ECE5984: Reinforcement Learning

Assignment #5

Nima Mohammadi

nimamo@vt.edu

Problem 1.
Let $f:\mathbb{R}^d \to \mathbb{R}$ be a smooth function, i.e., f is continuously differentiable function and has Lipschitz continuous gradient with constant $L > 0$

$$\|\nabla f(x) - \nabla f(y)\| \leq L\|x-y\| \quad \forall x, y \in \mathbb{R}^d$$

Show the following
1. For all $x, y \in \mathbb{R}^d$

$$\left|f(x) - f(y) - \nabla f(y)^T(x-y)\right| \leq \frac{L}{2}|x-y|^2$$

Let $g:\mathbb{R}\to\mathbb{R}$ defined as $g(\tau) \triangleq f(y+\tau(x-y))$. Then, we would have

$$f(x) = g(1) \text{ and } f(y) = g(0)$$

Which implies

$$f(x) - f(y) = g(1) - g(0) = \int_0^1 g'(\tau)d\tau$$

For the derivative of $g(t)$ we have

$$g'(\tau) = \nabla f(y+\tau(x-y))^T(x-y)$$

And plugging back to left-hand-side of the inequality we will have

$$\begin{aligned}
\left|f(x) - f(y) - \nabla f(y)^\top(x-y)\right| &= \left|\int_0^1 \nabla f(y+\tau(x-y))^\top(x-y)d\tau - \nabla f(y)^\top(x-y)\right| \\
&\leq \int_0^1 \left|\nabla f(y+\tau(x-y) - \nabla f(y))^\top(x-y)\right| d\tau \\
&\leq \int_0^1 \|\nabla f(y+\tau(x-y)) - \nabla f(y)\| \cdot \|x-y\| d\tau \\
&\leq \int_0^1 L\tau\|x-y\|^2 d\tau \\
&= \frac{L}{2}\|x-y\|^2
\end{aligned}$$

✓

where we used the Cauchy-Schwarz inequality and L-Lipschitz-ness of ∇f in last two inequalities.

2. In addition to the Lipschitz continuity of ∇f, suppose f is twice continuously differentiable. Show that the Hessian of f is bounded, i.e.,

$$\left\|\nabla^2 f(x)\right\| \leq L \quad \forall x \in \mathbb{R}^d$$

Let $Z = x - y$ such that for $\alpha \in [0,1]$,

$$\nabla f(y+\alpha Z) - \nabla f(y) = \left(\int_0^\alpha \nabla^2 f(y+\tau Z) d\tau\right)^T Z$$

holds. Then

$$\left\|\left(\int_0^\alpha \nabla^2 f(y+\tau Z) d\tau\right)^T Z\right\| = \|\nabla f(y+\alpha Z) - \nabla f(y)\| \leq \alpha L \|Z\|$$

$$\Rightarrow \frac{\left\|\left(\int_0^\alpha \nabla^2 f(y+\tau Z) d\tau\right)^T Z\right\|}{\alpha\|Z\|} \leq L$$

Then, as α approaches zero, we get

$$\left\|\nabla^2 f(x)\right\| \leq L \quad \forall x \in \mathbb{R}^d$$

✓

Problem 2.
In this problem we will derive the policy gradient formulas for different policies. First, we recall a few notation given in the class. Let V_π be the value function associated with the policy π

$$V_\pi(s) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r(s_k, a_k) \mid a_k \sim \pi(\cdot \mid s_k), s_0 = s\right]$$

where $\gamma \in (0,1)$ is the discount factor. Let ρ be the initial distribution. With some abuse of notation we denote by

$$V_\pi(\rho) = \mathbb{E}_{s\sim\rho}[V_\pi(s)]$$

Next, we parameterize the policy π by θ, i.e., $\{\pi_\theta \mid \theta \in \Theta\}$. We denote by the discounted state visitation distribution

$$d_{s_0}^\pi(s) := (1-\gamma)\sum_{k=0}^{\infty} \gamma^k \mathbb{P}^\pi(s_k = s \mid s_0)$$

and let $d_\rho^\pi(s) = \mathbb{E}_{s_0\sim\rho}\left[d_{s_0}^\pi(s)\right]$. By the policy gradient theorem we have

$$\nabla_\theta V_{\pi_\theta}(s_0) = \frac{1}{1-\gamma}\mathbb{E}_{s\sim d_{s_0}^{\pi_\theta}}\mathbb{E}_{a\sim\pi_\theta(\cdot|s)}\left[\nabla_\theta \log \pi_\theta(a \mid s) Q_{\pi_\theta}(s,a)\right]$$

where

$$Q_\pi(s,a) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^t c(s_k, a_k) \mid s_0 = s, a_0 = a\right], \quad a_k \sim \pi(\cdot \mid), \forall k \geq 1$$

$$A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s)$$

Denote the simplex over the set of actions as

$$\Delta(\mathcal{A})^{|\mathcal{S}|} \triangleq \left\{\theta \in \mathbb{R}^{|\mathcal{S}||\mathcal{A}|} \mid \theta_{s,a} \geq 0, \sum_{a\in\mathcal{A}} \theta_{s,a} = 1\right\}$$

Thus, if $\pi \in \Delta(\mathcal{A})^{|\mathcal{S}|}$ then we have

$$\nabla_\theta V_{\pi_\theta}(s_0) = \frac{1}{1-\gamma}\mathbb{E}_{a\sim d_{s_0}^{\pi_\theta}}\mathbb{E}_{a\sim\pi_\theta(\cdot|s)}\left[\nabla_\theta \log \pi_\theta(a \mid s) A_{\pi_\theta}(a,u)\right]$$

1. Consider direct parameterization, i.e., $\mathcal{S}$ and $\mathcal{A}$ are the set of states and actions, and

$$\pi_\theta(a \mid s) = \theta_{s,a}, \quad \text{where } \theta \in \Delta(\mathcal{A})^{|\mathcal{S}|}$$

Show that

$$\frac{\partial V^{\pi_\theta}(\rho)}{\theta_{s,a}} = \frac{1}{1-\gamma} d_\rho^{\pi_\theta} Q^{\pi_\theta}(s,a)$$

✓

We had

$$\nabla_\theta V_{\pi_\theta}(s_0) = \frac{1}{1-\gamma}\mathbb{E}_{s\sim d_{s_0}^{\pi_\theta}}\mathbb{E}_{a\sim\pi_\theta(\cdot\mid s)}\left[\nabla_\theta \log\pi_\theta(a\mid s)Q^{\pi_\theta}(s,u)\right]$$

And we had $\pi_\theta(a\mid s) = \theta_{s,a}$, so we can write

$$\frac{\partial \log\pi_\theta(a\mid s)}{\partial\theta_{a,s}} = \frac{\frac{\partial}{\partial\theta_{a,s}}\pi_\theta(a\mid s)}{\pi_\theta(a\mid s)} = \frac{1}{\pi_\theta(a,s)}$$

And from two results above,

$$\frac{\partial V_{\pi_\theta}(\rho)}{\partial\theta_{s,a}} = \frac{1}{1-\gamma}d_\rho^{\pi_\theta}Q^{\pi_\theta}(s,a)$$

✓

2. Consider softmax parameterization

$$\pi_\theta(a\mid s) = \frac{\exp(\theta_{s,a})}{\sum_{a'\in\mathcal{A}}\exp(\theta_{s,a'})}$$

Show that

$$\frac{\partial V_{\pi_\theta}(\rho)}{\partial\theta_{s,a}} = \frac{1}{1-\gamma}d_\rho^{\pi_\theta}\pi(a\mid s)A^{\pi_\theta}(s,a)$$

For the softmax parameterization, from the definition of $\pi_\theta(a\mid s)$,

$$\frac{\partial\pi_\theta(a\mid s)}{\partial\theta_{a,s}} = \pi_\theta(a,s) - (\pi_\theta(a\mid s))^2 \Rightarrow \frac{\partial\log\pi_\theta(a\mid s)}{\partial\theta_{a,s}} = 1-\pi_\theta(a\mid s)$$

$$\frac{\partial V^{\pi_\theta}(\rho)}{\partial\theta_{s,a}} = \mathbb{E}\left[\sum_{k=0}^{\infty}\gamma^k 1_{s_k=s}\left(1_{a_k=a}-\pi_\theta(a,s)\right)A^{\pi_\theta}(s_k,a_k)\right]$$

$$\Rightarrow \frac{\partial V_{\pi_\theta}(\rho)}{\partial\theta_{s,a}} = \mathbb{E}\left[\sum_{k=0}^{\infty}\gamma^k 1_{(s_k=s,a_k=a)}A^{\pi_\theta}(s_k,a_k)\right] - \pi_\theta(a\mid s)\mathbb{E}\left[\sum_{k=0}^{\infty}1_{s_k=s}A^{\pi_\theta}(s_k,a_k)\right]$$

We know $\mathbb{E}\left[\sum_{k=0}^{\infty}1_{s_k=s}A^{\pi_\theta}(s_k,a_k)\right]=0$, then

✓

$$\frac{\partial V_{\pi_\theta}(\rho)}{\partial\theta_{s,a}} = \frac{1}{1-\gamma}d_\rho^{\pi_\theta}\pi(a\mid s)A^{\pi_\theta}(s,a)$$

3. We denote by

$$Z(s) = \sum_{a\in\mathcal{A}}\pi_\theta(a\mid s)\exp\left(\alpha\frac{A^{\pi_\theta}(s,a)}{1-\gamma}\right)$$

where α is some positive constant. Show that

$$\log Z(s)\geq 0,\quad \forall\pi_\theta\in\Delta(\mathcal{A})^{|\mathcal{S}|}$$

After taking the logarithm of both sides of the equation above, we have

$$\log Z(s) = \log\sum_{a\in\mathcal{A}}\pi_\theta(a\mid s)\exp\left(\alpha\frac{A^{\pi_\theta}(s,u)}{1-\gamma}\right)$$

With log(.) being a concave function, we may apply the Jensen inequality, that is

$$\log Z(s)\geq\sum_{u\in\mathcal{U}}\pi_\theta(u\mid s)\underbrace{\log\left(\exp\left(\alpha\frac{A^{\pi_\theta}(s,u)}{1-\gamma}\right)\right)}$$

✓

$\hookrightarrow$ explain why this term is

which result in

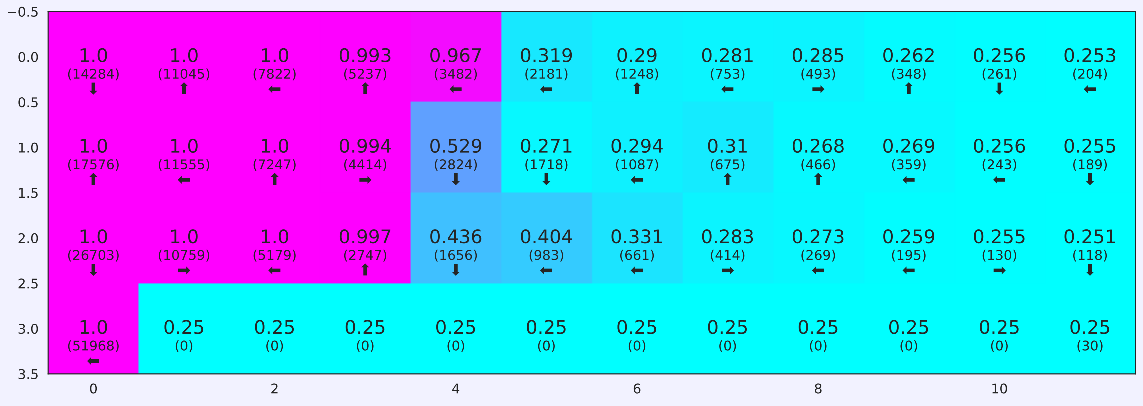
$$\log Z(s)\geq 0$$

Problem 4.
Write a simulation program to implement policy gradient in the tabular setting for the cliff-walking problem, similar to Hw3 and Hw4. For this task, the initial distribution always assigns a probability 1 to position S (i.e., you always start from S). Moreover, $f(\pi) = \mathbb{E}_{s\sim\rho}[V_\pi(s)]$ and we use the softmax parameterization

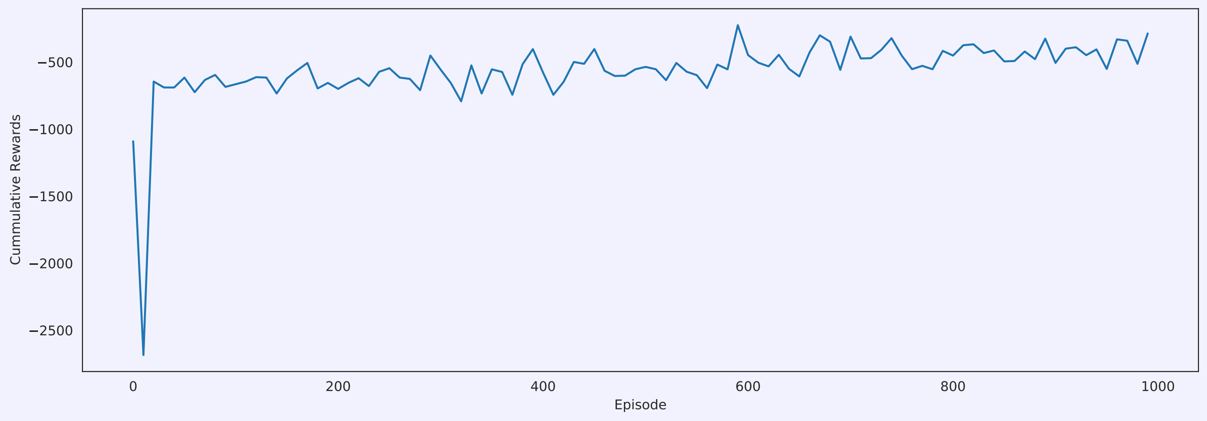
$$\pi_\theta(a \mid s) = \frac{\exp(\theta_{s,a})}{\sum_{a'\in\mathcal{A}} \exp(\theta_{s,a'})}, \quad \text{where } \theta \in \mathbb{R}^{|\mathcal{S}|\mathcal{A}|}$$

where the gradient is given in problem 2 . In your simulation, consider a number of episodes, where each episode runs until the program terminates or at most 200 steps. Submit your code and explain what's the optimal policy your algorithm can return, e.g., is it a deterministic policy? Is it the same with the policy returned by Q-learning in Hw4?

In this problem, we implement tabular Policy Gradient for the cliff-walking problem. In this approach, we forgo the value functions and directly optimize the parameterized policy model. I present several runs, and show the problem of the policy gradient approach with premature convergence.
I start with a randomized policy as the behavior policy. The plot below attempts to depict what the policy would look like. The plotted arrows are actions with highest probability at each state. The associated probability with the 'winning' action is written at each cell. Below that, there is a number (within parentheses) that indicate the number of times the state has been observed as s'. Notice that the environment implicitly move from the bottom-row states with $R = -100$ to the initial state, hence being reported as visited zero times.

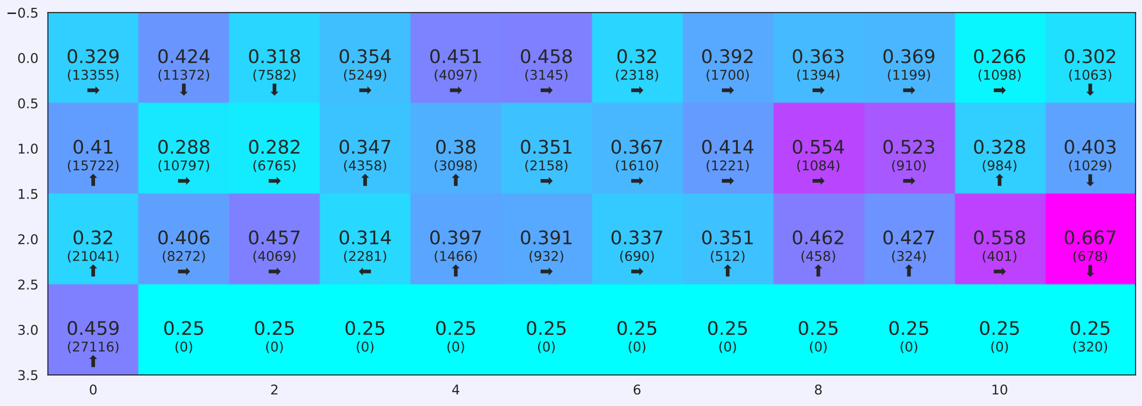


The result indicates that only 30 times the final terminating state has been observed, which can be expected given the uniformly random behavior policy used. It seems that policy gradient was able to figure out how to stay away from cliff states.
Next, we use the same policy that we are trying to optimize with policy gradient as the behavior policy. The returns of the 1000 simulated episodes, averaged every 10 episode, are shown below:

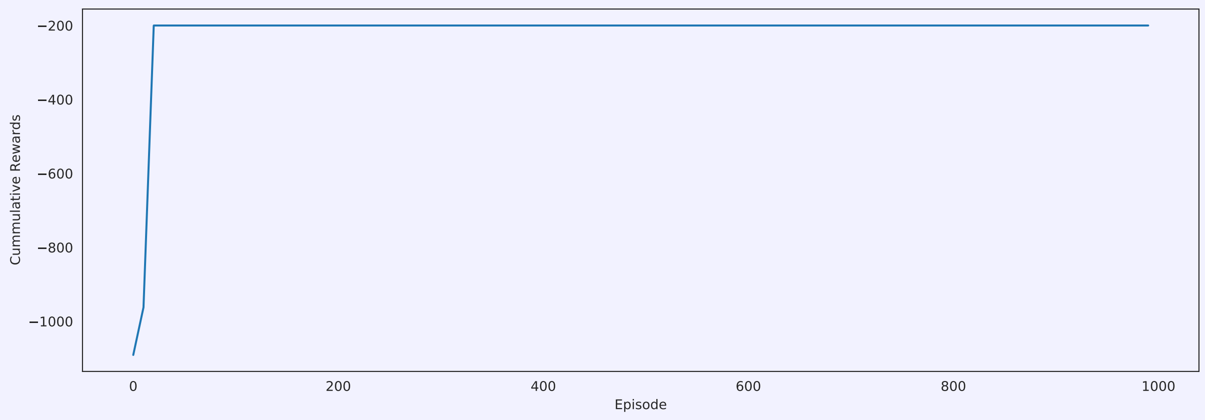


✓

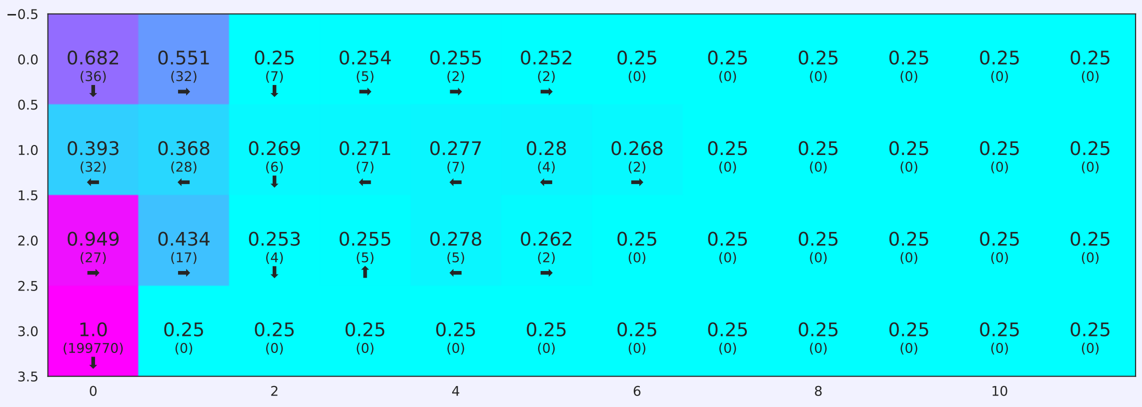
We see that the PG policy is improving with being trained. The policy is updated after episode. To shed light on the policy, we can look at the figure below:



Once more, we are far from the optimal policy, and in that regard, performing much worse than the Q-Learning algorithm. While, using the same PG behavior policy has caused the ending states of the optimal trajectory to be visited more, the certainty/probability of those beginning states' actions are lower.
Again, a main problem with this problem, is the severe constrain on limiting the episode lengths to 200 steps, which refrain it from
If we increase the learning rate to $\alpha = 0.003$, we observe the extreme case where we quickly converge to a policy that only obtains a return of -200.



That is the policy decides to dabble around the same initial state, refraining movement to cliff nodes, or exploring other states, as shown below: ✓



One possible solution to this premature convergence is to use Entropy Regularization, which pushes the agent to explore less-explored actions by penalizing over-confidence in actions during the training.

```python
class PolicyGradient:
  def __init__(self, alpha, gamma):
    self.alpha = alpha
    self.gamma = gamma
    self.env = env
    self.theta = np.zeros(shape=(self.env.observation_space.n, self.env.action_space.n),
   ↪  dtype=float)
```

```
    # self.theta = np.random.rand(self.env.observation_space.n, self.env.action_space.n)
    self.act_probs = np.ones([env.observation_space.n, env.action_space.n]) / env.action_space.n

  def softmax(self, x):
    # max_x = np.max(x)
    # return np.exp(x - max_x) / np.sum(np.exp(x - max_x), axis=0)
    return np.exp(x) / np.sum(np.exp(x), axis=0)

  def set_lr(self, new_alpha):
    self.alpha = new_alpha

  def featurize(self,state):
    s = np.zeros(shape=(self.env.observation_space.n,), dtype=float)
    s[state] = 1
    return s

  def get_action_probabilities(self, state):
    h = self.featurize(state).dot(self.theta).flatten()
    soft = self.softmax(h)
    return soft

  def gradient(self, state, action):
    x_s = self.featurize(state)
    action_probability = self.get_action_probabilities(state)[action]

    return x_s - action_probability * x_s

  def update_policy(self, buffer):
    buffer = np.array(buffer, dtype=np.int64)

    G = []
    for [state, action, reward] in buffer[::-1]:
      if len(G) == 0:
        G.append(reward)
      else:
        G.append(reward + G[-1])
    G = G[::-1]

    for t,[state, action, reward] in enumerate(buffer):
      gradient = self.gradient(state, action)
      self.theta[:,action] += self.alpha * np.power(self.gamma,t) * G[t] * gradient
    for state in range(self.env.observation_space.n):
      self.act_probs[state] = self.get_action_probabilities(state)

  def select_action(self,state):
    action_probabilities = self.act_probs[state]
    return np.random.choice(range(self.env.action_space.n), 1, p=action_probabilities)[0]
```

```
N_episodes = 1000
max_episode_length = 200
gamma = 0.9
alpha = .0003

state_count = np.zeros(env.observation_space.n)

pg = PolicyGradient(alpha, gamma)
ep_rews = []
```

```python
mean_rews = []

env.seed(45)
np.random.seed(27)

for ep in range(N_episodes):
  S = env.reset()
  i = 0
  done = False
  episode_reward = 0
  replay = []
  while ((not done) and (i < max_episode_length)):
    i += 1
    A = pg.select_action(S)
    # A = np.random.randint(4, size=1)[0]
    S_, R, done, _ = env.step(A)
    state_count[S_] += 1
    # print('R', R)

    replay.append([S, A, R])

    S = S_
    episode_reward += R
  ep_rews.append(episode_reward)
  pg.update_policy(replay)
  if (ep) % 10 == 0:
    mean_rews.append(np.mean(ep_rews[-10:]))
    if (ep) % 100 == 0:
      print("Ep: {}, Reward: {}, Avg last 10 rews: {}".format(ep, episode_reward,
      ↪  mean_rews[-1]))
```

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 policy gradient with Gaussian policy for the cart-pole problem. That is, the action is generated by

$$a \sim \mathcal{N}\left(\phi(s)^T\theta, \sigma^2\right)$$

where σ is your choice (e.g., 0.5) and $\phi(s)$ is your feature vector. The parameter $\theta \in \mathbb{R}^d$, where you can select the dimension d. The gradient of $f(\pi_\theta) = \mathbb{E}_{s\sim\rho}\left[V_{\pi_\theta}(s)\right]$ is then given as

$$\nabla_\theta f(\pi_\theta) = \frac{1}{1-\gamma}\mathbb{E}_{s\sim d_\rho^\pi}\mathbb{E}_{a\sim\pi_\theta(\cdot|s)}\left[\left(\nabla_\theta \log \pi_\theta(a \mid s)\right) Q_{\pi_\theta}(s,a)\right]$$

and

$$\nabla_\theta \log \pi_\theta(a \mid s) = \frac{\left(a - \phi(s)^T\theta\right)\phi(s)}{\sigma^2}$$

In your simulation, consider a number of episodes, where each episode runs until the program terminates. Submit your code and show whether the policy returned by your algorithm work? Try your code with different dimension d and σ, and explain the impacts of these choices on the performance of your algorithms.

In this part, we would like to implement the PG alrogithm for the Cartpole problem. The policy build on Fourier bases for approximation.
The zth-order Fourier basis for d state variables is the set of basis functions defined as

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

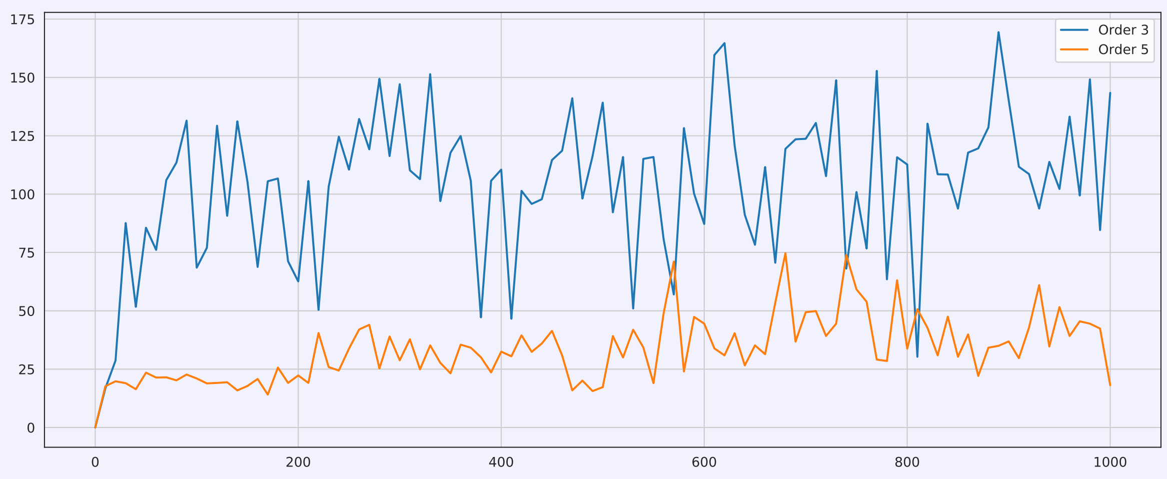
where $\mathbf{c}^i = [c_1, \ldots, c_d], c_j \in \{0, \ldots, z\}, 1 \leq j \leq d$. Each basis function is obtained using a vector $\mathbf{c}^i$ that attaches an integer coefficient (between 0 and z, inclusive) to each variable in s (after that variable has been scaled to $[0, 1]$).
The MDP has discrete 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. To update the policy parameters we make use of the follow update rule:

$$\nabla \ln \pi_\theta(a \mid s, \theta) = \phi(s, a) - \sum_{a' \in \mathcal{A}} \pi_\theta\left(a' \mid s, \theta\right) \phi\left(s, a'\right)$$

The plot below depicts the total rewards during the training over 1000 episodes, with a moving average of size 10. The plots shows two curves corresponding to Fourier orders of 3 and 5.



Sadly, the results of this experiment were quite unstable. One may achieve better results after repeating the experiment multiple times, but I reported these experiments which tend to be better representatives of what the majority of experiments would look like. Here we see that although the PG algorithm with order 3 Fourier basis does not converge to the optimal policy, there are gradual improvements and that it outperforms the experiment with Fourier basis of higher order. This could be attributed, to some extent, to dimensionality issues. The agent was found to be quite sensitive to hyperparameters such as α and γ, and one may find more stable models with more trial and error, and possibly an adaptive learning rate.

```python
import numpy as np
import gym
import time
import math
import matplotlib.pyplot as plt


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

✓

```python
env.reset()

lr = 0.0035

gamma = 0.9
n_episodes = 1000
total = 0
total_reward = 0
prior_reward = 0

Averaging_Step = 10
dim = 5
fourier_order = 3
coef_len = np.power(fourier_order+1, dim);

output = np.zeros([coef_len, dim])
for i in range(coef_len):
    index = i
    for j in range(dim-1):
        if index >= np.power(fourier_order+1, dim - (j+1)):
            output[i][j] = np.floor(index/np.power(fourier_order+1, dim - (j+1)))
            aa = np.int(output[i][j])
            index = index - np.power(fourier_order+1, dim - (j+1)) * aa
        else:
            output[i][j] = 0
    output[i][dim-1] = index

theta = np.zeros(coef_len)
phi1 = np.zeros(coef_len)
phi2 = np.zeros(coef_len)
cos_value1 = np.zeros(coef_len)
cos_value3 = np.zeros(coef_len)
episode_rewards = []
rews = np.zeros(n_episodes)
act_val_all = [0.1, 0.9]
Coef_Find = 0

def get_state(state):
    discrete_state = state/np.array([0.25, 0.25, 0.01, 0.1]) + np.array([15,10,1,10])
    return tuple(discrete_state.astype(np.int))

for episode in range(n_episodes + 1):
    S  = env.reset();
    discrete_state = get_state(S)
    done = False
    episode_reward = 0
    curr_state_norm = np.zeros(dim)

    A = 0
    Q_valu_next_state = np.zeros(env.action_space.n)
    traj = {}
    traj['curr_state_action'] = np.zeros([500, 5])
    traj['next_state'] = np.zeros([500, 5])
    traj['next_rew'] = np.zeros(500)
    traj['p'] = np.zeros([500, 2])
    step = 0;
    prob_norm = np.zeros(2)

    while not done:
        step = step + 1;
        curr_state_norm[0] = (S[0] + 2.5)/5.01
        curr_state_norm[1] = (S[1] + 3.6)/7.3
        curr_state_norm[2] = (S[2] + 0.27)/0.56
        curr_state_norm[3] = (S[3] + 3.7)/7.65
        prob_t                   = np.zeros(2);
        Phi_Value                  = np.zeros(2);
        for i in range(2):
            curr_state_norm[4] = act_val_all[i];
            cos_inp = curr_state_norm * output
            cos_inp = cos_inp.sum(1)
            cos_value1  = np.cos((np.pi) * cos_inp)
            cos_value2  = cos_value1 * theta
            cos_value2  = cos_value2.sum()
            prob_t[i] = pow(math.pi, cos_value2)

        prob_norm[0] = prob_t[0]/prob_t.sum()
        prob_norm[1] = prob_t[1]/prob_t.sum()
        if np.random.random() < prob_norm[0]:
            A = 0
            curr_state_norm[4] = act_val_all[0]
        else:
            A = 1
            curr_state_norm[4] = act_val_all[1]

        S_, R, done, _ = env.step(A)

        norm_S_ = normalize_state(S_)

        traj['curr_state_action'][step-1] = curr_state_norm
        traj['next_state'][step-1] = norm_S_
        traj['next_rew'][step-1] = R
        traj['p'][step-1] = prob_norm

        episode_reward += R ##add the reward
        new_discrete_state = get_state(S_)
        discrete_state = new_discrete_state
        S  = S_

    #print(episode_reward)
    G_t = np.zeros(step)
    for i in range(step):
        curr_state_norm = traj['curr_state_action'][i]
        cos_inp = curr_state_norm * output
        cos_inp = cos_inp.sum(1)
        cos_value1  = np.cos((np.pi) * cos_inp)
        if curr_state_norm[-1] == act_val_all[0]:
            curr_state_norm[-1] = act_val_all[1]
            ind = [0, 1]

        else:
            curr_state_norm[-1] = act_val_all[0]
            ind = [1, 0]
        cos_inp = curr_state_norm * output
        cos_inp = cos_inp.sum(1)
        cos_value2 = np.cos((np.pi) * cos_inp)
        prob_norm = traj['p'][i]
        grad = cos_value1 - prob_norm[ind[0]] * cos_value1 - prob_norm[ind[1]] * cos_value2

        for j in range(i,step):
            G_t[i] = G_t[i] + math.pow(gamma,j-i) * traj['next_rew'][j]
    theta = theta + lr * G_t[i] * grad
    if episode % Averaging_Step == 0:
        mean_reward = total_reward / Averaging_Step
        total_reward = 0
        episode_rewards.append(mean_reward)
        if mean_reward > 100:
            lr = lr * .9

    rews[episode-1] = episode_reward
    total_reward += episode_reward
    prior_reward = episode_reward

plt.plot(np.linspace(0, n_episodes, len(episode_rewards)), episode_rewards)
plt.show()
```