CSC411 - Project #4

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

April 1, 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.

```python
# 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')
```

```

# softplus is similar to ReLU. Activation function is g(x) = ln(1+e^x)
sigmas = tf.clip_by_value(fully_connected(
    inputs=hidden,
    num_outputs=output_units,
    activation_fn=tf.nn.softplus,
    weights_initializer=sw_init,
    weights_regularizer=None,
    biases_initializer=sb_init,
    scope='sigmas'),
    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.

```
# if we have the w's and b's saved, load it. Otherwise initialize it
if args.load_model:
    model = np.load(args.load_model)
    hw_init = tf.constant_initializer(model['hidden/weights'])
    hb_init = tf.constant_initializer(model['hidden/biases'])
    mw_init = tf.constant_initializer(model['mus/weights'])
    mb_init = tf.constant_initializer(model['mus/biases'])
    sw_init = tf.constant_initializer(model['sigmas/weights'])
    sb_init = tf.constant_initializer(model['sigmas/biases'])
else:
    hw_init = weights_init
    hb_init = relu_init
    mw_init = weights_init
    mb_init = relu_init
    sw_init = weights_init
    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 $pi_{s}ample$, 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 $ep_{r}ewards$ 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.

```
# log probability of y given mu and sigma
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

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

Part 2

Question Your job is to now write an implementation of REINFORCE that will run for the CartPole-v0 (source code here) environment.

In the Cart Pole task, two actions are possible applying a force pushing left, and applying a force pushing right. Each episode stops when the pole inclides at an angle that larger than a threshold, or when the cart moves out of the frame. The at each time-step before the episode stops, the reward is 1.

The policy function should have two outputs the probability of left and the probability of right. Implement the policy function as a softmax layer (i.e., a linear layer that is then passed through softmax.) Note that a softmax layer is simply a fully-connected layer with a softmax actication.

In your report, detail all the modifications that you had to make to the handout code, one-by-one, and briefly state why and how you made the modifications. Include new code (up to a few lines per modification) that you wrote in your report.

Answer The modifications we made are:

1. We changed the network model.
   In part 1, the network has a hidden layer which computes the feature vector of x, and pass the feature vector to one layer that computes μ, and another layer that computes σ, then use the output of μ and σ to generate π and use the π to generate action.
   Our network for part 2 is simpler, we don't have a hidden layer. We just have a fully connected layer that takes x as input, and output two output units. Then we use sigmoid as the activation function on the output to make the output in range of (0, 1), and then pass the output from activation function to softmax. The output of softmax is a size 2 vector that each element indicates the probability of an action.
   Therefore, we removed the setup of layers we don't need, and built our network as follows. (We also adjusted the value of α and γ.)

```
alpha = 1e-6
gamma = 0.99

try:
    output_units = env.action_space.shape[0]
except AttributeError:
    output_units = env.action_space.n

input_shape = env.observation_space.shape[0]
w = tf.get_variable("w", shape=[input_shape, output_units])
b = tf.get_variable("b", shape=[output_units])
x = tf.placeholder(tf.float32, shape=(None, input_shape), name='x')
y = tf.placeholder(tf.int32, shape=(None, 1), name='y')

layer1 = tf.sigmoid(tf.matmul(x, w)+b)
soft_max = tf.nn.softmax(layer1)
```

2. Since we have bernoulli distribution instead of the normal distribution used in part 1 for π, our π is the output of softmax. Our *pi_sample* gets the action of higher corresponding

probability from softmax. Out *log_pi* is the logarithm of softmax. And we use *act_pi* instead of *log_pi* to calculate the cost. We used the line of code which is provided to us to compute *act_pi*. The code converts y which is a list of actions of shape $n \times 1$ to the one hot encoding matrix which has shape $n \times 2 \times 1$, where n is the number of states. The code also add one more dimension to *log_pi* which converts the shape of *log_pi* from $n \times 2$ to $n \times 1 \times 2$. The result of multiplication of the two converted matrices is a $n \times 1 \times 1$ matrix which is a list of log probabilities of corresponding actions.
Then we used the *act_pi* to compute the cost.

```
pi_sample = tf.argmax(soft_max, axis=1)
log_pi = tf.log(soft_max)
act_pi = tf.matmul(tf.expand_dims(log_pi, 1), tf.one_hot(y, 2, axis=1))

Returns = tf.placeholder(tf.float32, name='Returns')
optimizer = tf.train.GradientDescentOptimizer(alpha)
train_op = optimizer.minimize(-1.0 * Returns * act_pi)
```

The rest of the code are mostly what we got from part 1.

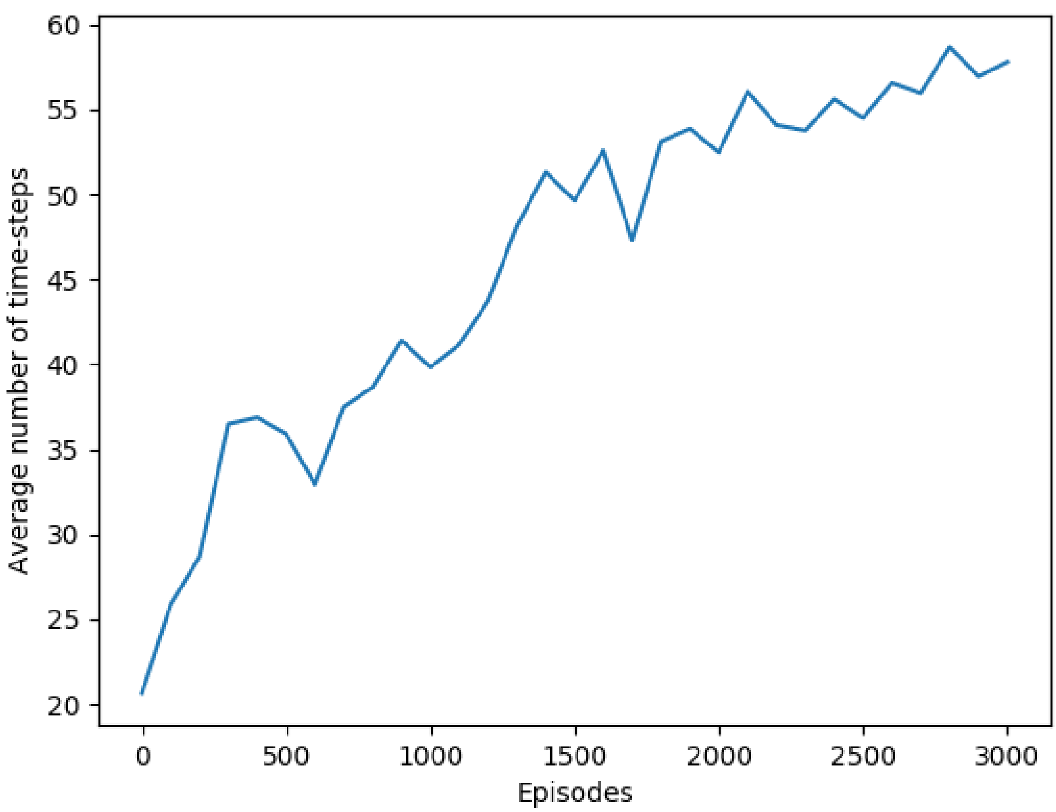
Part 3

Train CartPole using your implementation of REINFORCE. (Use a very small learning rate, and a discount rate $\gamma = 0.99$.) You do not have to train it until it works perfectly (i.e., the episode doesnt stop until time-step 200), but wait at least until the average number of time-steps per episode is 50. (Its better if you wait longer, although you are not required to do that.)

Part 3(a)

Question In your report, include a printout that shows how the weights of the policy function changed and how the average number of time-steps per eipsode change as you trained.

Answer The change of average number of time-steps per episode over the last 25 episodes are as follows:



Changes of Time-steps per Episode

The change of weights of the policy function are as follows:
Weight at episode 0:
[[0.55756891 0.44880387 0.45428711 0.01119721]
[0.60239559 -0.53826874 -0.63921934 0.78513044]]
Weight at episode 100:

[[0.55751288 0.46140882 0.4542748 -0.00796803]
[0.60192931 -0.55228651 -0.63857907 0.80743229]]
Weight at episode 200:
[[0.55744594 0.47939786 0.45419407 -0.03577379]
[0.60120898 -0.57182312 -0.63765335 0.83893186]]
Weight at episode 300:
[[0.55768639 0.51571321 0.45364195 -0.09334514]
[0.59695876 -0.61137104 -0.63584238 0.90098816]]
Weight at episode 400:
[[0.55669647 0.56195247 0.45266479 -0.17034987]
[0.59164733 -0.66522527 -0.6337384 0.98407555]]
Weight at episode 500:
[[0.55319363 0.61433756 0.45161384 -0.2601867]
[0.58770782 -0.72709823 -0.63237381 1.07760775]]
Weight at episode 600:
[[0.5493471 0.6586284 0.4501721 -0.33945304]
[0.58485538 -0.77985466 -0.63131249 1.1578542]]
Weight at episode 700:
[[0.54611695 0.70909899 0.44816312 -0.4303112]
[0.58224368 -0.83812582 -0.62907767 1.25205159]]
Weight at episode 800:
[[0.54470867 0.74866658 0.44712535 -0.49776825]
[0.58082283 -0.8807537 -0.62635362 1.32611656]]
Weight at episode 900:
[[0.54207498 0.80218232 0.44601309 -0.58896798]
[0.57786518 -0.94014865 -0.62367046 1.42442226]]
Weight at episode 1000:
[[0.53997892 0.85531533 0.44451818 -0.68093479]
[0.57664245 -0.9971112 -0.62065309 1.522493]]
Weight at episode 1100:
[[0.53804773 0.90899348 0.44291493 -0.77372855]
[0.57643706 -1.05087185 -0.61618227 1.62414622]]
Weight at episode 1200:
[[0.53436816 0.98216724 0.44084695 -0.90079987]
[0.57591343 -1.12369418 -0.61135685 1.75792503]]
Weight at episode 1300:
[[0.5304538 1.07761431 0.4385353 -1.06434882]
[0.57597333 -1.21745002 -0.60716885 1.92307317]]
Weight at episode 1400:
[[0.52535737 1.19890761 0.43557847 -1.27195978]
[0.57687956 -1.33788776 -0.6048944 2.12136078]]
Weight at episode 1500:
[[0.51959175 1.30228853 0.4316847 -1.45737672]
[0.57848126 -1.44308734 -0.60325855 2.29336071]]
Weight at episode 1600:
[[0.51401436 1.42097223 0.42739692 -1.66827142]
[0.58055174 -1.56275809 -0.60239846 2.48555636]]
Weight at episode 1700:

[[0.50750446 1.53061366 0.42203677 -1.87267733]
[0.58143383 -1.6764518 -0.60153365 2.66817904]]
Weight at episode 1800:
[[0.500332 1.65737295 0.41609058 -2.10924554]
[0.58377492 -1.80496466 -0.59965515 2.88181019]]
Weight at episode 1900:
[[0.49324197 1.77153444 0.41069824 -2.32351661]
[0.58694774 -1.92046285 -0.59812069 3.07449174]]
Weight at episode 2000:
[[0.48373222 1.88861907 0.40408787 -2.54988098]
[0.59000146 -2.04181719 -0.59656841 3.27574563]]
Weight at episode 2100:
[[0.47449186 2.02448845 0.3969498 -2.81081009]
[0.59316611 -2.1812861 -0.59471154 3.50856233]]
Weight at episode 2200:
[[0.46633637 2.13581109 0.3906489 -3.0269537]
[0.59572083 -2.29615307 -0.59274048 3.70167685]]
Weight at episode 2300:
[[0.45680553 2.26001072 0.38414222 -3.27014399]
[0.59872729 -2.42363763 -0.59078163 3.91966486]]
Weight at episode 2400:
[[0.44795224 2.381109 0.37753391 -3.50601459]
[0.60229701 -2.54769492 -0.5885542 4.13105249]]
Weight at episode 2500:
[[0.43924668 2.49611187 0.37183547 -3.7278161]
[0.60309607 -2.66729498 -0.58754814 4.32853603]]
Weight at episode 2600:
[[0.42905468 2.59504628 0.3656635 -3.929003]
[0.6061002 -2.77073932 -0.58561844 4.50803375]]
Weight at episode 2700:
[[0.41852862 2.70760155 0.35971603 -4.15303564]
[0.60998231 -2.88641882 -0.58367425 4.70958757]]
Weight at episode 2800:
[[0.40686017 2.81890082 0.35378745 -4.37579584]
[0.61278868 -3.00272322 -0.58225054 4.90878916]]
Weight at episode 2900:
[[0.39934814 2.92078543 0.34896737 -4.57286978]
[0.61418551 -3.10798955 -0.58114028 5.08576536]]
Weight at episode 3000:
[[0.39068386 3.01555085 0.34464303 -4.75818443]
[0.61748219 -3.20530534 -0.57989573 5.25346327]]

Part 3(b)

Question Explain why the final weights you obtained make sense. This requires understanding the what each input dimension means

Answer