 9.84.
```

In [9]:
```
# Run this cell without modification

# You should see a pop up window which display the movement of the cart and pole.
print("Average episode reward for your linear agent in CartPole-v0: ",
      linear_trainer.evaluate(1, render=True))
```

```
Average episode reward for your linear agent in CartPole-v0:  10.0
```

You will notice that the linear trainer only has 8 trainable parameters and its performance is quiet bad. In the following section, we will expand the size of parameters and introduce more features as the input to the system so that the system can learn more complex value function.

Section 2.2: Linear Model with Feature Construction

(15 / 100 points)

In [10…
```
# Solve the TODOs and remove `pass`

linear_fc_config = merge_config(dict(
    polynomial_order=1,
), linear_approximator_config)


def polynomial_feature(sequence, order=1):
    """Construct the order-n polynomial-basis feature of the state.
    Refer to Chapter 9.5.1 of the textbook. We expect to get a
    vector of length (n+1)^k as the output.
    Example:
    When the state is [2, 3, 4], the first order polynomial feature
    of the state is [
        1,
        2,
        3,
        4,
        2 * 3 = 6,
        2 * 4 = 8,
        3 * 4 = 12,
        2 * 3 * 4 = 24
    ]
```

```
    It's OK for function polynomial() to return values in different order.
    """
    # [TODO] finish this function.
    output = np.ones((order+1,1))
    j = 0
    for i in sequence:
        order_i = np.asarray([np.power(i,n) for n in range(order+1)]).reshape(1,-1)
        if j == 0:
            output = order_i.reshape(-1,1)
        else:
            output = np.matmul(output,order_i).reshape(-1,1)
        j += 1
    return output.squeeze()

assert sorted(polynomial_feature([2, 3, 4])) == [1, 2, 3, 4, 6, 8, 12, 24]
assert len(polynomial_feature([2, 3, 4], 2)) == 27
assert len(polynomial_feature([2, 3, 4], 3)) == 64

class LinearTrainerWithFeatureConstruction(LinearTrainer):
    """In this class, we will expand the dimension of the state.
    This procedure is done at self.process_state function.
    The modification of self.obs_dim and the shape of parameters
    is also needed.
    """
    def __init__(self, config):
        config = merge_config(config, linear_fc_config)
        # Initialize the abstract class.
        super().__init__(config)

        self.polynomial_order = self.config["polynomial_order"]

        # Expand the size of observation
        self.obs_dim = (self.polynomial_order + 1) ** self.obs_dim

        # Since we change self.obs_dim, reset the parameters.
        self.initialize_parameters()

    def process_state(self, state):
        """Please finish the polynomial function."""
        processed = polynomial_feature(state, self.polynomial_order)
        processed = np.asarray(processed)
        assert len(processed) == self.obs_dim, processed.shape
        return processed
```

In [11…

```
# Run this cell without modification

linear_fc_trainer, _ = run(LinearTrainerWithFeatureConstruction, dict(
    max_iteration=10000,
    evaluate_interval=1000,
    parameter_std=0.01,
    learning_rate=0.001,
    polynomial_order=1,
    n=3,
    env_name="CartPole-v0"
), reward_threshold=195.0)

assert linear_fc_trainer.evaluate() > 20.0, "The best episode reward happening " \
    "during training should be greater than the random baseline, that is greather than 20+."

# This cell should be finished within 10 minitines.
```

```
Initialize parameters with shape: (4, 2).
Initialize parameters with shape: (16, 2).
(0.2s,+0.2s)    Iteration 0, current mean episode reward is 72.78.
(3.1s,+2.8s)    Iteration 1000, current mean episode reward is 15.34.
(6.2s,+3.1s)    Iteration 2000, current mean episode reward is 53.58.
(10.0s,+3.8s)   Iteration 3000, current mean episode reward is 35.98.
(13.9s,+3.8s)   Iteration 4000, current mean episode reward is 27.76.
(17.6s,+3.7s)   Iteration 5000, current mean episode reward is 79.74.
(21.3s,+3.7s)   Iteration 6000, current mean episode reward is 30.3.
(24.9s,+3.6s)   Iteration 7000, current mean episode reward is 98.14.
(28.7s,+3.8s)   Iteration 8000, current mean episode reward is 106.24.
(32.7s,+4.0s)   Iteration 9000, current mean episode reward is 37.0.
(40.8s,+8.1s)   Iteration 10000, current mean episode reward is 109.04.
```

In [12…

```
# Run this cell without modification

# You should see a pop up window which display the movement of the cart and pole.
print("Average episode reward for your linear agent with feature constructioin in CartPole-v0: ",
      linear_fc_trainer.evaluate(1, render=True))
```

```
Average episode reward for your linear agent with feature constructioin in CartPole-v0:  114.0
```

Section 3: Multi-layer Perceptron as the approximiator

In this section, you are required to implement a single agent MLP using purely Numpy package. The differences between MLP and linear function are (1). MLP has a hidden layer which increase its representation capacity (2). MLP can leverage activation function after the output of each layer which introduce not linearity.

Consider a MLP with one hidden layer containing 100 neurons and activation function `f()`. We call the layer that accepts the state as input and output the activation **hidden layer**, and the layer that accepts the activation as input and produces the values **output layer**. The activation of the hidden layer is:

$$a(s_t) = f(W_h^T s_t)$$

obvious the activation is a 100-length vector. The output values is:

$$Q(s_t) = f(W_o^T a(s_t))$$

wherein W_h, W_o are the parameters of hidden layer and output layer, respectively. In this section we do not add activation function and hence $f(x) = x$.

Moreover, we also introduce the gradient clipping mechanism. In on-policy learning, the norm of gradient is prone to vary drastically, since the output of Q function is unbounded and it can be as large as possible, which leads to exploding gradient issue. Gradient clipping is used to bound the norm of gradient while keeps the direction of gradient vector unchanged. Concretely, the formulation of gradient clipping is:

$$g_{clipped} = g_{original} \frac{c}{\max(c, \text{norm}(g))}$$

wherein c is a hyperparameter which is `config["clip_norm"]` in our implementation. Gradient clipping bounds the gradient norm to c if the norm of

In [13…

```
# Solve the TODOs and remove `pass`

# Build the algorithm-specify config.
mlp_trainer_config = merge_config(dict(
    parameter_std=0.01,
    learning_rate=0.01,
    hidden_dim=100,
    n=3,
    clip_norm=1.0,
    clip_gradient=True
), default_config)


class MLPTrainer(LinearTrainer):
    def __init__(self, config):
        config = merge_config(config, mlp_trainer_config)
        self.hidden_dim = config["hidden_dim"]
        super().__init__(config)

    def initialize_parameters(self):
        # [TODO] Initialize self.hidden_parameters and self.output_parameters,
        #  which are two dimensional matrices, and subject to normal
        #  distributions with scale config["parameter_std"]
        std = self.config["parameter_std"]
        self.hidden_parameters = np.random.normal(0, std, size=(self.obs_dim, self.hidden_dim))
        self.output_parameters = np.random.normal(0, std, size=(self.hidden_dim, self.act_dim))


    def compute_values(self, processed_state):
        """[TODO] Compute the value for each potential action. Note that you
        should NOT preprocess the state here."""
        assert processed_state.ndim == 1, processed_state.shape
        activation = self.compute_activation(processed_state)
        values = np.dot(self.output_parameters.T,activation)
        values = values.squeeze()


        return values

    def compute_activation(self, processed_state):
        """[TODO] Compute the action values values.
        Given a processed state, first we need to compute the activtaion
        (the output of hidden layer). Then we compute the values (the output of
        the output layer).
        """
        activation = np.dot(self.hidden_parameters.T,processed_state[:,np.newaxis])

        return activation

    def compute_gradient(self, processed_states, actions, rewards, tau, T):
        n = self.n

        # [TODO] compute the target value.
        # Hint: copy your codes in LinearTrainer.
        G = 0
        for i in range(tau,min(T,tau + n)):
            G += np.power(self.gamma,i - tau) * rewards[i+1]

        if tau + n < T:
            # [TODO] If at time step tau + n the episode is not terminated,
            # then we should add the state action value at tau + n
            # to the G.
            G += np.power(self.gamma,n) * self.compute_values(processed_states[tau + n])[actions[tau + n]]

        # Denote the state-action value function Q, then the loss of
        # prediction error w.r.t. the output layer weights can be
        # separated into two parts (the chain rule):
        #     dError / dweight = (dError / dQ) * (dQ / dweight)
        # We call the first one loss_grad, and the latter one
        # value_grad. We consider the Mean Square Error between the target
        # value (G) and the predict value (Q(s_t, a_t)) to be the loss.
        cur_state = processed_states[tau]

        loss_grad = np.zeros((self.act_dim, 1))  # [act_dim, 1]
        # [TODO] compute loss_grad
        loss_grad[[actions[tau]]] = -(G - self.compute_values(cur_state)[actions[tau]])


        # [TODO] compute the gradient of output layer parameters
        output_gradient = np.dot(loss_grad, self.compute_activation(cur_state).T).T


        # [TODO] compute the gradient of hidden layer parameters
```

```
        # Hint: using chain rule and derive the formulation
        hidden_gradient = np.dot(np.dot(self.output_parameters, loss_grad), cur_state.reshape(cur_state.shape[0], 1).T).T

        assert np.all(np.isfinite(output_gradient)), \
            "Invalid value occurs in output_gradient! {}".format(
                output_gradient)
        assert np.all(np.isfinite(hidden_gradient)), \
            "Invalid value occurs in hidden_gradient! {}".format(
                hidden_gradient)
        return [hidden_gradient, output_gradient]

    def apply_gradient(self, gradients):
        """Apply the gradientss to the two layers' parameters."""
        assert len(gradients) == 2
        hidden_gradient, output_gradient = gradients

        assert output_gradient.shape == (self.hidden_dim, self.act_dim)
        assert hidden_gradient.shape == (self.obs_dim, self.hidden_dim)

        # [TODO] Implement the clip gradient mechansim
        # Hint: when the old gradient has norm less that clip_norm,
        #  then nothing happens. Otherwise shrink the gradient to
        #  make its norm equal to clip_norm.
        if self.config["clip_gradient"]:
            clip_norm = self.config["clip_norm"]
            hidden_gradient *= clip_norm / max(clip_norm, np.linalg.norm(hidden_gradient))
            output_gradient *= clip_norm / max(clip_norm, np.linalg.norm(output_gradient))
        # [TODO] update the parameters
        # Hint: Remember to check the sign when applying the gradient
        #  into the parameters. Should you add or minus the gradients?
        self.hidden_parameters -= self.learning_rate * hidden_gradient
        self.output_parameters -= self.learning_rate * output_gradient
```

In [14…

```
# Run this cell without modification

print("Now let's see what happen if clip gradient is not enable!")
try:
    failed_mlp_trainer, _ = run(MLPTrainer, dict(
        max_iteration=3000,
        evaluate_interval=100,
        parameter_std=0.01,
        learning_rate=0.001,
        hidden_dim=100,
        clip_gradient=False,  # <<< Gradient clipping is OFF!
        env_name="CartPole-v0"
    ), reward_threshold=195.0)
    print("We expect to see bad performance (<195). "
          "The performance without gradient clipping: {}."
          "".format(failed_mlp_trainer.evaluate()))
except AssertionError as e:
    print(traceback.format_exc())
    print("Infinity happen during training. It's OK since the gradient is not bounded.")
finally:
    print("Try next cell to see the impact of gradient clipping.")
```

```
Now let's see what happen if clip gradient is not enable!
(0.0s,+0.0s)    Iteration 0, current mean episode reward is 33.52.
(0.6s,+0.6s)    Iteration 100, current mean episode reward is 92.24.
(1.2s,+0.6s)    Iteration 200, current mean episode reward is 87.3.
(1.8s,+0.6s)    Iteration 300, current mean episode reward is 79.46.
(2.4s,+0.6s)    Iteration 400, current mean episode reward is 79.7.
(3.0s,+0.6s)    Iteration 500, current mean episode reward is 73.84.
(3.6s,+0.6s)    Iteration 600, current mean episode reward is 76.54.
(4.0s,+0.5s)    Iteration 700, current mean episode reward is 63.74.
(4.5s,+0.5s)    Iteration 800, current mean episode reward is 60.8.
(5.0s,+0.5s)    Iteration 900, current mean episode reward is 59.18.
(5.4s,+0.4s)    Iteration 1000, current mean episode reward is 56.48.
(5.9s,+0.4s)    Iteration 1100, current mean episode reward is 52.9.
(6.2s,+0.4s)    Iteration 1200, current mean episode reward is 48.04.
(6.6s,+0.4s)    Iteration 1300, current mean episode reward is 46.54.
(7.0s,+0.4s)    Iteration 1400, current mean episode reward is 42.66.
(7.4s,+0.4s)    Iteration 1500, current mean episode reward is 45.02.
(7.7s,+0.3s)    Iteration 1600, current mean episode reward is 40.32.
(8.0s,+0.3s)    Iteration 1700, current mean episode reward is 44.48.
(8.4s,+0.3s)    Iteration 1800, current mean episode reward is 42.72.
(8.7s,+0.3s)    Iteration 1900, current mean episode reward is 40.04.
(9.0s,+0.3s)    Iteration 2000, current mean episode reward is 33.58.
(9.3s,+0.3s)    Iteration 2100, current mean episode reward is 33.58.
(9.6s,+0.3s)    Iteration 2200, current mean episode reward is 32.98.
(9.8s,+0.3s)    Iteration 2300, current mean episode reward is 33.58.
(10.1s,+0.3s)   Iteration 2400, current mean episode reward is 32.94.
(10.4s,+0.3s)   Iteration 2500, current mean episode reward is 32.66.
(10.6s,+0.3s)   Iteration 2600, current mean episode reward is 33.08.
(10.9s,+0.3s)   Iteration 2700, current mean episode reward is 33.22.
(11.2s,+0.3s)   Iteration 2800, current mean episode reward is 33.28.
(11.5s,+0.3s)   Iteration 2900, current mean episode reward is 33.1.
(11.7s,+0.3s)   Iteration 3000, current mean episode reward is 33.24.
We expect to see bad performance (<195). The performance without gradient clipping: 33.24.
Try next cell to see the impact of gradient clipping.
```

In [15…

```
# Run this cell without modification

print("Now let's see what happen if clip gradient is not enable!")
mlp_trainer, _ = run(MLPTrainer, dict(
    max_iteration=3000,
    evaluate_interval=100,
    parameter_std=0.01,
    learning_rate=0.001,
    hidden_dim=100,
```

```
    clip_gradient=True,  # <<< Gradient clipping is ON!
    env_name="CartPole-v0"
), reward_threshold=195.0)

assert mlp_trainer.evaluate() > 195.0, "Check your codes. " \
    "Your agent should achieve {} reward in 200 iterations." \
    "But it achieve {} reward in evaluation."

# In our implementation, the task is solved in 200 iterations.
```

```
Now let's see what happen if clip gradient is not enable!
(0.0s,+0.0s)    Iteration 0, current mean episode reward is 33.52.
(0.7s,+0.7s)    Iteration 100, current mean episode reward is 92.44.
(1.5s,+0.8s)    Iteration 200, current mean episode reward is 93.76.
(2.3s,+0.8s)    Iteration 300, current mean episode reward is 98.18.
(3.1s,+0.8s)    Iteration 400, current mean episode reward is 103.18.
(3.9s,+0.8s)    Iteration 500, current mean episode reward is 108.46.
(4.8s,+0.9s)    Iteration 600, current mean episode reward is 111.26.
(5.7s,+0.9s)    Iteration 700, current mean episode reward is 121.48.
(6.6s,+0.9s)    Iteration 800, current mean episode reward is 135.22.
(7.8s,+1.1s)    Iteration 900, current mean episode reward is 178.46.
(9.2s,+1.4s)    Iteration 1000, current mean episode reward is 198.96.
In 1000 iteration, current mean episode reward 198.960 is greater than reward threshold 195.0. Congratulation! Now we exi
t the training process.
```

In [16…

```
# Run this cell without modification

# You should see a pop up window which display the movement of the cart and pole.
print("Average episode reward for your MLP agent with gradient clipping in CartPole-v0: ",
      mlp_trainer.evaluate(1, render=True))
```

```
Average episode reward for your MLP agent with gradient clipping in CartPole-v0:  200.0
```

Interesting right? The gradient clipping technique makes the training converge much faster!

Section 4: Implement Deep Q Learning in Pytorch

(50 / 100 points)

In this section, you will get familiar with the basic logic of pytorch, which lay the ground for further learning. We will implement a MLP similar to the one in Section 3 using Pytorch, a powerful Deep Learning framework. Before start, you need to make sure using `pip install torch` to install it.

If you are not familiar with Pytorch, we suggest you to go through pytorch official quickstart tutorials:

1. quickstart
2. tutorial on RL

Different from the algorithm in Section 3, we will implement Deep Q Network (DQN) in this section. The main differences are concluded as following:

DQN requires an experience replay memory to store the transitions. A replay memory is implemented in the following `ExperienceReplayMemory` class. It can contain a certain amount of transitions: `(s_t, a_t, r_t, s_t+1, done_t)`. When the memory is full, the earliest transition is discarded to store the latest one.

The introduction of replay memory increase the sample efficiency (since each transition might be used multiple times) when solving complex task, though you may find it learn slowly in this assignment since the CartPole-v0 is a relatively easy environment.

DQN is an off-policy algorithm and has difference when computing TD error, compared to Sarsa. In Sarsa, the TD error is computed as:

$$(r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$$

wherein the next action a_{t+1} is the one the policy selects. However, in traditional Q learning, it assume the next action is the one that maximizes the action values and use this assumption to compute the TD:

$$(r_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$$

DQN has make delayed update target network, which is another difference even compared to the traditional Q learning. DQN maintains another neural network called target network that has identical structure of the Q network. After a certain amount of steps has been taken, the target network copies the parameters of the Q network to itself. Normally, the update of target network is much less frequent than the update of the Q network. The Q network is updated in each step.

The reason to leverage the target network is to stabilize the estimation of TD error. In DQN, the TD error is evaluated as:

$$(r_t + \gamma \max_{a_{t+1}} Q^{target}(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$$

The Q values of next state is estimated by the target network, not the Q network that is updating. This mechanism can reduce the variance of gradient because the estimation of Q values of next states is not influenced by the update of the Q network.

In the engineering aspect, the differences between `DQNTrainer` and the previous `MLPTrainer` are:

1. DQN uses pytorch model to serve as the approximator. So we need to rewrite the `initialize_parameter` function to build the pytorch model. Also the `train` function is changed since the gradient optimization is conducted by pytorch, therefore we need to write the pytorch pipeline in `train`.
2. DQN has replay memory. So we need to initialize it, feed data into it and take the transitions out.
3. Thank to the replay memory and pytorch, DQN can be updated in a batch. So you need to carefully compute the Q target via matrix computation.
4. We use Adam optimizer to conduct the gradient optimization. You need to get familiar with how to compute the loss and conduct backward propagation.

In [17…

```
# Solve the TODOs and remove `pass`

from collections import deque
```

```python
import random

class ExperienceReplayMemory:
    """Store and sample the transitions"""
    def __init__(self, capacity):
        # deque is a useful class which acts like a list but only contain
        # finite elements.When appending new element make deque exceeds the
        # `maxlen`, the oldest element (the index 0 element) will be removed.

        # [TODO] uncomment next line.
        self.memory = deque(maxlen=capacity)

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

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

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

In [18…

```python
# Solve the TODOs and remove `pass`

class PytorchModel(nn.Module):
    def __init__(self, input_shape, num_actions):
        super(PytorchModel, self).__init__()

        # [TODO] Build a sequential model with two layers.
        # The first hidden layer has 100 hidden nodes, followed by
        # a ReLU activation function.
        # The second output layer take the activation vector, who has
        # 100 elements, as input and return the action values.
        # So the return values is a vector with num_actions elements.
        self.action_value = nn.Sequential(
            nn.Linear(input_shape[0], 100),
            nn.ReLU(),
            nn.Linear(100, num_actions),
        )

    def forward(self, obs):
        return self.action_value(obs)

# Test
assert isinstance(PytorchModel((3,), 7).action_value, nn.Module)
```

In [19…

```python
# Solve the TODOs and remove `pass`

pytorch_config = merge_config(dict(
    memory_size=50000,
    learn_start=5000,
    batch_size=32,
    target_update_freq=500,  # in steps
    learn_freq=1,  # in steps
    n=1
), mlp_trainer_config)


def to_tensor(x):
    """A helper function to transform a numpy array to a Pytorch Tensor"""
    if isinstance(x, np.ndarray):
        x = torch.from_numpy(x).type(torch.float32)
    assert isinstance(x, torch.Tensor)
    if x.dim() == 3 or x.dim() == 1:
        x = x.unsqueeze(0)
    assert x.dim() == 2 or x.dim() == 4, x.shape
    return x


class DQNTrainer(MLPTrainer):
    def __init__(self, config):
        config = merge_config(config, pytorch_config)
        self.learning_rate = config["learning_rate"]
        super().__init__(config)

        self.memory = ExperienceReplayMemory(config["memory_size"])
        self.learn_start = config["learn_start"]
        self.batch_size = config["batch_size"]
        self.target_update_freq = config["target_update_freq"]
        self.clip_norm = config["clip_norm"]
        self.step_since_update = 0
        self.total_step = 0

    def initialize_parameters(self):
        input_shape = self.env.observation_space.shape

        # [TODO] Initialize two network using PytorchModel class
        self.network = PytorchModel((self.obs_dim,),  self.act_dim)

        self.network.eval()
        self.network.share_memory()

        # [TODO] Initialize target network then copy the weight
        # of original network to it. So you should
        # put the weights of self.network into self.target_network.
        self.target_network = PytorchModel((self.obs_dim,), self.act_dim)
        self.target_network.load_state_dict(self.network.state_dict())
```

```
        self.target_network.eval()

        # Build Adam optimizer and MSE Loss.
        # [TODO] Uncomment next few lines
        self.optimizer = torch.optim.Adam(
            self.network.parameters(), lr=self.learning_rate
        )
        self.loss = nn.MSELoss()


    def compute_values(self, processed_state):
        """Compute the value for each potential action. Note that you
        should NOT preprocess the state here."""
        # [TODO] Convert the output of neural network to numpy array
        values = self.network(processed_state).detach().numpy()

        return values

    def train(self):
        s = self.env.reset()
        processed_s = self.process_state(s)
        act = self.compute_action(processed_s)
        stat = {"loss": []}

        for t in range(self.max_episode_length):
            next_state, reward, done, _ = self.env.step(act)
            next_processed_s = self.process_state(next_state)

            # Push the transition into memory.
            self.memory.push(
                (processed_s, act, reward, next_processed_s, done)
            )

            processed_s = next_processed_s
            act = self.compute_action(next_processed_s)
            self.step_since_update += 1
            self.total_step += 1

            if done:
                break

            if t % self.config["learn_freq"] != 0:
                # It's not necessary to update in each step.
                continue

            if len(self.memory) < self.learn_start:
                continue
            elif len(self.memory) == self.learn_start:
                print("Current memory contains {} transitions, "
                      "start learning!".format(self.learn_start))

            batch = self.memory.sample(self.batch_size)

            # Transform a batch of state / action / .. into a tensor.
            state_batch = to_tensor(
                np.stack([transition[0] for transition in batch])
            )
            action_batch = to_tensor(
                np.stack([transition[1] for transition in batch])
            )
            reward_batch = to_tensor(
                np.stack([transition[2] for transition in batch])
            )
            next_state_batch = torch.stack(
                [transition[3] for transition in batch]
            )
            done_batch = to_tensor(
                np.stack([transition[4] for transition in batch])
            )

            with torch.no_grad():
                # [TODO] Compute the values of Q in next state in batch.
                Q_t_plus_one = torch.max(self.target_network(next_state_batch).detach(), 1)[0]

                assert isinstance(Q_t_plus_one, torch.Tensor)
                assert Q_t_plus_one.dim() == 1

                # [TODO] Compute the target value of Q in batch.
                Q_target = (reward_batch + (1 - done_batch) * self.gamma * Q_t_plus_one).reshape(self.batch_size,)

                assert Q_target.shape == (self.batch_size,)

            # [TODO] Collect the Q values in batch.
            # Hint: Remember to call self.network.train()
            #  before you get the Q value from self.network(state_batch),
            #  otherwise the graident will not be recorded by pytorch.
            self.network.train()
            Q_t = self.network(state_batch).gather(1, action_batch.reshape(self.batch_size,1).long()).squeeze()


            assert Q_t.shape == Q_target.shape

            # Update the network
            self.optimizer.zero_grad()
            loss = self.loss(input=Q_t, target=Q_target)
            loss_value = loss.item()
            stat['loss'].append(loss_value)
            loss.backward()
```

```
            # [TODO] Gradient clipping. Uncomment next line
            nn.utils.clip_grad_norm_(self.network.parameters(), self.clip_norm)


            self.optimizer.step()
            self.network.eval()

        if len(self.memory) >= self.learn_start and \
                self.step_since_update > self.target_update_freq:
            print("{} steps has passed since last update. Now update the"
                  " parameter of the behavior policy. Current step: {}".format(
                self.step_since_update, self.total_step
            ))
            self.step_since_update = 0
            # [TODO] Copy the weights of self.network to self.target_network.
            self.target_network.load_state_dict(self.network.state_dict())

            self.target_network.eval()

        return {"loss": np.mean(stat["loss"]), "episode_len": t}

    def process_state(self, state):
        return torch.from_numpy(state).type(torch.float32)
```

In [20…

```
# Run this cell without modification

# Build the test trainer.
test_trainer = DQNTrainer({})

# Test compute_values
fake_state = test_trainer.env.observation_space.sample()
processed_state = test_trainer.process_state(fake_state)
assert processed_state.shape == (test_trainer.obs_dim, ), processed_state.shape
values = test_trainer.compute_values(processed_state)
assert values.shape == (test_trainer.act_dim, ), values.shape

test_trainer.train()
print("Now your codes should be bug-free.")

_ = run(DQNTrainer, dict(
    max_iteration=20,
    evaluate_interval=10,
    learn_start=100,
    env_name="CartPole-v0",
))

print("Test passed!")
```

```
Now your codes should be bug-free.
(0.1s,+0.1s)    Iteration 0, current mean episode reward is 9.24. {'loss': nan, 'episode_len': 9.0}
Current memory contains 100 transitions, start learning!
/home/baifan/anaconda3/envs/pt2/lib/python3.7/site-packages/numpy/core/fromnumeric.py:3373: RuntimeWarning: Mean of empty
slice.
  out=out, **kwargs)
/home/baifan/anaconda3/envs/pt2/lib/python3.7/site-packages/numpy/core/_methods.py:170: RuntimeWarning: invalid value enc
ountered in double_scalars
  ret = ret.dtype.type(ret / rcount)
(0.2s,+0.1s)    Iteration 10, current mean episode reward is 9.3. {'loss': 0.0233, 'episode_len': 16.0}
(0.5s,+0.3s)    Iteration 20, current mean episode reward is 9.6. {'loss': 0.0026, 'episode_len': 10.0}
Test passed!
```

In [21…

```
# Run this cell without modification

pytorch_trainer, pytorch_stat = run(DQNTrainer, dict(
    max_iteration=2000,
    evaluate_interval=10,
    learning_rate=0.01,
    clip_norm=10.0,
    memory_size=50000,
    learn_start=1000,
    eps=0.1,
    target_update_freq=1000,
    batch_size=32,
    env_name="CartPole-v0",
), reward_threshold=195.0)

reward = pytorch_trainer.evaluate()
assert reward > 195.0, "Check your codes. " \
    "Your agent should achieve {} reward in 1000 iterations." \
    "But it achieve {} reward in evaluation.".format(195.0, reward)

# Should solve the task in 10 minutes
```

```
(0.2s,+0.2s)    Iteration 0, current mean episode reward is 35.44. {'loss': nan, 'episode_len': 28.0}
(0.4s,+0.2s)    Iteration 10, current mean episode reward is 35.44. {'loss': nan, 'episode_len': 38.0}
(0.7s,+0.2s)    Iteration 20, current mean episode reward is 35.44. {'loss': nan, 'episode_len': 34.0}
Current memory contains 1000 transitions, start learning!
1032 steps has passed since last update. Now update the parameter of the behavior policy. Current step: 1032
(0.9s,+0.2s)    Iteration 30, current mean episode reward is 26.84. {'loss': 0.0689, 'episode_len': 13.0}
(1.1s,+0.2s)    Iteration 40, current mean episode reward is 9.24. {'loss': 0.011, 'episode_len': 8.0}
(1.3s,+0.2s)    Iteration 50, current mean episode reward is 13.04. {'loss': 0.0503, 'episode_len': 8.0}
(1.5s,+0.2s)    Iteration 60, current mean episode reward is 9.48. {'loss': 0.0445, 'episode_len': 8.0}
(1.7s,+0.2s)    Iteration 70, current mean episode reward is 9.46. {'loss': 0.0495, 'episode_len': 9.0}
(1.9s,+0.2s)    Iteration 80, current mean episode reward is 9.82. {'loss': 0.0626, 'episode_len': 8.0}
(2.1s,+0.2s)    Iteration 90, current mean episode reward is 9.24. {'loss': 0.0524, 'episode_len': 11.0}
(2.3s,+0.2s)    Iteration 100, current mean episode reward is 9.58. {'loss': 0.0598, 'episode_len': 9.0}
(2.5s,+0.2s)    Iteration 110, current mean episode reward is 9.24. {'loss': 0.0549, 'episode_len': 8.0}
(2.7s,+0.2s)    Iteration 120, current mean episode reward is 9.48. {'loss': 0.0484, 'episode_len': 7.0}
1004 steps has passed since last update. Now update the parameter of the behavior policy. Current step: 2036
(2.9s,+0.2s)    Iteration 130, current mean episode reward is 9.3. {'loss': 0.1032, 'episode_len': 10.0}
```

```
(3.0s,+0.2s)	Iteration 140, current mean episode reward is 10.14. {'loss': 0.0598, 'episode_len': 9.0}
(3.2s,+0.2s)	Iteration 150, current mean episode reward is 9.9. {'loss': 0.0615, 'episode_len': 10.0}
(3.4s,+0.2s)	Iteration 160, current mean episode reward is 9.78. {'loss': 0.0411, 'episode_len': 10.0}
(3.6s,+0.2s)	Iteration 170, current mean episode reward is 9.24. {'loss': 0.0933, 'episode_len': 8.0}
(3.8s,+0.2s)	Iteration 180, current mean episode reward is 9.24. {'loss': 0.0595, 'episode_len': 9.0}
(4.0s,+0.2s)	Iteration 190, current mean episode reward is 9.24. {'loss': 0.0826, 'episode_len': 9.0}
(4.2s,+0.2s)	Iteration 200, current mean episode reward is 9.24. {'loss': 0.0422, 'episode_len': 8.0}
(4.4s,+0.2s)	Iteration 210, current mean episode reward is 9.52. {'loss': 0.066, 'episode_len': 10.0}
1003 steps has passed since last update. Now update the parameter of the behavior policy. Current step: 3039
(4.6s,+0.2s)	Iteration 220, current mean episode reward is 9.24. {'loss': 0.1081, 'episode_len': 8.0}
(4.8s,+0.2s)	Iteration 230, current mean episode reward is 9.4. {'loss': 0.065, 'episode_len': 14.0}
(5.2s,+0.3s)	Iteration 240, current mean episode reward is 13.56. {'loss': 0.0861, 'episode_len': 9.0}
(5.4s,+0.2s)	Iteration 250, current mean episode reward is 11.94. {'loss': 0.0685, 'episode_len': 13.0}
(5.7s,+0.2s)	Iteration 260, current mean episode reward is 18.1. {'loss': 0.075, 'episode_len': 10.0}
(5.9s,+0.2s)	Iteration 270, current mean episode reward is 9.34. {'loss': 0.0697, 'episode_len': 7.0}
(6.1s,+0.3s)	Iteration 280, current mean episode reward is 11.14. {'loss': 0.0867, 'episode_len': 27.0}
1009 steps has passed since last update. Now update the parameter of the behavior policy. Current step: 4048
(6.4s,+0.2s)	Iteration 290, current mean episode reward is 11.18. {'loss': 0.1162, 'episode_len': 11.0}
(6.8s,+0.4s)	Iteration 300, current mean episode reward is 9.56. {'loss': 0.0855, 'episode_len': 33.0}
(7.3s,+0.5s)	Iteration 310, current mean episode reward is 12.62. {'loss': 0.0947, 'episode_len': 73.0}
(8.0s,+0.7s)	Iteration 320, current mean episode reward is 50.88. {'loss': 0.1124, 'episode_len': 9.0}
1017 steps has passed since last update. Now update the parameter of the behavior policy. Current step: 5065
(9.4s,+1.4s)	Iteration 330, current mean episode reward is 154.6. {'loss': 0.1592, 'episode_len': 31.0}
1016 steps has passed since last update. Now update the parameter of the behavior policy. Current step: 6081
(10.7s,+1.3s)	Iteration 340, current mean episode reward is 77.28. {'loss': 0.0864, 'episode_len': 66.0}
(12.9s,+2.2s)	Iteration 350, current mean episode reward is 179.62. {'loss': 0.1299, 'episode_len': 144.0}
1075 steps has passed since last update. Now update the parameter of the behavior policy. Current step: 7156
1069 steps has passed since last update. Now update the parameter of the behavior policy. Current step: 8225
(16.0s,+3.0s)	Iteration 360, current mean episode reward is 143.04. {'loss': 0.1457, 'episode_len': 179.0}
1068 steps has passed since last update. Now update the parameter of the behavior policy. Current step: 9293
1087 steps has passed since last update. Now update the parameter of the behavior policy. Current step: 10380
(19.3s,+3.3s)	Iteration 370, current mean episode reward is 187.6. {'loss': 0.2074, 'episode_len': 155.0}
1127 steps has passed since last update. Now update the parameter of the behavior policy. Current step: 11507
(22.3s,+3.0s)	Iteration 380, current mean episode reward is 164.2. {'loss': 0.3052, 'episode_len': 137.0}
1001 steps has passed since last update. Now update the parameter of the behavior policy. Current step: 12508
(24.7s,+2.4s)	Iteration 390, current mean episode reward is 107.42. {'loss': 0.2782, 'episode_len': 145.0}
1051 steps has passed since last update. Now update the parameter of the behavior policy. Current step: 13559
1090 steps has passed since last update. Now update the parameter of the behavior policy. Current step: 14649
(27.2s,+2.5s)	Iteration 400, current mean episode reward is 109.88. {'loss': 0.389, 'episode_len': 140.0}
1007 steps has passed since last update. Now update the parameter of the behavior policy. Current step: 15656
(29.7s,+2.5s)	Iteration 410, current mean episode reward is 160.44. {'loss': 0.4734, 'episode_len': 125.0}
1092 steps has passed since last update. Now update the parameter of the behavior policy. Current step: 16748
(32.7s,+3.0s)	Iteration 420, current mean episode reward is 199.4. {'loss': 0.6133, 'episode_len': 163.0}
In 420 iteration, current mean episode reward 199.400 is greater than reward threshold 195.0. Congratulation! Now we exit the training process.
```

In [22…

```python
# Run this cell without modification

# You should see a pop up window which display the movement of the cart and pole.
print("Average episode reward for your Pytorch agent in CartPole-v0: ",
      pytorch_trainer.evaluate(1, render=True))
```

```
Average episode reward for your Pytorch agent in CartPole-v0:  200.0
```

In [34…

```python
# [optional] BONUS!!! Train DQN in "Pong-ram-v0" environment
# Tune the hyperparameter and take some time to train agent
# You need to install gym[atari] first via `pip install gym[atari]`

pytorch_trainer2, _ = run(DQNTrainer2, dict(
    max_episode_length=500,
    max_iteration=5000,
    evaluate_interval=50,
    evaluate_num_episodes=10,
    learning_rate=0.00025,
    clip_norm=10.0,
    memory_size=10000,
    learn_start=1000,
    eps=0.1,
    target_update_freq=10000,
    learn_freq=4,
    batch_size=32,
    env_name="Pong-ram-v0"
), reward_threshold=-20)

# This environment is hard to train.
```

```
(11.6s,+11.6s)	Iteration 0, current mean episode reward is -21.0. {'loss': nan, 'episode_len': 499.0}
10500 steps has passed since last update. Now update the parameter of the behavior policy. Current step: 10500
10500 steps has passed since last update. Now update the parameter of the behavior policy. Current step: 21000
(141.5s,+129.8s)	Iteration 50, current mean episode reward is -21.0. {'loss': 0.0191, 'episode_len': 499.0}
10500 steps has passed since last update. Now update the parameter of the behavior policy. Current step: 31500
10500 steps has passed since last update. Now update the parameter of the behavior policy. Current step: 42000
(262.7s,+121.2s)	Iteration 100, current mean episode reward is -20.6. {'loss': 0.0046, 'episode_len': 499.0}
10500 steps has passed since last update. Now update the parameter of the behavior policy. Current step: 52500
10500 steps has passed since last update. Now update the parameter of the behavior policy. Current step: 63000
10500 steps has passed since last update. Now update the parameter of the behavior policy. Current step: 73500
(406.6s,+143.9s)	Iteration 150, current mean episode reward is -21.0. {'loss': 0.0026, 'episode_len': 499.0}
10500 steps has passed since last update. Now update the parameter of the behavior policy. Current step: 84000
10500 steps has passed since last update. Now update the parameter of the behavior policy. Current step: 94500
(523.4s,+116.8s)	Iteration 200, current mean episode reward is -21.0. {'loss': 0.0011, 'episode_len': 499.0}
10500 steps has passed since last update. Now update the parameter of the behavior policy. Current step: 105000
10500 steps has passed since last update. Now update the parameter of the behavior policy. Current step: 115500
(645.6s,+122.2s)	Iteration 250, current mean episode reward is -21.0. {'loss': 0.0023, 'episode_len': 499.0}
10500 steps has passed since last update. Now update the parameter of the behavior policy. Current step: 126000
10500 steps has passed since last update. Now update the parameter of the behavior policy. Current step: 136500
10500 steps has passed since last update. Now update the parameter of the behavior policy. Current step: 147000
(785.0s,+139.4s)	Iteration 300, current mean episode reward is -21.0. {'loss': 0.002, 'episode_len': 499.0}
10500 steps has passed since last update. Now update the parameter of the behavior policy. Current step: 157500
10500 steps has passed since last update. Now update the parameter of the behavior policy. Current step: 168000
(898.4s,+113.3s)	Iteration 350, current mean episode reward is -21.0. {'loss': 0.0005, 'episode_len': 499.0}
```

```
10500 steps has passed since last update. Now update the parameter of the behavior policy. Current step: 178500
10500 steps has passed since last update. Now update the parameter of the behavior policy. Current step: 189000
10500 steps has passed since last update. Now update the parameter of the behavior policy. Current step: 199500
(1038.2s,+139.8s)       Iteration 400, current mean episode reward is -21.0. {'loss': 0.0022, 'episode_len': 499.0}
10500 steps has passed since last update. Now update the parameter of the behavior policy. Current step: 210000
10500 steps has passed since last update. Now update the parameter of the behavior policy. Current step: 220500
(1200.5s,+162.4s)       Iteration 450, current mean episode reward is -18.8. {'loss': 0.0014, 'episode_len': 499.0}
In 450 iteration, current mean episode reward -18.800 is greater than reward threshold -20. Congratulation! Now we exit t
```

In [35…

```python
# [optional] If you have train the agent in Pont-ram-v0, please save the weights so that
# we can restore it. Please include the pong-agent.pkl into the zip.

import pickle
with open("pong-agent.pkl", "wb") as f:
    pickle.dump(pytorch_trainer2.network.state_dict(), f)
```

In [36…

```python
print("Average episode reward for your Pytorch agent in Pong-ram-v0: ",
      pytorch_trainer2.evaluate(1, render=True))
```

```
Average episode reward for your Pytorch agent in Pong-ram-v0:  -16.0
```

Conclusion and Discussion

In this assignment, we learn how to build several function approximation algorithm, how to implement basic gradient descent methods and how to use pytorch.

It's OK to leave the following cells empty. In the next markdown cell, you can write whatever you like. Like the suggestion on the course, the confusing problems in the assignments, and so on.

If you want to do more investigation, feel free to open new cells via `Esc + B` after the next cells and write codes in it, so that you can reuse some result in this notebook. Remember to write sufficient comments and documents to let others know what you are doing.

Following the submission instruction in the assignment to submit your assignment to our staff. Thank you!

.

In [23…

```python
class PytorchModel2(nn.Module):
    def __init__(self, input_shape, num_actions):
        super(PytorchModel2, self).__init__()
        # [TODO] Build a sequential model with two layers.
        # The first hidden layer has 100 hidden nodes, followed by
        # a ReLU activation function.
        # The second output layer take the activation vector, who has
        # 100 elements, as input and return the action values.
        # So the return values is a vector with num_actions elements.
        self.action_value = nn.Sequential(
            nn.Linear(input_shape[0], 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Linear(64, 64),
            nn.ReLU(),
            nn.Linear(64, 64),
            nn.ReLU(),
            nn.Linear(64, 64),
            nn.ReLU(),
            nn.Linear(64, 32),
            nn.ReLU(),
            nn.Linear(32, 32),
            nn.ReLU(),
            nn.Linear(32, 16),
            nn.ReLU(),
            nn.Linear(16, num_actions),
        )


    def forward(self, obs):
        return self.action_value(obs)
```

In [24…

```python
class DQNTrainer2():
    def __init__(self, config):
        self.config = merge_config(config, pytorch_config)
        # Create the environment
        self.env_name = self.config['env_name']
        self.env = gym.make(self.env_name)
        if self.env_name == "Pong-ram-v0":
            self.env = wrap_deepmind_ram(self.env)

        # We set self.obs_dim to the number of possible observation
        # if observation space is discrete, otherwise the number
        # of observation's dimensions. The same to self.act_dim.
        if isinstance(self.env.observation_space, gym.spaces.box.Box):
```

```
            assert len(self.env.observation_space.shape) == 1
            self.obs_dim = self.env.observation_space.shape[0]
            self.discrete_obs = False
        elif isinstance(self.env.observation_space,
                        gym.spaces.discrete.Discrete):
            self.obs_dim = self.env.observation_space.n
            self.discrete_obs = True
        else:
            raise ValueError("Wrong observation space!")

        if isinstance(self.env.action_space, gym.spaces.box.Box):
            assert len(self.env.action_space.shape) == 1
            self.act_dim = self.env.action_space.shape[0]
        elif isinstance(self.env.action_space, gym.spaces.discrete.Discrete):
            self.act_dim = self.env.action_space.n
        else:
            raise ValueError("Wrong action space!")

        # Apply the random seed
        self.seed = self.config["seed"]
        np.random.seed(self.seed)
        self.env.seed(self.seed)
        self.learning_rate = self.config["learning_rate"]
        self.max_episode_length = self.config["max_episode_length"]
        self.gamma = self.config["gamma"]
        self.n = self.config["n"]

        self.memory = ExperienceReplayMemory(self.config["memory_size"])
        self.learn_start = self.config["learn_start"]
        self.batch_size = self.config["batch_size"]
        self.target_update_freq = self.config["target_update_freq"]
        self.clip_norm = self.config["clip_norm"]
        self.eps = self.config['eps']
        self.step_since_update = 0
        self.total_step = 0

        self.initialize_parameters()

    def initialize_parameters(self):
        input_shape = self.env.observation_space.shape

        # [TODO] Initialize two network using PytorchModel class
        self.network = PytorchModel2((self.obs_dim,),  self.act_dim)

        self.network.eval()
        self.network.share_memory()

        # [TODO] Initialize target network then copy the weight
        # of original network to it. So you should
        # put the weights of self.network into self.target_network.
        self.target_network = PytorchModel2((self.obs_dim,), self.act_dim)
        self.target_network.load_state_dict(self.network.state_dict())


        self.target_network.eval()

        # Build Adam optimizer and MSE Loss.
        # [TODO] Uncomment next few lines
        self.optimizer = torch.optim.Adam(
             self.network.parameters(), lr=self.learning_rate
        )
        self.loss = nn.MSELoss()


    def compute_values(self, processed_state):
        """Compute the value for each potential action. Note that you
        should NOT preprocess the state here."""
        # [TODO] Convert the output of neural network to numpy array
        values = self.network(processed_state).detach().numpy()

        return values

    def train(self):
        s = self.env.reset()
        processed_s = self.process_state(s)
        act = self.compute_action(processed_s)
        stat = {"loss": []}

        for t in range(self.max_episode_length):
            next_state, reward, done, _ = self.env.step(act)
            next_processed_s = self.process_state(next_state)

            # Push the transition into memory.
            self.memory.push(
                (processed_s, act, reward, next_processed_s, done)
            )

            processed_s = next_processed_s
            act = self.compute_action(next_processed_s)
            self.step_since_update += 1
            self.total_step += 1

            if done:
                break

            if t % self.config["learn_freq"] != 0:
                # It's not necessary to update in each step.
                continue
```

```python
            if len(self.memory) < self.learn_start:
                continue
            elif len(self.memory) == self.learn_start:
                print("Current memory contains {} transitions, "
                      "start learning!".format(self.learn_start))

            batch = self.memory.sample(self.batch_size)

            # Transform a batch of state / action / .. into a tensor.
            state_batch = to_tensor(
                np.stack([transition[0] for transition in batch])
            )
            action_batch = to_tensor(
                np.stack([transition[1] for transition in batch])
            )
            reward_batch = to_tensor(
                np.stack([transition[2] for transition in batch])
            )
            next_state_batch = torch.stack(
                [transition[3] for transition in batch]
            )
            done_batch = to_tensor(
                np.stack([transition[4] for transition in batch])
            )

            with torch.no_grad():
                # [TODO] Compute the values of Q in next state in batch.
                Q_t_plus_one = torch.max(self.target_network(next_state_batch).detach(), 1)[0]

                assert isinstance(Q_t_plus_one, torch.Tensor)
                assert Q_t_plus_one.dim() == 1

                # [TODO] Compute the target value of Q in batch.
                Q_target = (reward_batch + (1 - done_batch) * self.gamma * Q_t_plus_one).reshape(self.batch_size,)

                assert Q_target.shape == (self.batch_size,)

            # [TODO] Collect the Q values in batch.
            # Hint: Remember to call self.network.train()
            #  before you get the Q value from self.network(state_batch),
            #  otherwise the graident will not be recorded by pytorch.
            self.network.train()
            Q_t = self.network(state_batch).gather(1, action_batch.reshape(self.batch_size,1).long()).squeeze()


            assert Q_t.shape == Q_target.shape

            # Update the network
            self.optimizer.zero_grad()
            loss = self.loss(input=Q_t, target=Q_target)
            loss_value = loss.item()
            stat['loss'].append(loss_value)
            loss.backward()

            # [TODO] Gradient clipping. Uncomment next line
            nn.utils.clip_grad_norm_(self.network.parameters(), self.clip_norm)


            self.optimizer.step()
            self.network.eval()

        if len(self.memory) >= self.learn_start and \
                self.step_since_update > self.target_update_freq:
            print("{} steps has passed since last update. Now update the"
                  " parameter of the behavior policy. Current step: {}".format(
                self.step_since_update, self.total_step
            ))
            self.step_since_update = 0
            # [TODO] Copy the weights of self.network to self.target_network.
            self.target_network.load_state_dict(self.network.state_dict())

            self.target_network.eval()

        return {"loss": np.mean(stat["loss"]), "episode_len": t}

    def process_state(self, state):
        return torch.from_numpy(state).type(torch.float32)

    def compute_action(self, processed_state, eps=None):
        """Compute the action given the state. Note that the input
        is the processed state."""

        values = self.compute_values(processed_state)
        assert values.ndim == 1, values.shape

        if eps is None:
            eps = self.eps

        # [TODO] Implement the epsilon-greedy policy here. We have `eps`
        #  probability to choose a uniformly random action in action_space,
        #  otherwise choose action that maximizes the values.
        # Hint: Use the function of self.env.action_space to sample random
        # action.
        if np.random.uniform(0, 1)  <= eps:
            action = np.random.randint(self.env.action_space.n)
        else:
            action = np.argmax(values)

        return action
```

```
    def evaluate(self, num_episodes=50, *args, **kwargs):
        """Use the function you write to evaluate current policy.
        Return the mean episode reward of 50 episodes."""
        policy = lambda raw_state: self.compute_action(
            self.process_state(raw_state), eps=0.0)
        result = evaluate(policy, num_episodes, seed=self.seed,
                          env_name=self.env_name, *args, **kwargs)
        return result
```

In [ ]: