

# Optimal and Learning Control for Robotics

## Exercise Series 4

Bo Peng, New York University

Spring 2020

### Question 1

**Solution** a) The value function  $V$  is a linear combination of parameter  $\theta_V$  and basis functions  $B(x)$ :

$$\begin{aligned} V &= \theta_V^T B(x) \\ &= \sum_{i=1}^N \theta_{V_i} B_i(x) \end{aligned} \tag{1}$$

The derivative of  $\theta_V$  is:

$$\nabla_{\theta_V} V = \begin{bmatrix} \frac{\partial V}{\partial \theta_{V1}} \\ \frac{\partial V}{\partial \theta_{V2}} \\ \vdots \\ \frac{\partial V}{\partial \theta_{VN}} \end{bmatrix} = \begin{bmatrix} B_1(x) \\ B_2(x) \\ \vdots \\ B_N(x) \end{bmatrix} = B(x) \tag{2}$$

b) The derivative of  $\ln \pi(u|x, \theta_\pi)$  is:

$$\begin{aligned} \nabla_{\theta_\pi} \ln \pi(u|x, \theta_\pi) &= \nabla_{\theta_\pi} \ln \frac{\exp\{h(x, u, \theta_\pi)\}}{\sum_a \exp\{h(x, a, \theta_\pi)\}} \\ &= \nabla_{\theta_\pi} \ln \exp\{h(x, u, \theta_\pi)\} - \ln \sum_a \exp\{h(x, a, \theta_\pi)\} \\ &= \nabla_{\theta_\pi} h(x, u, \theta_\pi) - \ln \sum_a \exp\{h(x, a, \theta_\pi)\} \\ &= \nabla_{\theta_\pi} h(x, u, \theta_\pi) - \nabla_{\theta_\pi} \ln \sum_a \exp\{h(x, a, \theta_\pi)\} \end{aligned} \tag{3}$$

In this case, the preference function  $h(x, u, \theta_\pi)$  is a linear combination of parameter  $\theta_\pi$  and basis functions  $\Psi(x, u)$ :

$$\begin{aligned} h(x, u, \theta_\pi) &= \theta_\pi^T \Psi(x, u) \\ &= \sum_{i=1}^M \theta_{\pi i} \Psi_i(x, u) \end{aligned} \tag{4}$$

So the first part of derivative is:

$$\nabla_{\theta_\pi} h(x, u, \theta_\pi) = \nabla_{\theta_\pi} \theta_\pi^T \Psi(x, u) = \Psi(x, u) \quad (5)$$

Let  $u = \sum_a \exp\{h(x, a, \theta_\pi)\}$ , so the second part of derivative is:

$$\begin{aligned} \nabla_{\theta_\pi} \ln \sum_a \exp\{h(x, a, \theta_\pi)\} &= \nabla_{\theta_\pi} \ln u \\ &= \frac{\partial \ln u}{\partial u} \cdot \frac{\partial u}{\partial \theta_\pi} \\ &= \frac{1}{u} \sum_a \frac{\partial \exp\{h(x, a, \theta_\pi)\}}{\partial \theta_\pi} \\ &= \frac{1}{u} \sum_a \frac{\partial \exp\{h(x, a, \theta_\pi)\}}{\partial h(x, a, \theta_\pi)} \cdot \frac{\partial h(x, a, \theta_\pi)}{\partial \theta_\pi} \\ &= \frac{1}{u} \sum_a \exp\{h(x, a, \theta_\pi)\} \cdot \Psi(x, a) \\ &= \sum_a \frac{\exp\{h(x, a, \theta_\pi)\}}{u} \cdot \Psi(x, a) \\ &= \sum_a \frac{\exp\{h(x, a, \theta_\pi)\}}{\sum_a \exp\{h(x, a, \theta_\pi)\}} \cdot \Psi(x, a) \\ &= \sum_a \pi(a|x, \theta_\pi) \cdot \Psi(x, a) \\ &= \Psi \pi \end{aligned} \quad (6)$$

where,

$$\pi = [\pi(u_1|x, \theta_\pi), \quad \pi(u_2|x, \theta_\pi), \quad \dots, \quad \pi(u_M|x, \theta_\pi)] \quad (7)$$

$$\Psi = [\Psi(x, u_1), \quad \Psi(x, u_2), \quad \dots, \quad \Psi(x, u_M)] \quad (8)$$

suppose we have  $M$  possible control inputs.

Finally, add the 2 parts together to get the final derivative:

$$\nabla_{\theta_\pi} \ln \pi(u|x, \theta_\pi) = \Psi(x, u) - \Psi \pi \quad (9)$$

c) The python code for REINFORCE algorithm is shown here:

```

1  class Reinforce:
2      """
3      An implementation of the reinforce algorithm (with or without baseline)
4      """
5
6      def __init__(self, model, cost, policy, discount_factor=0.99,
7                  episode_length=100, policy_learning_rate = 0.000001):
8          """
9          the class constructor
10         """
11         self.model = model
12         self.cost = cost

```

```

13         self.policy = policy
14
15         self.discount_factor = discount_factor
16         self.episode_length = episode_length
17
18         self.policy_learning_rate = policy_learning_rate
19
20     def iterate(self, num_iter=1):
21         """
22         the main loop
23         """
24         learning_progress = []
25
26         # here we allocate some useful vectors
27         x_traj = np.zeros([self.episode_length+1, self.model.num_states])
28         u_traj = np.zeros([self.episode_length, 1])
29         u_index = np.zeros([self.episode_length], dtype=np.int)
30         cost_traj = np.zeros([self.episode_length])
31
32         for i in range(num_iter):
33             # generate an episode - start from 0
34             x_traj[0,:] = np.zeros([2])
35             # TO COMPLETE #
36             # you can use the step function of self.model (i.e. the pendulum) to get the
37             next state
38             G = 0
39             ## generate an episode
40             for t in range(self.episode_length):
41                 x = x_traj[t,:]
42
43                 ## take one step and get cost
44                 idx, u = self.policy.sample(x)
45                 g = self.cost(x, u)
46
47                 ## update buffer
48                 x_traj[t+1,:] = self.model.step(x, u)
49                 u_traj[t] = u
50                 u_index[t] = idx
51                 cost_traj[t] = g
52
53                 G += g * self.discount_factor**t
54
55             # now compute the returns Gt and update the policy parameters self.policy.
56             theta through gradient descent
57             # TO COMPLETE #
58             for k in range(self.episode_length):
59                 alpha = np.array([self.discount_factor**i for i in range(self.
60 episode_length-k)])
61                 Gt = alpha @ cost_traj[k:]
62
63                 x, idx = x_traj[k], u_index[k]
64                 dist, basis_fun = self.policy.get_distribution(x)
65                 dtheta = basis_fun[:, idx] - basis_fun @ dist
66                 self.policy.theta -= self.policy_learning_rate * self.discount_factor**k
67                 * Gt * dtheta
68
69             # here we store the return at t=0 to get the learning progress
70             print(f"Iteration: {i}, Discounted Cost: {G}")
71             learning_progress.append(G)

```

```

69         return learning_progress
70
71     def get_Policy(self):
72         """
73         This helper function generate a 50x50 grid (theta x omega) with a policy (for
74         display)
75         the policy is computed as the expected control from pi
76         we also compute a value function (to be used for the baseline part) - for now it
77         is 0
78         """
79         n_discrete = 50
80         pol = np.zeros([n_discrete,n_discrete])
81         val = np.zeros([n_discrete,n_discrete])
82         x_range = np.linspace(self.model.state_range[0,0], self.model.state_range[0,1],
83                               n_discrete)
84         v_range = np.linspace(self.model.state_range[1,0], self.model.state_range[1,1],
85                               n_discrete)
86
87         for i, x in enumerate(x_range):
88             for j,v in enumerate(v_range):
89                 dist, basis = self.policy.get_distribution(np.array([x,v]))
90                 pol[i,j] = dist.dot(self.model.controls)
91                 # this can be used later to get a value estimate
92                 # val[i,j] = self.value.getValue(np.array([x,v]))[0]
93         return pol, val

```

Listing 1: REINFORCE Algorithm

The hyper-parameters for training the inverted pendulum are: discounted factor  $\alpha = 0.99$ , episode length 100, order  $p = 2$ , learning rate  $\gamma = 5 \times 10^{-8}$ , iteration  $10^5$ . The training results are shown below:

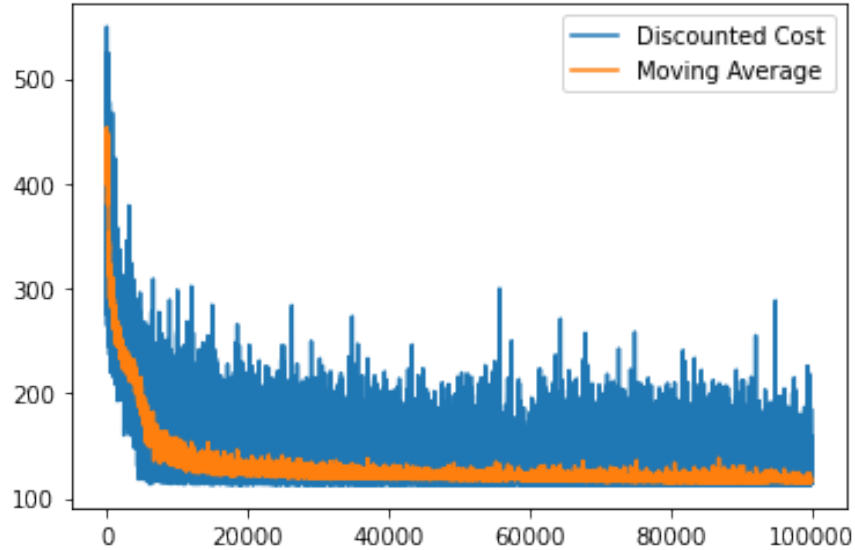


Figure 1: REINFORCE Training History

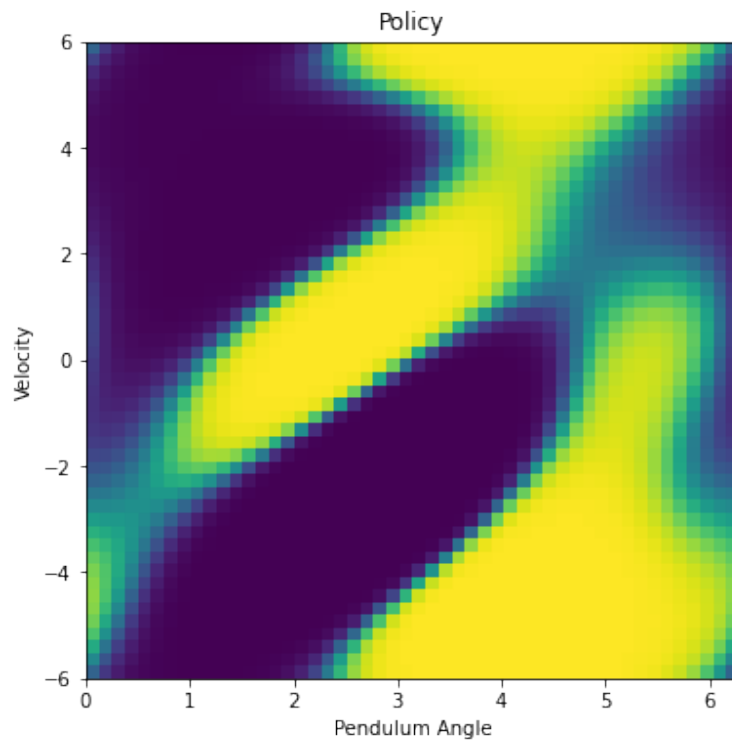


Figure 2: REINFORCE Policy

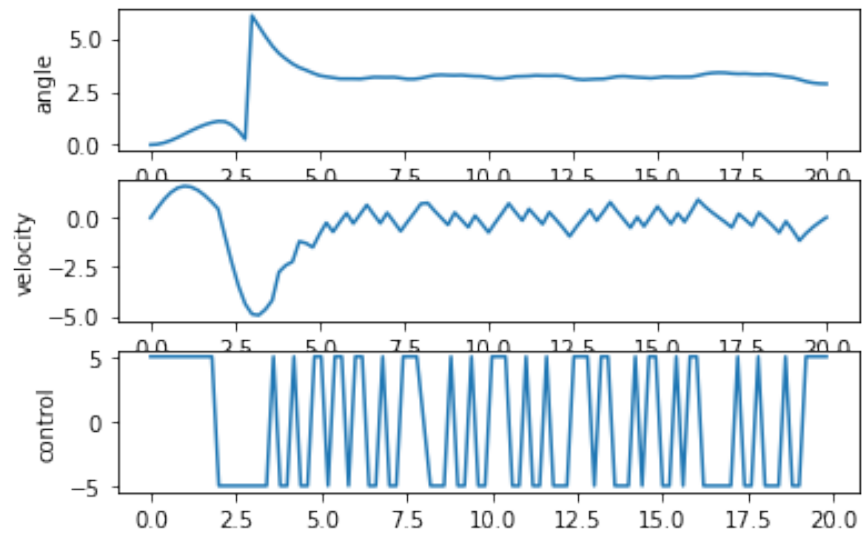


Figure 3: REINFORCE Trajectory

The animation of the trajectory can be found [here](#). As is shown in Fig. 1, the discounted cost converges after 40,000 iterations of training and the learned policy could balance the pendulum appropriately.

c) The python code for REINFORCE algorithm with baseline is shown here:

```

1  class ReinforceBaseline:
2      """
3      An implementation of the reinforce algorithm with baseline.
4      """
5
6      def __init__(self, model, cost, policy, value, discount_factor=0.99,
7          episode_length=100, value_learning_rate=1e-3, policy_learning_rate =
8          0.000001):
9          """
10             the class constructor
11             """
12             self.model = model
13             self.cost = cost
14
15             self.policy = policy
16             self.value = value
17
18             self.discount_factor = discount_factor
19             self.episode_length = episode_length
20
21             self.policy_learning_rate = policy_learning_rate
22             self.value_learning_rate = value_learning_rate
23
24     def iterate(self, num_iter=1):
25         """
26         the main loop
27         """
28         learning_progress = []
29
30         # here we allocate some useful vectors
31         x_traj = np.zeros([self.episode_length+1, self.model.num_states])
32         u_traj = np.zeros([self.episode_length, 1])
33         u_index = np.zeros([self.episode_length], dtype=np.int)
34         cost_traj = np.zeros([self.episode_length])
35
36         for i in range(num_iter):
37             # generate an episode - start from 0
38             x_traj[0,:] = np.zeros([2])
39             # TO COMPLETE #
40             # you can use the step function of self.model (i.e. the pendulum) to get the
41             next state
42             G = 0
43             ## generate an episode
44             for t in range(self.episode_length):
45                 x = x_traj[t,:]
46
47                 ## take one step and get cost
48                 idx, u = self.policy.sample(x)
49                 g = self.cost(x, u)
50
51                 ## update buffer
52                 x_traj[t+1] = self.model.step(x, u)
53                 u_traj[t] = u
54                 u_index[t] = idx
55                 cost_traj[t] = g

```

```

55         G += g * self.discount_factor**t
56
57         # now compute the returns Gt and update the policy parameters self.policy.
58         # theta through gradient descent
59         # TO COMPLETE #
60         for k in range(self.episode_length):
61             x, idx = x_traj[k], u_index[k]
62             dist, basis_fun = self.policy.get_distribution(x)
63
64             alpha = np.array([self.discount_factor**i for i in range(self.
65 episode_length-k)])
66             Gt = alpha @ cost_traj[k:]
67             V, v_basis = self.value.getValue(x)
68             delta = Gt - V
69
70             ## update value function
71             dv = v_basis * delta
72             self.value.theta += self.value_learning_rate * self.discount_factor**k *
73 dv
74
75             ## update policy
76             dtheta = basis_fun[:, idx] - basis_fun @ dist
77             self.policy.theta -= self.policy_learning_rate * self.discount_factor**k
78 * delta * dtheta
79
80             # here we store the return at t=0 to get the learning progress
81             print(f"Iteration: {i}, Discounted Cost: {G}")
82             learning_progress.append(G)
83
84         return learning_progress
85
86     def get_Policy(self):
87         """
88         This helper function generate a 50x50 grid (theta x omega) with a policy (for
89 display)
90         the policy is computed as the expected control from pi
91         we also compute a value function (to be used for the baseline part) - for now it
92 is 0
93         """
94         n_discrete = 50
95         pol = np.zeros([n_discrete, n_discrete])
96         val = np.zeros([n_discrete, n_discrete])
97         x_range = np.linspace(self.model.state_range[0,0], self.model.state_range[0,1],
98 n_discrete)
99         v_range = np.linspace(self.model.state_range[1,0], self.model.state_range[1,1],
100 n_discrete)
101
102         for i, x in enumerate(x_range):
103             for j, v in enumerate(v_range):
104                 dist, basis = self.policy.get_distribution(np.array([x,v]))
105                 pol[i,j] = dist.dot(self.model.controls)
106                 # this can be used later to get a value estimate
107                 val[i,j] = self.value.getValue(np.array([x,v]))[0]
108         return pol, val

```

Listing 2: REINFORCE Algorithm with Baseline

The hyper-parameters for training the inverted pendulum are: discounted factor  $\alpha = 0.99$ , episode length 100, order  $p = 2$ , value learning rate  $\gamma_V = 0.01$ , policy learning rate  $\gamma_\pi = 1 \times 10^{-6}$ , iteration  $10^5$ . The training

results are shown below:

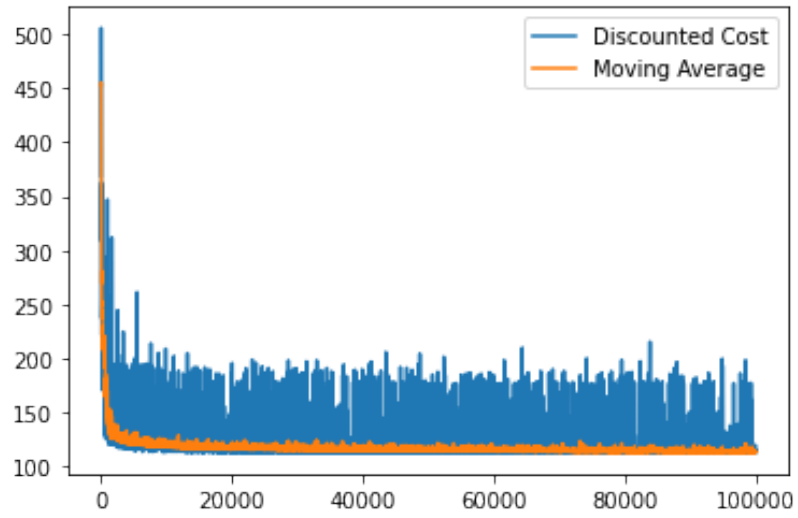


Figure 4: REINFORCE with Baseline Training History

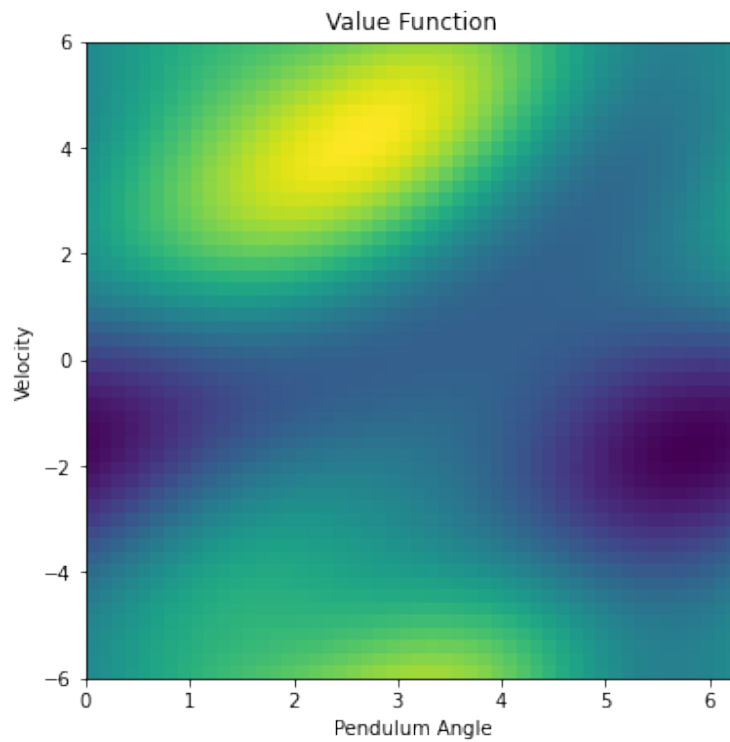


Figure 5: REINFORCE with Baseline Value Function



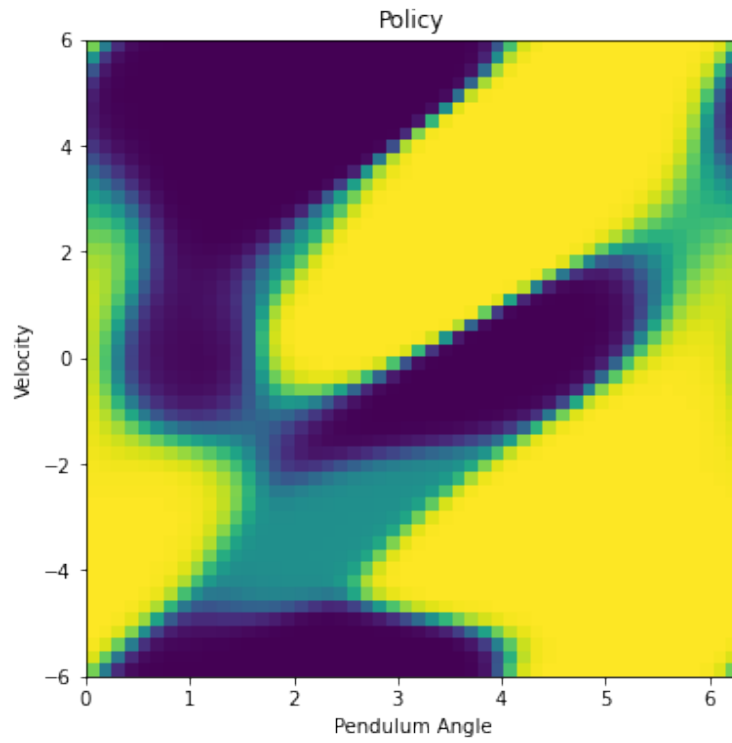


Figure 6: REINFORCE with Baseline Policy

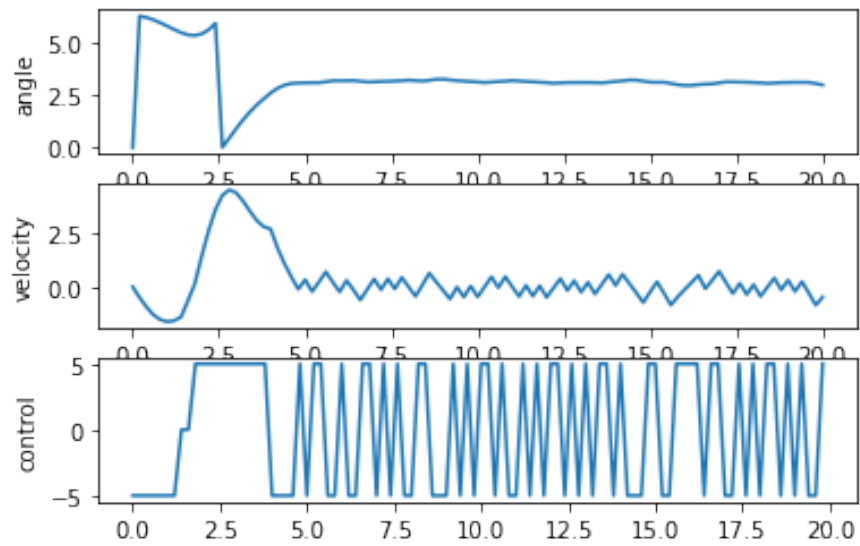


Figure 7: REINFORCE with Baseline Trajectory

The animation of the trajectory can be found [here](#). As is shown in Fig.4, the training converges in about 40,000 steps, which is much faster than vanilla REINFORCE algorithm. And the system is stabilized with the learned policy.

e) Fig.1 and Fig.4 show that REINFORCE with baseline has a faster converge speed and it allows a larger learning rate for policy, which is useful for hyper-parameter tuning. In general, REINFORCE with baseline is easier to use compared to the vanilla algorithm.