CSC411: Project 4

Due on Tuesday, April 4, 2017

Zheng Yu Chen, Michael Pham-Hung

October 14, 2017

Part 1

From Sutton & Barto, we have the following pseudocode for the REINFORCE policy algorithm.

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
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\gamma^t G_t \nabla_{\boldsymbol{\theta}} \log \pi(A_t|S_t, \boldsymbol{\theta})$

Figure 1: REINFORCE Algorithm

Initializing Policy Function and Weights

From Figure 1, the differentiable policy function $\pi(A_t|S_t, \theta)$ is initialized as a normal distribution given μ and σ, defined from linear layers from lines 60 to 91 in the starter code.

Similarly, the policy weights and hidden layer parameters are initialized in lines 27 to 54.

Policy Gradient

Listing 1 shows a snippet of the starter code that corresponds the "repeat forever" loop from Figure 1. Within the while loop (line 11), state observations and actions are produced using the `env.step(action)` (line 16), with each time step of the episode, up to a maximum of 200 steps. Additonally, for each step, G_t is determined and discounted and then summed to a total reward in lines 18 and 19. The updating of the θ parameters are handled with TensorFlow in line 30, where the TensorFlow placeholders are defined in lines 56 to 97 in the original starter code.

Listing 1: Snippet of starter code

```
for ep in range(16384):
    obs = env.reset()

    G = 0
    ep_states = []
    ep_actions = []
    ep_rewards = [0]
    done = False
    t = 0
    I = 1
    while not done:
        ep_states.append(obs)
        #env.render()
        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
        I *= gamma

        t += 1
        if t >= MAX_STEPS:
            break

    if not args.load_model:
        returns = np.array([G - np.cumsum(ep_rewards[:-1])]).T
        index = ep % MEMORY


        _ = sess.run([train_op],
                     feed_dict={x:np.array(ep_states),
                                y:np.array(ep_actions),
                                Returns:returns })
```

Part 2

In this part, we wish to implement the REINFORCE algorithm described in Part 1 for the "CartPole-v0" environment in OpenAI Gym. Although the majority of the code will be similar, there are some inherent differences that will require modifications. Firstly, we are not using a hidden layer so we can safely remove the code that implements the hidden layer. This will be discussed further below.

After creating the environment, we make the training rate α smaller to a value of 1×10^{-5}. We noticed that this was necessary to get the algorithm to converge, which suggests that this environment is sensitive to smaller changes. Next, we changed the discount γ from 0.98 to 0.99. This was part of the instructions in the guidelines.

The most drastic change, as mentioned earlier, was removing the hidden layer as well as the layers for learning the μ's and σ's (See Listing 2). We replaced this architecture with a simple fully connected layer with two output units activated with a softmax layer. These two output units represent the probabilities of going left or right, respectively.

Listing 2: Defining the single linear layer with softmax output

```
mus = fully_connected(
    inputs=x,
    num_outputs=output_units,
    activation_fn=tf.nn.softmax,
    weights_initializer=mw_init,
    weights_regularizer=None,
    biases_initializer=mb_init,
    scope='mus')
```

The probabilities are then fed into a TensorFlow multinomial distribution to be sampled from during episode generations (See Listing 3). In the starter code, the actions were sampled from a normal distribution, but since the CartPole environment takes in a single action from a choice of two discrete actions, using a normal distribution did not make much sense.

Since the output layer was required to output two probabilities, a multinomial distribution made the most sense to use. However, a Bernoulli distribution variable would have also worked since we know that both probabilities sum to one.

Listing 3: Defining the multinomial distribution

```
pi = tf.contrib.distributions.Multinomial(p=mus,n=1.,name='pi')
pi_sample = pi.sample(name='pi_sample')
```

The final modification to the code was a minor change in the sampling of the action during each episode (See Listing 4). When sampling, the `sess.run` returns a 2x1 vector representing a one-of-2 encoding of the action. However, the environment takes in either a 0 or 1 as an action. Therefore, the one-of-2 encoding was appended to the list, but the first element was used for the actions.

Listing 4: Sampling the actions

```
while not done:
    ep_states.append(obs)
    if ep % 100 == 0:
      env.render()
    output = sess.run([pi_sample], feed_dict={x:[obs]})[0][0]
    action = int(output[0])
    ep_actions.append(output)
      ...
```

Part 3(a)

The following is a printout of how the weights of the policy function and mean number of steps changed as the agent was trained with an $\alpha = 1 \times 10^{-5}$. Note that for a different value of α, the average number of steps do fluctuate, indicating "ravines" withing the training data.

For an $\alpha = 1 \times 10^{-5}$, the agent stopped training after it reached 200 steps, five times in total.

```
Episode 0 finished after 10 steps with return 9.56179249912
Mean number of steps over the last 25 episodes is 10.0
incoming weights for the mu's from the first hidden unit:
[[ 0.22189391  1.16820109]
 [ 0.93455452 -0.03667339]
 [-0.01391068  0.23235601]
 [ 1.31279278  0.29721767]]
Episode 100 finished after 34 steps with return 28.9446772728
Mean number of steps over the last 25 episodes is 46.52
incoming weights for the mu's from the first hidden unit:
[[ 0.09089724  1.29919767]
 [ 0.47358927  0.42429188]
 [ 0.03318011  0.18526526]
 [ 1.86034942 -0.25033867]]
Episode 200 finished after 64 steps with return 47.4403512474
Mean number of steps over the last 25 episodes is 84.48
incoming weights for the mu's from the first hidden unit:
[[ 0.54071873  0.8493762 ]
 [ 1.62985456 -0.73197329]
 [ 0.34817681 -0.12973146]
 [ 2.44792747 -0.83791655]]
Episode 276 finished after 200 steps with return 86.6020325142
Mean number of steps over the last 25 episodes is 111.72
incoming weights for the mu's from the first hidden unit:
[[ 0.37455574  1.01553917]
 [ 1.65997195 -0.76209062]
 [ 0.63444477 -0.41599941]
 [ 3.21391726 -1.60390699]]
```

Part 3(b)

Using the CartPole source code, we can take a look to see what the four input variables actually represent. From the source code,

```
x, x_dot, theta, theta_dot = state
```

The variables on the left hand side of the equation represent the four input variables, or features. Knowing what each feature represents, we can analyze the weights that we have obtained to see if they make sense. The final weights, after 900 episodes of training, are:

```
[[-0.87293446  2.26302886]
 [ 3.27809143 -2.38021255]
 [ 1.93965244 -1.72120702]
 [ 4.73692036 -3.12690759]]
```

Given that the goals of the cart are to: a) Balance the pole and b) Stay on the screen, we can see that weights make sense. Let us define, for each variable in the state vector, that left is positive and right is negative in value. Additionally, the first and second column correspond to probability of going left and right respectively.

Looking at the first weight for each output, which correspond to the position of the cart, we say that being left of the center of screen, a positive 'x' value, makes going right more probable. Looking at the third and fourth weights, corresponding to θ and $\dot{\theta}$, if the rod is leaning and falling towards left, it is more likely to make the cart go left. However, if the the rod was leaning towards the left, but was going in the right direction (i.e. returning to the center), the cart is more likely to go right. This can be thought of as a dampening of the motion. Looking at the second weight, if the cart was already going left, it is more likely that the cart will want to keep going left, as a way to counteract the tendency for the cart to move in the opposite direction once the rod starts centering.

In conclusion, the cart has learned its policy in the same manner as one would intuitively balance a pole, upright, on a hand.