110/120

ECE5984: Reinforcement Learning

Assignment #4

Nima Mohammadi

nimamo@vt.edu

Problem 1.
In this problem, we study the convergence of policy iteration for state-action value function Q. In particular, consider a discounted cost MDP with finite state and finite action spaces. Given a fixed stationary policy μ let Q_μ satisfies

$$Q_\mu = T_\mu\left(Q_\mu\right)$$

where

$$T_\mu(Q)(s,a) = \mathbb{E}[r(s,a)] + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}(a) Q\left(s', \mu\left(s'\right)\right)$$

Consider the following policy iteration algorithm:

1. Start with a stationary policy μ_0. Set $k = 0$
2. Compute Q_{μ_k} by solving
$$Q_{\mu_k} = T_{\mu_k}\left(Q_{\mu_k}\right)$$
3. Find a new policy μ_{k+1} s.t.
$$\mu_{k+1}(s) = \arg\max_a Q(s,a)$$
4. Increase k by 1 and go to step (2)

Show that μ_k converges to an optimal stationary policy.

$$T_{\mu_{k+1}}\left(Q_{\mu_k}\right)(s,a) \geq T_{\mu_k}\left(Q_{\mu_k}\right)(s,a)$$
$$T_{\mu_{k+1}}\left(Q_{\mu_k}\right)(s,a) \geq Q_{\mu_k}(s,a)$$

With n times applying $T_{\mu_{k+1}}$, as n approaches infinity, we will have $Q_{\mu_{k+1}} \geq Q_{\mu_k}$, which along finite stationary policies, indicates convergence. To show the optimality of μ_k, and consequently Q_{μ_k}, assume $Q_{\mu_{k+1}} = Q_{\mu_k}$,

$$\begin{aligned}
Q_{\mu_{k+1}} &= \mathbb{E}[r(s,a)] + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}(a) Q_{\mu_{k+1}}\left(s, \mu_{k+1}(s')\right) \\
&= \mathbb{E}[r(s,a)] + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}(a) Q_{\mu_k}\left(s, \mu_{k+1}(s')\right) \\
&= \mathbb{E}[r(s,a)] + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}(a) \max_{a'} Q_{\mu_k}(s,a')
\end{aligned}$$

Problem 2. Consider a discounted cost MDP with 6 states $\{1,2,3,4,5,6\}$. Given a fixed policy, we are interested in approximating its discounted cost vector V as follows:

$$V = \Phi\theta, \quad \Phi = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \end{bmatrix}, \quad \theta = \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix}$$

Such a scheme is called hard aggregation since states $\{1,2,3\}$ are assumed to have the same discounted cost and the same holds for states $\{4,5,6\}$. As studied in class, given a vector V we want to solve the weighted least squares problem

$$\min_{\theta} \|V - \Phi\theta\|_W^2$$

where $\|x\|_W^2 = x^T W x$ and W is some diagonal matrix with positive entries $\{w_1, w_2, \ldots, w_6\}$. Show that the optimal solution to the above problem satisfies

$$\theta_i^* = \sum_{j=1}^{6} a_{ij} V_j, \quad i = 1,2$$

where the vectors $\sum_{j=1}^{6} a_{1j} = \sum_{j=1}^{6} a_{2j} = 1$. Also, identify which are the nonzero components in each of the two probability vectors $a_1 = [a_{11}, \ldots, a_{16}]^T$ and $a_2 = [a_{21}, \ldots, a_{26}]^T$.

$$\begin{aligned} \theta^* &= \arg\min_{\theta} \|V - \Phi\theta\|_W^2 \\ &= \arg\min_{\theta} (V - \Phi\theta)^\top W (V - \Phi\theta) \end{aligned}$$

$$\begin{aligned} (V - \Phi\theta)^\top W (V - \Phi\theta) &= V^\top W V - V^\top W \Phi\theta - \theta^\top \Phi^\top W V + \theta^\top \Phi^\top W \Phi\theta \\ &= \theta^\top \Phi^\top W \Phi\theta - 2V^\top W \Phi\theta + V^\top W V \\ &= 2\Phi^\top W \Phi\theta - 2V^\top W \Phi \end{aligned}$$

Differentiating with respect to θ and setting it to zero:

$$\nabla_\theta \|V - \Phi\theta\|_W^2 = 2\Phi^\top W \Phi\theta - 2V^\top W \Phi = 0$$

Then,

$$\theta^* = \left(\Phi^\top W \Phi\right)^{-1} \Phi^\top W V$$

The solution satisfies

$$\theta^* = \sum_{j=1}^{6} a_{ij} J_j, i = 1,2$$

where we have

$$\begin{aligned} a_1 &= \begin{bmatrix} \frac{w_1}{\sum_{i=1}^3 w_i} & \frac{w_2}{\sum_{i=1}^3 w_i} & \frac{w_3}{\sum_{i=1}^3 w_i} & 0 & 0 & 0 \end{bmatrix}^\top \\ a_2 &= \begin{bmatrix} 0 & 0 & 0 & \frac{w_4}{\sum_{i=4}^6 w_i} & \frac{w_5}{\sum_{i=4}^6 w_i} & \frac{w_6}{\sum_{i=4}^6 w_i} \end{bmatrix}^\top \end{aligned}$$

Problem 3.

An individual desires to sell her car. A new offer is made to her every day and she must decide whether to accept or not the offer. Once rejected, the offer is lost. Suppose that successive offers are independent of each other and take on the value i with probability P_i, $i = 0,1,\ldots,N$. Suppose also that the car has a maintenance cost C per day, while future costs are discounted at rate α. Let the state at time k be the corresponding offer. Answer the following questions (note in this case we have a minimization problem):

1. Using the context of MDP, determine the Bellman equation satisfied by the optimal value $V^*(i)$ for all $i \in \{0,1,\ldots,N\}$.

$$\Omega_S = \{0, 1, \cdots, N\}$$
$$\Omega_A = \{\text{'reject', 'accept'}\}$$
$$c(s = i, a) = \begin{cases} -i & \text{if } a \text{ is 'accept'} \\ C & \text{if } a \text{ is 'reject'} \end{cases}$$
$$J^*(i) = \min\left\{-i, C + \gamma \sum_{j=0}^{N} P_j J^*(j)\right\}$$

2. Based on your answer in (1), the optimal policy should have a threshold form: there exists i^* such that the individual should accept any offer i such that $i \leq i^*$ and she should reject any offer such that $i < i^*$. Give the description of i^*.

$$i^* = \min\left\{ i \text{ s.t. } -i < C + \gamma \sum_{j=0}^{N} P_j J^*(j)\right\}$$

Then the optimal policy would be to accept if $i \geq i^*$, and reject otherwise.

3. Let μ_i be the policy which accepts any offer greater than or equal to i. For this policy, find the conditional expected discounted cost given K, which is the number of rejected offers.

$$\sum_{j=0}^{K-1} \gamma^j C - \gamma^K \frac{\sum_{i'=1}^{N} i' P_{i'}}{\sum_{i'=1}^{N} P_{i'}}$$

4. Based on your answer in (3), consider the expected discounted cost by averaging over K and be more explicit about how i^* should be chosen.

The mean of the geometrically distributed k is

$$\mathbb{E}(k) = \frac{\sum_{j=0}^{i-1} P_j}{\sum_{j-0}^{N} P_j}$$

Then the expected discounted cost of μ_i can be calculated as below:

$$\sum_{j=0}^{N} P_j J_{\mu_i}(j) = \frac{C \sum_{j=0}^{i-1} P_j - \sum_{j=i}^{N} j P_j}{1 - \gamma \sum_{j=0}^{i-1} P_j}$$

$$i^* = \arg\min_i \left(\frac{C \sum_{j=0}^{i-1} P_j - \sum_{j=i}^{N} j P_j}{1 - \gamma \sum_{j=0}^{i-1} P_j} \right)$$

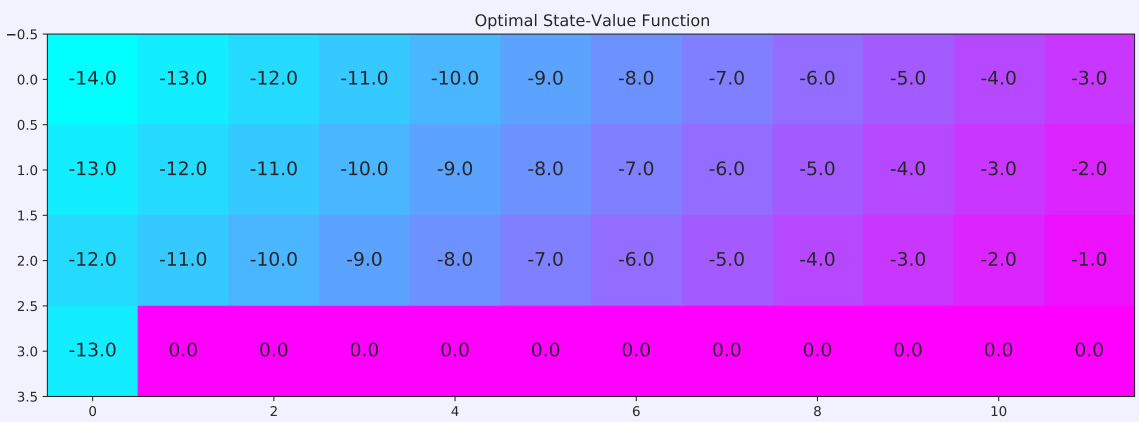
x How did you get this?

-10

Problem 4.
Write a simulation program to implement Q-learning in the tabular setting for the cliff-walking problem. In your simulation, consider a number of episodes, where each episode runs until the program terminates or at most 50 steps. You can choose your own behavior policy but here is one suggestion: the behavior policy of each new episode could be the policy returned by the last run. The initial behavior policy could be a randomized policy.
Consider the Cliff-walking problem shown in Fig. 1 and given in Homework 3. Submit your code and explain what's the optimal policy your algorithm can return? Plot a curve to show the total reward returned by each episode, i.e., the number of points should be equal to the number of episode.

The implementation code in Python can be found at the end of this section. As for the behavior policy I have tested with two different options, namely the `'random'` policy where the possible actions are selected randomly with equal probabilities, and `'last_run'` policy where the actions are selected with an ϵ-greedy scheme over the Q-table from the last episode.

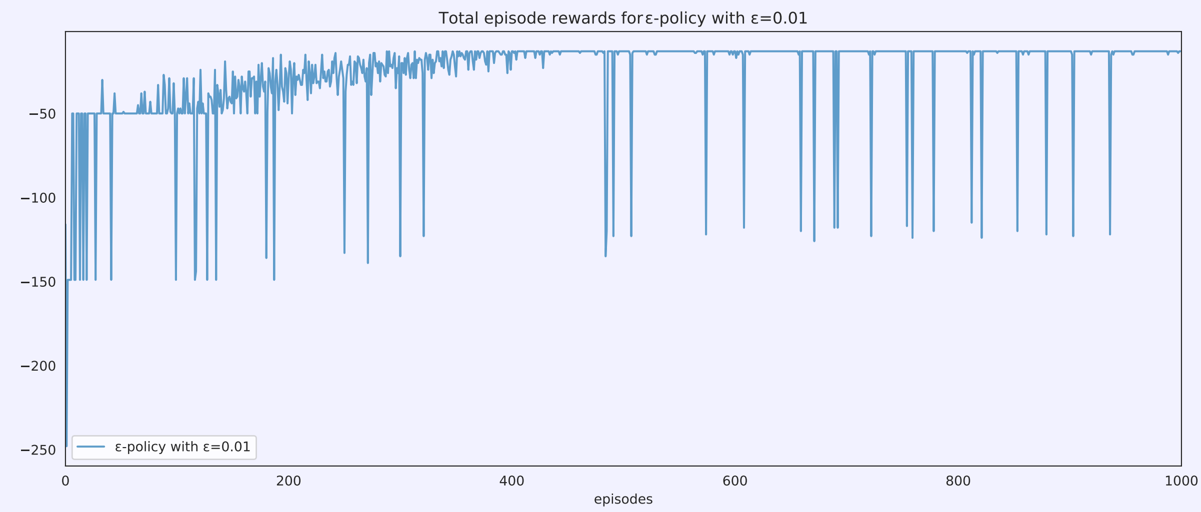
For the sake of comparison, I'll start off by showing how the optimal value function would look like:



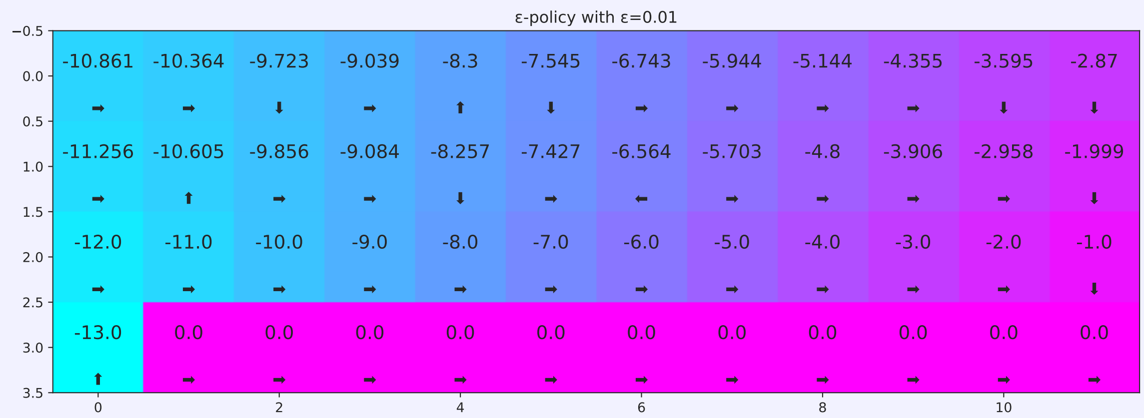
Next, let's use the random behavior policy first. In this case, the total rewards obtained through each episode are depicted below, where it does not really show any convergence.

The corresponding V-table (the maximum Q-values and corresponding actions) are depicted below. The plot shows that the algorithm was slowly approaching toward finding the optimal policy but given the rather 'dumb' behavior policy, it has not been able to visit many of the state-action pairs to get a proper estimate of true Q values.

Next, we use an ϵ-greedy behavior policy based on the last run. Notice that in my implementation, ϵ is not decaying and stays constant with a value of 10^{-2}. The total episode rewards are depicted below, indicating much better stability and convergence compared to the run with random behavior policy:



Similarly, the corresponding state-value table (the maximum Q-values and corresponding actions) are depicted below:



We see that the second experiment has reached the optimal policy. However, we can see that the experiment using the random behavior policy has also been able to identify the correct preliminary moves, but simply has not been able to converge as it had not even observed many of the states. I can attribute this to the fact, that using the second run with the behavior policy being the Q-table from the last run and using ϵ-greedy offers a better balance between exploration and exploitation, that is, making knowledgeable decisions based on prior interaction while still allowing for randomly exploring other actions.

```
N_episodes = 1000
max_episode_length = 50
gamma = 1.0
alpha = .1

configs = [
           {'behavior': 'random', 'epsilon': 'XX'},
           {'behavior': 'last_run', 'epsilon': .01}]
configs_results = []
configs_qtables = []

for conf in configs:
  behavior_policy = conf['behavior']
  epsilon = conf['epsilon']
  q_table = np.zeros([env.observation_space.n, env.action_space.n])
  rewards = []
  for ep in range(N_episodes):
    S = env.reset()
    i = 0
    done = False
    episode_reward = 0
    while ((not done) and (i < max_episode_length)):
```

```
      i += 1
      if behavior_policy == 'random':
        A = policy_uniform_sample(S)
      elif behavior_policy == 'last_run':
        if epsilon > np.random.random():
          A = policy_uniform_sample(S)
        else:
          A = np.argmax(q_table[S])
      else:
        raise Exception('Invalid behavior policy')
      S_, R, done, _ = env.step(A)
      q_table[S, A] = (1-alpha) * q_table[S, A] + alpha * (R + gamma * np.max(q_table[S_]))
      S = S_
      episode_reward += R
    rewards.append(episode_reward)
  configs_results.append(rewards)
  configs_qtables.append(q_table)
```

Problem 5.
Consider the Cartpole environment from OpenAI Gym, which can be found here:
`https://github.com/openai/gym/blob/master/gym/envs/classic_control/cartpole.py`
In this question, we will consider Q-learning with linear function approximation using Fourier basis [1]. For this problem, consider discount factor is $\gamma = 0.9$ and a behavior policy a randomized policy.

[1.] Value Function Approximation in Reinforcement Learning using the Fourier Basis. George Konidaris and Sarah Osentoski and Philip Thomas.

Questions: Write a simulation program to implement Q-learning with linear function approximation for the cart-pole problem. In your simulation, consider a number of episodes, where each episode runs until the program terminates. You can choose your own behavior policy but here is one suggestion: the behavior policy of each new episode could be the policy returned by the last run. The initial behavior policy could be a randomized policy.

Submit your code and explain what's your linear function approximation? Plot a curve to show the total reward returned by each episode, i.e., the number of points should be equal to the number of episode. Does your returned policy solve the problem, i.e., how long your program last by running that policy?

Under function approximation, θ parameters are updates as follows:

$$\theta \leftarrow \theta + \alpha\delta\nabla_\theta \widehat{Q}(s,a;\theta)$$
$$\text{where} \quad \delta := r + \gamma \max_{a' \in \mathcal{A}} \widehat{Q}\left(s', a'; \theta\right) - \widehat{Q}(s,a;\theta)$$

The Q-learning's update rule is agnostic to the choice of function class, and so in principle any differentiable and parameterized function class could be used in conjunction with the above update to learn the parameters. To this end, Konidaris et al. (2011) used Fourier basis functions whereas Mnih et al. (2015) used convolutional neural networks. With a set of basis functions $\phi_1, \ldots, \phi_n$, our approximation would look like

$$\hat{Q}(s,a) = \theta \cdot \Phi(s,a) = \sum_{i=1}^{n} \theta_i \phi(s_i, a_i)$$

Then for our goal of minimizing the TD error,

$$\min_\theta \sum_{i=0}^{n} \left(\theta \cdot \phi(s_i, a_i) - r_i - \gamma\theta \cdot \phi(s_{i+1}, a_{i+1})\right)^2$$

for each parameter θ_j, we would have

$$\frac{\partial}{\partial \theta_j} \sum_{i=0}^{n} \left(\theta \cdot \phi(s_i, a_i) - r_i - \gamma\theta \cdot \phi(s_{i+1}, a_{i+1})\right)^2$$
$$= 2\sum^{n} \left(\theta \cdot \phi(s_i, a_i) - r_i - \gamma\theta \cdot \phi(s_{i+1}, a_{i+1})\right) \phi_j(s_i, a_i)$$

And the update rule would be:

$$\theta_{j,i+1} = \theta_{j,i} + \alpha\left(\theta\cdot\phi(s_i,a_i) - r_i - \gamma\theta\cdot\phi(s_{i+1},a_{i+1})\right)\phi_j(s_i,a_i)$$

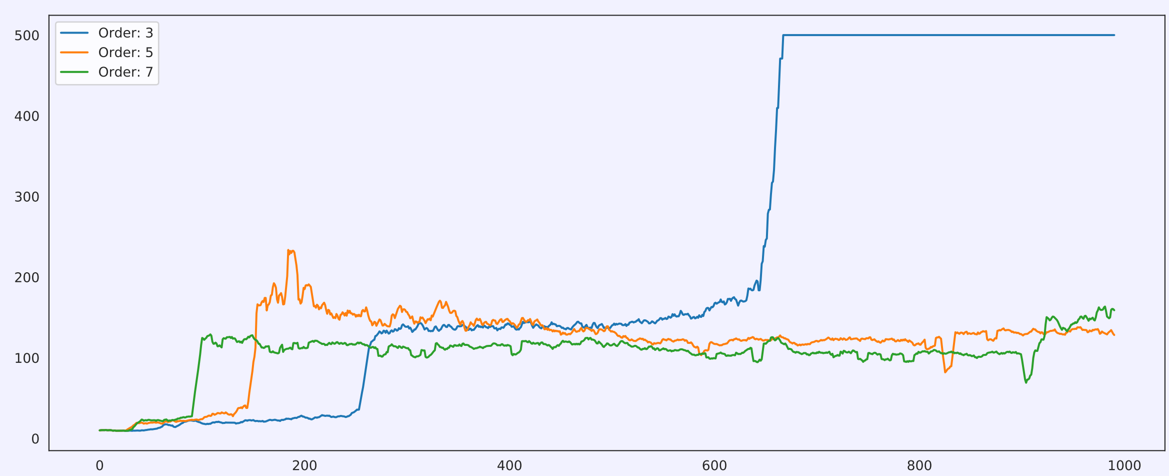
The zth-order Fourier basis for d state variables is the set of basis functions defined as

$$\phi_i(\mathbf{s}) = \cos\left(\pi\mathbf{c}^i\cdot\mathbf{s}\right)$$

where $\mathbf{c}^i = [c_1,\ldots,c_d], c_j \in \{0,\ldots,z\}, 1 \le j \le d$. Each basis function is obtained using a vector $\mathbf{c}^i$ that attaches an integer coefficient (between 0 and z, inclusive) to each variable in s (after that variable has been scaled to $[0,1]$).
Here I am using CartPole-v1 for experimentation where each episode can reach a maximum of reward of 500, equivalent to balancing a pole on a moving cart for 500 time steps. The MDP has discrete actions, namely '`left`' and '`right`', and observations/states comprised of four features shown below:

Num	Observation	Min	Max	Clipping Range
0	Cart Position	-4.8	4.8	$(-2.5, 2.5)$
1	Cart Velocity	-Inf	Inf	$(-3.6, 3.6)$
2	Pole Angle	-0.418 rad (-24 deg)	0.418 rad (24deg)	$(-0.27, 0.27)$
3	Pole Angular Velocity	$-$ Inf	Inf	$(-3.7, 3.7)$

As depicted above, features can assume a wide range of values. To make the problem tractable, we clip the observed values based on the range shown in the table and then normalize the values. The plot below depicts the total rewards during the training over 1000 episodes, with a moving average of size 10.



The results suggest that Fourier basis of order 3 is adequate for this problem and has been able to reach a policy that has been able to achieve the perfect total reward. However this does not seem to be the case for higher orders of approximation. We know that the number of terms grow exponentially with higher orders, implying higher dimension of θ to be estimated. The curse of dimensionality here is evident from the performance of model with orders of 5 and 7, as depicted above.

```
import gym
import numpy as np
import matplotlib.pyplot as plt
from itertools import combinations, product
import seaborn as sns
sns.set_style("white")

class QLearningFourier():
    def __init__(self, state_space, action_space, state_ranges=None, alpha=.01, gamma=.9,
     ↪ epsilon=.01, fourier_order=3, max_non_zero_fourier=2):
        self.alpha = alpha
```

```
        self.gamma = gamma
        self.epsilon = epsilon

        self.state_space = state_space
        self.state_dim = self.state_space.shape[0]
        self.action_space = action_space
        self.action_dim = self.action_space.n

        self.order = fourier_order
        self.state_ranges = state_ranges
        self.max_non_zero = min(max_non_zero_fourier, state_space.shape[0])
        self.coeff = self._build_coeffs()

        if state_ranges is not None:
            self.intervals = np.squeeze(np.diff(state_ranges, axis=0))

        self.lr = self.get_learning_rates(self.alpha)

        self.num_basis = len(self.coeff)

        self.theta = {a: np.zeros(self.num_basis) for a in range(self.action_dim)}

        self.q_old = None
        self.action = None

    def learn(self, state, action, reward, next_state, done):
        phi = self.get_features(state)
        next_phi = self.get_features(next_state)
        q = self.get_q_value(phi, action)
        if not done:
            next_q = max([self.get_q_value(next_phi, a) for a in range(self.action_dim)])
        else:
            next_q = 0.0

        error = reward + self.gamma * next_q - q
        if self.q_old is None:
            self.q_old = q

        self.theta[action] += self.lr*(error) * phi

        self.q_old = next_q

    def get_q_value(self, features, action):
        return np.dot(self.theta[action], features)

    def get_features(self, state):
        if self.state_ranges is not None:
            state = (state - self.state_ranges[0]) / self.intervals
            #state = (state - self.state_space.low) / (self.state_space.high -
             ↪  self.state_space.low)
        return np.cos(np.dot(np.pi*self.coeff, state))

    def act(self, obs):
        features = self.get_features(obs)
        return self.get_action(features)

    def get_action(self, features):
        if np.random.rand() < self.epsilon:
            return np.random.randint(self.action_dim)
        else:
```

```
            q_values = [self.get_q_value(features, a) for a in range(self.action_dim)]
            return q_values.index(max(q_values))

    def get_learning_rates(self, alpha):
        lrs = np.linalg.norm(self.coeff, axis=1)
        lrs[lrs==0.] = 1.
        lrs = alpha/lrs
        return lrs

    def get_num_basis(self) -> int:
        return len(self.coeff)

    def _build_coeffs(self):
        coeff = np.array(np.zeros(self.state_dim))  # Bias

        for i in range(1, self.max_non_zero + 1):
            for indices in combinations(range(self.state_dim), i):
                for c in product(range(1, self.order + 1), repeat=i):
                    coef = np.zeros(self.state_dim)
                    coef[list(indices)] = list(c)
                    coeff = np.vstack((coeff, coef))
        return coeff

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

clipped_high        = env.observation_space.high
clipped_high = np.array([2.5, 3.6, 0.27, 3.7])
clipped_low         = -clipped_high
state_ranges = np.array([clipped_low, clipped_high])

f_order = 3

gamma = .9
alpha = .005
epsilon = .08

agent = QLearningFourier(env.observation_space, env.action_space, alpha=alpha, fourier_order=f_order,
                         gamma=gamma, epsilon=epsilon, state_ranges=state_ranges)

N_episodes = 1000

seed = 47
env.seed(seed)
np.random.seed(seed)

ep = 0
S = env.reset()
ret = 0
rets = []
while ep < N_episodes:
    A = agent.act(S)
    S_, R, done, _ = env.step(A)
    #if ep > 1000 and (((ep+1) % 10) == 0):
        #env.render()
    ret += R
    agent.learn(S, A, R, S_, done)
    S = S_
    if done:
        # if ret > 400:#((ep+1) % 10) == 0:
```

```
        #     print(ep, "Return:", ret)
        rets.append(ret)
        ret = 0
        ep += 1
        S = env.reset()

rets_orders[f_order] = rets
```