CSC411 - Project #4

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

March 27, 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.

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.

Once everything is initialized, we will start training. For each iteration, we will reset the environment (line 118), then generate the states, actions, and rewards from time 0 to time T. When generating the actions, we will randomly sample a π_{sample} from the π normal distribution (line 133-135). Then based on the π_{sample}, we will generate the corresponding action, 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 ($ep_{s}tates, ep_{a}ctions$, and $ep_{r}ewards$). We will also keep track of the total discounted rewards using the variable G (line 137). 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 148). Thus $returns$ would be storing the total discouned rewards for each time from time 0 to $T-1$.

Then we will pass the list of states, actions, and the returns into the training step (line 154-157). Then tensorflow will use the state and the weights to generate a new μ and σ (line 63-93). Then it will compute the log probability of the actions given the generate μ and σ (line 102). 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 (line 105-107).

```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)

all_vars = tf.global_variables()

# use a Gaussian dist on pi because action is continuous
pi = tf.contrib.distributions.Normal(mus, sigmas, name='pi')
pi_sample = tf.tanh(pi.sample(), name='pi_sample')
```

Generating distribution on π

```python
# 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
```

Weight (θ) initialization

```
track_returns = []
for ep in range(16384):
    # reset the environment
    obs = env.reset()

    # generating all the states and actions and rewards
    G = 0
    ep_states = []
    ep_actions = []
    ep_rewards = [0]
    done = False
    t = 0
    I = 1
    while not done:
        ep_states.append(obs)
        env.render()
        # pi_sample is the list of randomly generated probablity
        # then we use pi_sample to generate the list of actions
        action = sess.run([pi_sample], feed_dict={x:[obs]})[0][0]
        ep_actions.append(action)
        obs, reward, done, info = env.step(action)
        ep_rewards.append(reward * I)
        G += reward * I # G is the total discounted reward
        I *= gamma

        t += 1
        if t >= MAX_STEPS:
            break
    # done generating

    if not args.load_model:
        # G_t = total - culmulative up to time t
        # set of all G_t's
        returns = np.array([G - np.cumsum(ep_rewards[:-1])]).T
        index = ep % MEMORY
```

States, actions, and rewards generation based on π_{sample}

Part 2

Question

Answer

Part 3

Question

Answer