CSC411 - Project #4

Yui Chit (Michael) Wong - 999806232
Yijin (Catherine) Wang - 998350476

March 29, 2017

Part 1

Question Explain precisely why the code corresponds to the pseudocode below. Specifically, in your report, explain how all the terms (G_t , π , and the update to θ) are computed, quoting the relevant lines of Python.

REINFORCE, A Monte-Carlo Policy-Gradient Method (episodic)

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta}), \forall a \in \mathcal{A}, s \in \mathcal{S}, \boldsymbol{\theta} \in \mathbb{R}^n$
Initialize policy weights $\boldsymbol{\theta}$
Repeat forever:
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  For each step of the episode $t = 0, \ldots, T-1$:
    $G_t \leftarrow$ return from step t
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\gamma^t G_t \nabla_{\boldsymbol{\theta}} \log \pi(A_t|S_t,\boldsymbol{\theta})$

Pseudocode

Answer As mentioned on the assignment page, the policy function π_θ is implemented with a single-hidden-layer of neural network. Since the actions for the bipedal walker is continuous, we have to use a Gaussian distribution on π. Thus we pass the hidden layer into two separately fully connected output, which represents the μ and σ to the normal distribution. The activation function are *tanh* and *softplus* (variation on *ReLU*) respectively. The *sigma* value are also clipped if it is too small or big.

```
# 1 layer of hidden unit. Activation is ReLU
hidden = fully_connected(
    inputs=x,
    num_outputs=hidden_size,
    activation_fn=tf.nn.relu,
    weights_initializer=hw_init,
    weights_regularizer=None,
    biases_initializer=hb_init,
    scope='hidden')

# use last layer of neural network as phi(a, s) (the feature)
# mu = phi(s, a)^T dot theta
mus = fully_connected(
    inputs=hidden,
    num_outputs=output_units,
    activation_fn=tf.tanh,
    weights_initializer=mw_init,
    weights_regularizer=None,
    biases_initializer=mb_init,
    scope='mus')
```

```
21
22 # softplus is similar to ReLU. Activation function is g(x) = ln(1+e^x)
23 sigmas = tf.clip_by_value(fully_connected(
24     inputs=hidden,
25     num_outputs=output_units,
26     activation_fn=tf.nn.softplus,
27     weights_initializer=sw_init,
28     weights_regularizer=None,
29     biases_initializer=sb_init,
30     scope='sigmas'),
31     TINY, 5)
```

As for the weight intialization, if there is no weight saved from the previous run, we will initialize the weight θ. There are one w and b for each of the layers (hidden, μ, σ). When initializing the weight to each layer, the program uses *xavierinitialization*, another variation of random weight initialization that keep the scale of the gradients in roughly the same scale.

```
1 # if we have the w's and b's saved, load it. Otherwise initialize it
2 if args.load_model:
3     model = np.load(args.load_model)
4     hw_init = tf.constant_initializer(model['hidden/weights'])
5     hb_init = tf.constant_initializer(model['hidden/biases'])
6     mw_init = tf.constant_initializer(model['mus/weights'])
7     mb_init = tf.constant_initializer(model['mus/biases'])
8     sw_init = tf.constant_initializer(model['sigmas/weights'])
9     sb_init = tf.constant_initializer(model['sigmas/biases'])
10 else:
11     hw_init = weights_init
12     hb_init = relu_init
13     mw_init = weights_init
14     mb_init = relu_init
15     sw_init = weights_init
16     sb_init = relu_init
```

Once everything is initialized, we will start training. For each iteration, we will reset the environment (line 2), then generate the states, actions, and rewards from time 0 to time T. When generating the actions, we will randomly sample from the π normal distribution (line 16). Then based on the $pi.sample()$, we will generate the corresponding action pi_sample, and using the action, the new state, reward would be generated. We will keep track of all the states, actions, and rewards in 3 lists ($ep\_states, ep\_actions$, and $ep\_rewards$). We will also keep track of the total discounted rewards using the variable G (line 20). Then to obtain G_t, the discounted reward starting from time t, the program calls a culmulation sum function on the $ep_rewards$ then subtract it from G (line 30). Thus *returns* would be storing the total discouned rewards for each time from time 0 to $T-1$.

```
1 for ep in range(16384):
2     obs = env.reset()    # reset the environment
3
4     G = 0            # generating all the states and actions and rewards
5     ep_states = []
```

```
6       ep_actions = []
7       ep_rewards = [0]
8       done = False
9       t = 0
10      I = 1
11      while not done:
12          ep_states.append(obs)
13          env.render()
14          # pi.sample() is the list of randomly generated probablity
15          # pi_sample becomes the action
16          action = sess.run([pi_sample], feed_dict={x:[obs]})[0][0]
17          ep_actions.append(action)
18          obs, reward, done, info = env.step(action)
19          ep_rewards.append(reward * I)
20          G += reward * I # G is the total discounted reward
21          I *= gamma
22
23          t += 1
24          if t >= MAX_STEPS:
25              break
26      # done generating
27
28      if not args.load_model:
29          # G_t = total - culmulative up to time t.
30          returns = np.array([G - np.cumsum(ep_rewards[:-1])]).T
31          index = ep % MEMORY
32
33          # ep_states contains all the state S_0 to S_T-1
34          # ep_actions contains all the actions from A_0 to A_T-1
35          # returns (ie reward) contains all the G_t's form t=0 to t=T
36          _ = sess.run([train_op],
37                       feed_dict={x:np.array(ep_states),
38                                  y:np.array(ep_actions),
39                                  Returns:returns })
```

Then we will pass the list of states, actions, and the returns into the training step (line 36-39). Then tensorflow will use the state and the weights to generate a new μ and σ. Then it will compute the log probability of the actions given the generate μ and σ.

```
1 # log probability of y given mu and sigma
2 log_pi = pi.log_prob(y, name='log_pi')
```

The cost function used is $J(\theta) = -\sum[G_t log\pi(A_t|S_t, \theta)]$. The program uses gradient descent to adjust the θ to minimize the cost function

```
1 # Returns is a 1 x (T-1) array for float (rewards)
2 Returns = tf.placeholder(tf.float32, name='Returns')
3 optimizer = tf.train.GradientDescentOptimizer(alpha)
4 train_op = optimizer.minimize(-1.0 * Returns * log_pi)
```

Part 2

Question

Answer

Part 3

Question

Answer