

assignment3

October 25, 2021

1 IERG 5350 Assignment 3: Value Function Approximation in RL

2021-2022 1st term, IERG 5350: Reinforcement Learning. Department of Information Engineering, The Chinese University of Hong Kong. Course Instructor: Professor ZHOU Bolei. Assignment author: PENG Zhenghao.

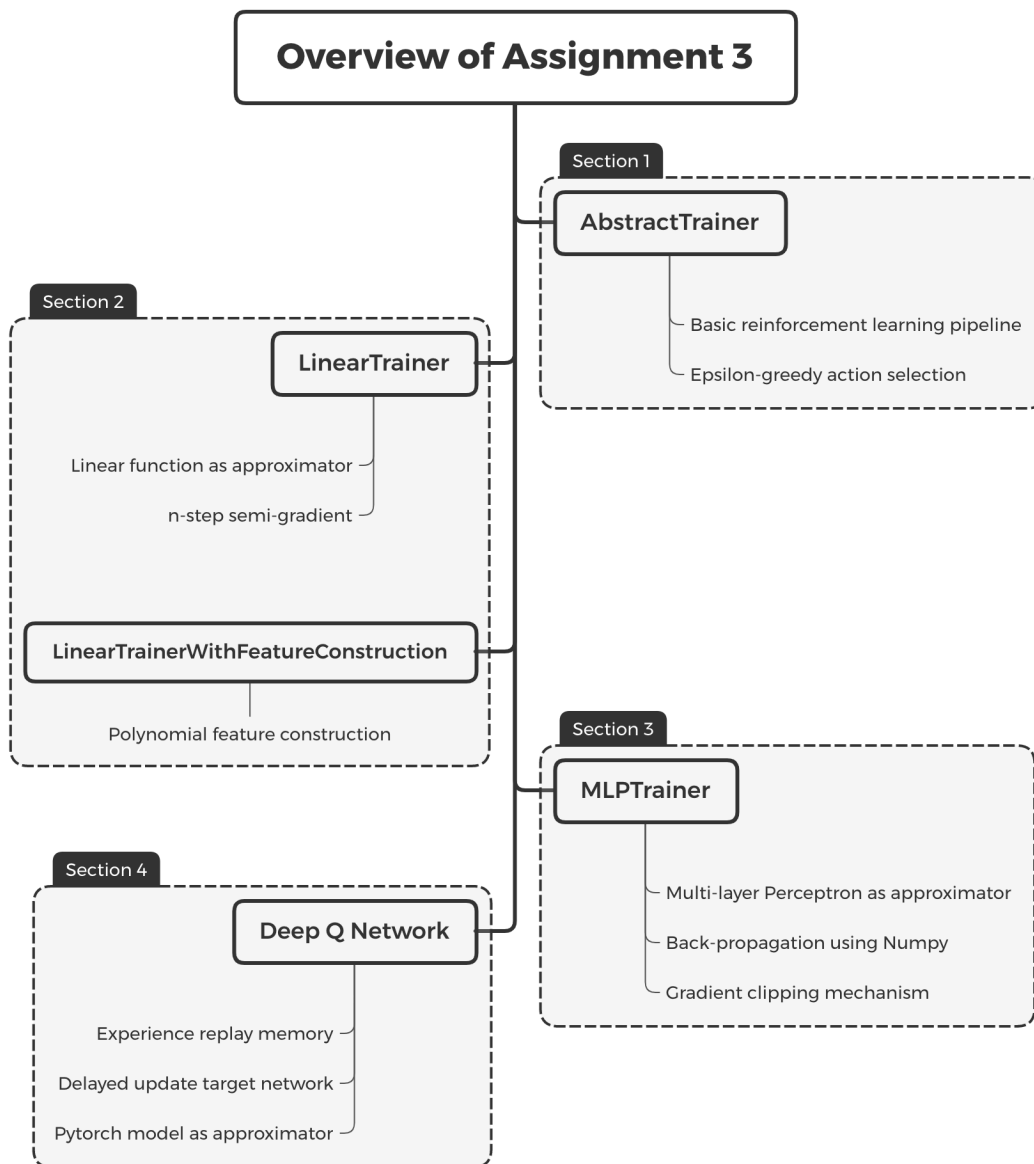
Student Name	Student ID
Wen Junjie	1155170251

Welecome to the assignment 3 of our RL course.

We will cover the following knowlededge in this assignment:

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

The following figure demonstrates the structure of this assignment.



Before starting, make sure you have installed the following packages:

1. Python 3
2. Jupyter Notebook
3. Gym, Please install via `pip install 'gym[all]'` to ensure all functionality of gym are properly set up.
4. Numpy
5. Pytorch, install via `pip install torch`. Please refer to official website <https://pytorch.org> for detailed installation guideline.
6. Opencv, install via `pip install opencv-python`

1.1 Section 1: Basic Reinforcement Learning Pipeline

(5 / 100 points)

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

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

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

def evaluate(policy, num_episodes=1, seed=0, env_name='FrozenLake8x8-v1',
            render=False, existing_env=None):
    """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.
    """
    if existing_env is None:
        env = gym.make(env_name)
        env.seed(seed)
    else:
        env = existing_env
    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)

```

[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
    """
    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

```

[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 inherit
    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.
        policy_table = self.eps * np.ones(self.env.action_space.n) / self.env.
→action_space.n
        idx_max = np.argmax(values)
        policy_table[idx_max] += 1 - self.eps
        action = np.random.choice(np.arange(self.env.action_space.n),
→p=policy_table)

        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)
        if "MetaDrive" in self.env_name:
            kwargs["existing_env"] = self.env
        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.")

```

[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:
21.722

1.2 Section 2: Linear function approximation

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

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 processes the input state and provide polynomial features that increase the effectiveness of the linear function approximation.

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

In this section, we leverage the n-step semi-gradient.

What is the "correct value" of a state-action pair in one-step TD learning? We consider it is $r_t + \gamma Q(s_{t+1}, a_{t+1})$ and thus lead to the one-step 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 of the textbook to implement this logic. Note that notation of the time step is different in this assignment and 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 as the R_{t+1} in the textbook. In the code below, we store the states, actions and rewards to lists during training. **You need to make sure the indices of these lists, namely 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 estimated 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, the final update rule of parameters is:

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

In the following code, we denote $G = \sum_{i=t}^{t+n-1} \gamma^{i-t} r_i + \gamma^n Q(s_{t+n}, a_{t+n})$ and will compute dQ/dW according to the form of the approximator.

1.2.1 Section 2.1: Basics

(30 / 100 points)

We want to approximate the state-action values, 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 :

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

Note that $W \in \mathbb{R}^{(O,A)}$ and $s \in \mathbb{R}^O$, wherein O is the observation (state) dimensions, the `self.obs_dim`, and A is the action dimension, the `self.act_dim` in the trainer. The output $v(s, W) \in \mathbb{R}^A$. Each entry to the output corresponds to one action.

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

```

[6]: # 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(scale=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.matmul(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. Your 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[-1])

            processed_s = self.process_state(next_state)
            processed_states.append(processed_s)
            rewards.append(reward)
            if done:
                break
            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 accumulated 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 discount factor  $\gamma^n$  to
        # the second part of G if applicable.
        valid_states = processed_states[-n-1:]
        valid_rewards = rewards[-n:]

        q = sum([reward * self.gamma ** i for i, reward in enumerate(valid_rewards)])
        G = np.zeros((self.act_dim, 1))
        action = actions[-n-1]
        G[action,0] = q

    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.
        next_n_values = self.compute_values(valid_states[-1])
        G[action, 0] += self.gamma ** n * next_n_values[action]

    # 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 proper value of loss_grad, denoting the gradient
    # of the MSE w.r.t. the output of the linear function.
    current_state = valid_states[0]
    current_q = self.compute_values(current_state)[: , None]
    loss_grad = (G - current_q)

    # [TODO] compute the value of value_grad, denoting the gradient of
    # the output of the linear function w.r.t. the parameters.
    value_grad = current_state[:,None]

    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.matmul(value_grad, loss_grad.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

```

[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.

[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).

```

(0.1s,+0.1s)    Iteration 0, current mean episode reward is 10.34.
(0.9s,+0.8s)    Iteration 1000, current mean episode reward is 10.4.
(1.6s,+0.8s)    Iteration 2000, current mean episode reward is 10.8.
(2.4s,+0.8s)    Iteration 3000, current mean episode reward is 10.34.
(3.2s,+0.8s)    Iteration 4000, current mean episode reward is 10.66.
(4.0s,+0.8s)    Iteration 5000, current mean episode reward is 10.96.
(4.8s,+0.8s)    Iteration 6000, current mean episode reward is 10.6.
(5.5s,+0.8s)    Iteration 7000, current mean episode reward is 10.7.
(6.4s,+0.8s)    Iteration 8000, current mean episode reward is 10.82.
(7.2s,+0.8s)    Iteration 9000, current mean episode reward is 11.4.
(8.0s,+0.8s)    Iteration 10000, current mean episode reward is 11.1.

```

```

[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: 11.0

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

1.2.2 Section 2.2: Linear Model with Feature Construction

(15 / 100 points)

```

[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  $(order+1)^k$  as the output,
    """

```

wherein k is the dimensions of the state.

For example:

When the state is $[2, 3, 4]$ (so $k=3$),
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
].
```

We have $(1+1)^3=8$ output dimensions.

Note: it is not necessary to follow the ascending order.

"""

[TODO] finish this function.

```
output = []
```

```
k = len(sequence)
```

```
if k > 1:
```

```
    v = sequence[0]
```

```
    seq = [v**i for i in range(order+1)]
```

```
    remain_seqs = polynomial_feature(sequence[1:], order=order)
```

```
    for s in seq:
```

```
        output += [s*r for r in remain_seqs]
```

```
elif k == 1:
```

```
    v = sequence[0]
```

```
    return [v**i for i in range(order+1)]
```

```
return output
```

```
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

```

[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 more_
↳ than 20+."

# This cell should be finished within 10 minutes.

```

Initialize parameters with shape: (4, 2).

Initialize parameters with shape: (16, 2).

(0.3s,+0.3s) Iteration 0, current mean episode reward is 59.72.

(9.8s,+9.5s) Iteration 1000, current mean episode reward is 154.78.

(26.6s,+16.8s) Iteration 2000, current mean episode reward is 182.72.

(44.0s,+17.3s) Iteration 3000, current mean episode reward is 191.32.

(61.8s,+17.8s) Iteration 4000, current mean episode reward is 184.18.

(79.5s,+17.8s) Iteration 5000, current mean episode reward is 191.82.

(97.2s,+17.7s) Iteration 6000, current mean episode reward is 195.72.

In 6000 iteration, current mean episode reward 195.720 is greater than reward

threshold 195.0. Congratulation! Now we exit the training process.

```
[12]: # Run this cell without modification

# You should see a pop up window which display the movement of the cart and
# ↪ pole.
print(
    "In CartPole-v0, the average episode reward for the value estimator "
    "with feature construction is: ",
    linear_fc_trainer.evaluate(1, render=True))
```

In CartPole-v0, the average episode reward for the value estimator with feature construction is: 200.0

1.3 Section 3: Multi-layer Perceptron as the approximator

In this section, you are required to implement a Multi-layer Perceptron (MLP) as the value estimator using purely Numpy package.

The differences between MLP and linear function are (1) MLP has a hidden layer which increase its representation capacity and (2) MLP utilizes activation function after the output of each layer such that non-linearity is introduced.

Consider a MLP with one hidden layer containing 100 neurons and an activation function $f()$. We call the layer that receives the state as input and output the activations as the **hidden layer**. The next layer that accepts the activations as input and produces the estimated values is the **output layer**.

The activations of the hidden layer is:

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

It is obvious that the activations is a 100-length vector.

The output estimated values are:

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

wherein W_h, W_o are the parameters of the 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. The unbounded values lead to the *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 specified by `config["clip_norm"]` in our implementation. Gradient clipping bounds the gradient norm to c if the norm of original gradient is greater than c . You need to implement this mechanism in function `apply_gradient` in the following cell.

```
[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(scale=std, size=(self.
→obs_dim, self.hidden_dim))
        self.output_parameters = np.random.normal(scale=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.matmul(self.output_parameters.T, activation)

        return values

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

```

        """
        activation = np.matmul(self.hidden_parameters.T, processed_state)

        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.
        valid_states = processed_states[-n-1:]
        valid_rewards = rewards[-n:]

        q = sum([reward * self.gamma ** i for i, reward in
        ↪ enumerate(valid_rewards)])
        G = np.zeros((self.act_dim, 1))
        action = actions[-n-1]
        G[action,0] = q

        if tau + n < T:
            next_n_values = self.compute_values(valid_states[-1])
            G += self.gamma ** n * next_n_values[:, None]

        # 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.
        # current_state = processed_states[tau]

        loss_grad = np.zeros((self.act_dim, 1)) # [act_dim, 1]
        # [TODO] compute loss_grad
        current_state = valid_states[0]
        current_q = self.compute_values(current_state)[:, None]
        loss_grad = G - current_q

        # [TODO] compute the gradient of output layer parameters
        output_gradient = np.matmul(np.matmul(self.hidden_parameters.T,
        ↪ current_state[:, None]), loss_grad.T)

        # [TODO] compute the gradient of hidden layer parameters
        hidden_gradient = np.matmul(np.matmul(current_state[:,None], loss_grad.
        ↪ T), self.output_parameters.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 gradients 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 mechanism
    # Hint: when the old gradient has norm less than 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"]
        output_gradient = output_gradient * clip_norm / max(np.
→max(abs(output_gradient)), clip_norm)
        hidden_gradient = hidden_gradient * clip_norm / max(np.
→max(abs(hidden_gradient)), clip_norm)

    # [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

```

[14]: # Run this cell without modification

```

print("Now let's see what happens if gradient clipping is not enabled!\n")
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("\nWe 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 gradient clipping is not enable!

```

(0.1s,+0.1s)    Iteration 0, current mean episode reward is 35.66.
(1.0s,+0.8s)    Iteration 100, current mean episode reward is 73.34.
(2.2s,+1.3s)    Iteration 200, current mean episode reward is 130.96.
(4.7s,+2.4s)    Iteration 300, current mean episode reward is 194.48.
(7.4s,+2.7s)    Iteration 400, current mean episode reward is 193.74.
(9.8s,+2.4s)    Iteration 500, current mean episode reward is 190.56.
(12.3s,+2.5s)   Iteration 600, current mean episode reward is 193.16.
(14.8s,+2.5s)   Iteration 700, current mean episode reward is 194.46.
(17.3s,+2.5s)   Iteration 800, current mean episode reward is 196.4.
In 800 iteration, current mean episode reward 196.400 is greater than reward
threshold 195.0. Congratulation! Now we exit the training process.

```

We expect to see bad performance (<195). The performance without gradient clipping: 182.88.

Try next cell to see the impact of gradient clipping.

```

[15]: # Run this cell without modification

print("Now let's see what happen if gradient clipping is enable!\n")
mlp_trainer, _ = run(MLPTrainer, dict(
    max_iteration=3000,
    evaluate_interval=100,
    parameter_std=0.01,
    learning_rate=0.01,
    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 gradient clipping is enable!

(0.2s,+0.2s) Iteration 0, current mean episode reward is 55.22.
(2.3s,+2.2s) Iteration 100, current mean episode reward is 181.96.
(5.2s,+2.9s) Iteration 200, current mean episode reward is 198.46.
In 200 iteration, current mean episode reward 198.460 is greater than reward threshold 195.0. Congratulation! Now we exit the training process.

```
[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!

1.4 Section 4: Implement Deep Q Learning in Pytorch

(50 / 100 points)

In this section, we will get familiar with the basic logic of pytorch, a powerful Deep Learning framework, which lays the ground for future tasks.

We will implement a MLP similar to the one in Section 3 using Pytorch. Before start, you need to make sure the pytorch is properly installed.

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

1. quickstart: https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html
2. tutorial on RL: https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

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

1. DQN requires an experience replay memory (buffer) to store the transitions.

A replay memory is implemented in the following `ExperienceReplayMemory` class. It can store 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.

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

2. DQN is an off-policy algorithm and computes TD error in a different way compared to Sarsa.

In Sarsa, the TD error is computed as:

$$TD = r_t + \gamma \mathbb{E}_{a_{t+1} \sim \pi} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$$

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

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

3. DQN delayed updates the target network. This is another difference even compared to the traditional Q learning.

DQN maintains another neural network called the *target network*. The target network has identical structure of the Q network. After certain amount of steps, the target network replaces its parameters by the latest parameters of the Q network. The update of target network is much less frequent than the update of the Q network since 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:*

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

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

In the engineering perspective, 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. 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 from it.
3. Thank to the replay memory and pytorch, DQN can be updated through batches of transitions. Therefore 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**.

```
[17]: # Solve the TODOs and remove `pass`

from collections import deque
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)

```

```

[18]: # Solve the TODOs and remove `pass`
import torch.nn as nn
import torch.nn.functional as F

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.input_shape = input_shape[0]
        self.num_actions = num_actions

        self.action_value = nn.Sequential(
            nn.Linear(self.input_shape, 100),
            nn.ReLU(True),
            nn.Linear(100, self.num_actions)
        )

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

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

```

```

[19]: # 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(input_shape, 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(input_shape, 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)
    values = values.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 = self.target_network(next_state_batch)
        Q_t_plus_one = Q_t_plus_one.max(dim=1).values

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

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

        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.
    actions = action_batch.squeeze().detach().numpy()
    Q_t = self.network(state_batch)[np.arange(actions.shape[0]),
    ↪actions]
```

```

        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)

```

```

[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!")

```

```

/home/wenjj/anaconda3/envs/cluster/lib/python3.8/site-
packages/numpy/core/fromnumeric.py:3372: RuntimeWarning: Mean of empty slice.
    return _methods._mean(a, axis=axis, dtype=dtype,
/home/wenjj/anaconda3/envs/cluster/lib/python3.8/site-
packages/numpy/core/_methods.py:170: RuntimeWarning: invalid value encountered
in double_scalars
    ret = ret.dtype.type(ret / rcount)

```

Now your codes should be bug-free.

```

(0.1s,+0.1s)    Iteration 0, current mean episode reward is 11.8. {'loss': nan,
'episode_len': 8.0}

```

Current memory contains 100 transitions, start learning!

```

(0.3s,+0.2s)    Iteration 10, current mean episode reward is 10.78. {'loss':
0.0806, 'episode_len': 19.0}

```

```

(0.6s,+0.3s)    Iteration 20, current mean episode reward is 10.94. {'loss':
0.0259, 'episode_len': 8.0}

```

Test passed!

[23]: *# 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 within 1000 iterations." \
    "But it achieve {} reward in evaluation.".format(195.0, reward)

# Should solve the task in 10 minutes

```

```

(0.1s,+0.1s)    Iteration 0, current mean episode reward is 14.72. {'loss': nan,
'episode_len': 10.0}
(0.3s,+0.2s)    Iteration 10, current mean episode reward is 15.04. {'loss':
nan, 'episode_len': 11.0}
(0.5s,+0.2s)    Iteration 20, current mean episode reward is 15.02. {'loss':
nan, 'episode_len': 10.0}
(0.6s,+0.2s)    Iteration 30, current mean episode reward is 15.2. {'loss': nan,
'episode_len': 16.0}
(0.8s,+0.2s)    Iteration 40, current mean episode reward is 15.3. {'loss': nan,
'episode_len': 13.0}
(0.9s,+0.2s)    Iteration 50, current mean episode reward is 15.1. {'loss': nan,
'episode_len': 18.0}
(1.1s,+0.2s)    Iteration 60, current mean episode reward is 15.22. {'loss':
nan, 'episode_len': 14.0}
Current memory contains 1000 transitions, start learning!
1001 steps has passed since last update. Now update the parameter of the
behavior policy. Current step: 1001
(1.3s,+0.2s)    Iteration 70, current mean episode reward is 9.56. {'loss':
0.0639, 'episode_len': 7.0}
(1.6s,+0.3s)    Iteration 80, current mean episode reward is 9.9. {'loss': 0.1,
'episode_len': 9.0}
(1.8s,+0.2s)    Iteration 90, current mean episode reward is 9.7. {'loss':
0.0906, 'episode_len': 8.0}
(2.1s,+0.2s)    Iteration 100, current mean episode reward is 9.2. {'loss':
0.0628, 'episode_len': 8.0}
(2.3s,+0.3s)    Iteration 110, current mean episode reward is 10.06. {'loss':
0.0773, 'episode_len': 15.0}
(2.6s,+0.2s)    Iteration 120, current mean episode reward is 9.8. {'loss':
0.0641, 'episode_len': 11.0}
(2.8s,+0.2s)    Iteration 130, current mean episode reward is 9.64. {'loss':
0.0965, 'episode_len': 10.0}
(3.1s,+0.2s)    Iteration 140, current mean episode reward is 9.96. {'loss':
0.0723, 'episode_len': 10.0}
(3.3s,+0.2s)    Iteration 150, current mean episode reward is 9.78. {'loss':
0.0531, 'episode_len': 10.0}
(3.5s,+0.2s)    Iteration 160, current mean episode reward is 9.82. {'loss':
0.047, 'episode_len': 8.0}
1006 steps has passed since last update. Now update the parameter of the
behavior policy. Current step: 2007
(3.8s,+0.3s)    Iteration 170, current mean episode reward is 9.94. {'loss':
0.0814, 'episode_len': 10.0}
(4.0s,+0.2s)    Iteration 180, current mean episode reward is 10.04. {'loss':
0.0552, 'episode_len': 9.0}
(4.3s,+0.2s)    Iteration 190, current mean episode reward is 10.12. {'loss':
0.0603, 'episode_len': 10.0}
(4.6s,+0.3s)    Iteration 200, current mean episode reward is 10.08. {'loss':
0.0671, 'episode_len': 10.0}
(4.8s,+0.3s)    Iteration 210, current mean episode reward is 10.32. {'loss':

```

```

0.0224, 'episode_len': 9.0}
(5.1s,+0.3s)    Iteration 220, current mean episode reward is 10.46. {'loss':
0.0343, 'episode_len': 9.0}
(5.4s,+0.3s)    Iteration 230, current mean episode reward is 12.08. {'loss':
0.0425, 'episode_len': 9.0}
(5.6s,+0.3s)    Iteration 240, current mean episode reward is 10.6. {'loss':
0.0494, 'episode_len': 8.0}
(5.9s,+0.2s)    Iteration 250, current mean episode reward is 10.3. {'loss':
0.0384, 'episode_len': 13.0}
1005 steps has passed since last update. Now update the parameter of the
behavior policy. Current step: 3012
(6.3s,+0.4s)    Iteration 260, current mean episode reward is 18.46. {'loss':
0.095, 'episode_len': 27.0}
(6.7s,+0.5s)    Iteration 270, current mean episode reward is 14.66. {'loss':
0.0459, 'episode_len': 11.0}
(7.3s,+0.5s)    Iteration 280, current mean episode reward is 17.52. {'loss':
0.0469, 'episode_len': 10.0}
(7.7s,+0.5s)    Iteration 290, current mean episode reward is 20.52. {'loss':
0.0829, 'episode_len': 10.0}
(8.1s,+0.4s)    Iteration 300, current mean episode reward is 11.68. {'loss':
0.0629, 'episode_len': 10.0}
1002 steps has passed since last update. Now update the parameter of the
behavior policy. Current step: 4014
(8.6s,+0.5s)    Iteration 310, current mean episode reward is 14.86. {'loss':
0.1855, 'episode_len': 41.0}
(10.1s,+1.5s)   Iteration 320, current mean episode reward is 50.8. {'loss':
0.1091, 'episode_len': 107.0}
1001 steps has passed since last update. Now update the parameter of the
behavior policy. Current step: 5015
(11.7s,+1.7s)   Iteration 330, current mean episode reward is 105.06. {'loss':
0.1135, 'episode_len': 134.0}
1011 steps has passed since last update. Now update the parameter of the
behavior policy. Current step: 6026
(15.2s,+3.5s)   Iteration 340, current mean episode reward is 174.1. {'loss':
0.0913, 'episode_len': 178.0}
1178 steps has passed since last update. Now update the parameter of the
behavior policy. Current step: 7204
(19.7s,+4.5s)   Iteration 350, current mean episode reward is 187.84. {'loss':
0.1253, 'episode_len': 190.0}
1188 steps has passed since last update. Now update the parameter of the
behavior policy. Current step: 8392
1143 steps has passed since last update. Now update the parameter of the
behavior policy. Current step: 9535
(23.8s,+4.1s)   Iteration 360, current mean episode reward is 135.16. {'loss':
0.1972, 'episode_len': 199.0}
1121 steps has passed since last update. Now update the parameter of the
behavior policy. Current step: 10656
1200 steps has passed since last update. Now update the parameter of the

```

behavior policy. Current step: 11856
(29.0s,+5.2s) Iteration 370, current mean episode reward is 195.9. {'loss': 0.3672, 'episode_len': 199.0}
In 370 iteration, current mean episode reward 195.900 is greater than reward threshold 195.0. Congratulation! Now we exit the training process.

```
[24]: # 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

1.5 Conclusion and Discussion

In this assignment, we learn how to build several value approximation algorithms. We also get familiar with the basic gradient descent methods and Pytorch.

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

[]: