Reinforcement Learning with Policy Gradients

Hao Sheng

April 1^{st} 2017

Part 1

Below is the pseudo code which reinfornce will be compared to:

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: Pseudocode on p. 271 of Sutton Barto

[**Step 1**] **Input**: The code begins by initializing a neural network with a single hidden layer and outputs of a mean and deviation. These outputs are then used to create π , which is a normal probability distribution using each mean and deviation pair of the neural network. A continuous probability distribution is required because the output can be a continuous range of values.

```
x = tf.placeholder(tf.float32, shape=(None, NUM_INPUT_FEATURES), name='x')
y = tf.placeholder(tf.float32, shape=(None, output_units), name='y')
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')
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')
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()
pi = tf.contrib.distributions.Normal(mus, sigmas, name='pi')
```

[**Step 2**] **Initialize policy weights** θ: Initializing the weights and biases of the neural network.

```
weights_init = xavier_initializer(uniform=False)
relu_init = tf.constant_initializer(0.1)
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
```

[Step 3] Generate an episode:An episode must be simulated to generate rewards in order to assess the current performance of the neural network and train it.

```
done = False
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
```

[Step 4] For each step of the episode...:Using the rewards generated from the previous step, the reward at each time step is calculated. The neural network is then trained by maximizing the expected value of the total reward from the episode. In the episode, the reward gained at each time step is multiplied by the probability by taking that action at that time step, representing an expected return. The returns from all the time steps are summed together and represent the loss function that is then maximized by gradient descending on the negative loss.

Code for generating the returns at each time step, this represents G_t. Moreover tensorflow is used to train the neural net a single time using the history of the episode just performed (history including all states, actions, and returns at each time state).

```
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 })
```

Code for calculating the loss function and using the "GradientDescentOptimizer" to minimize the negative loss. This last line of the below code updates θ or the weights of the neural network.

```
pi = tf.contrib.distributions.Normal(mus, sigmas, name='pi')
pi_sample = tf.tanh(pi.sample(), name='pi_sample')
log_pi = pi.log_prob(y, name='log_pi')
Returns = tf.placeholder(tf.float32, name='Returns')
optimizer = tf.train.GradientDescentOptimizer(alpha)
train_op = optimizer.minimize(-1.0 * Returns * log_pi)
```

Part 2

Because the cart pole environment requires a discrete action of either moving the cart left or right, this is slightly different than the reinforce program given for the bipedal walker which had actions that were continuous.

First, the fully connected neural network used had no hidden layers, four inputs, and two outputs. The two outputs would go through a softmax and represent the probability of moving left and right, respectively.

```
NUM_INPUT_FEATURES = 4
x = tf.placeholder(tf.float32, shape=(None, NUM_INPUT_FEATURES), name='x')
y = tf.placeholder(tf.float32, shape=(None, output_units), name='y')
mus = fully_connected(
    inputs=x,
    num_outputs=output_units,
    weights_initializer=mw_init,
    weights_regularizer=None,
    biases_initializer=mb_init,
    scope='mus')
all_vars = tf.global_variables()
pi = tf.nn.softmax(mus)
```

When making the decision to move the cart left or right during the episode, the probability generated from this softmax is used. Note that we do not always choose the action with the higher probability, rather we make a random choice using the probability of the softmax.

```
action = sess.run([pi], feed_dict={x:[obs]})[0][0]
action = np.random.choice([0,1], 1, p = action)[0]
```

The action performed at the time step is saved using one hot encoding where moving left is [1, 0] and moving right is [0, 1].

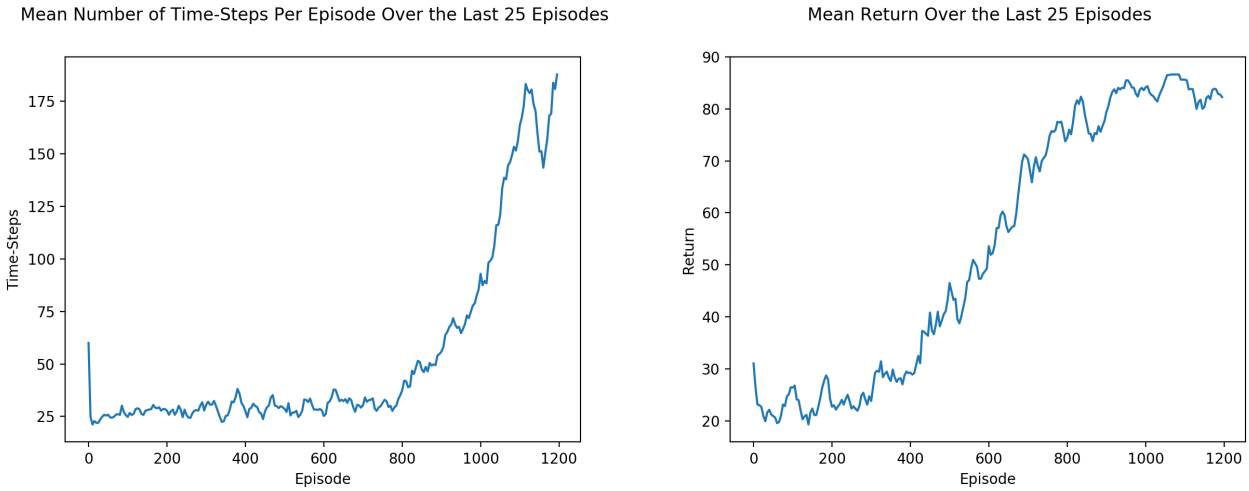
```
if action == 0:
    ep_actions.append([1,0])
else:
    ep_actions.append([0,1])
```

Lastly, to train the network, at each time step we multiply the probability of taking that action by the reward that was gained by that action. Adding up these values represents the expected return of the episode which is our loss that we want to maximize. Or minimize the negative loss.

```
log_pi = tf.log(pi, name = 'log_pi')
act_pi = tf.multiply(y, log_pi)
act_pi = tf.reduce_sum(act_pi, 1, keep_dims=True)
Returns = tf.placeholder(tf.float32, name='Returns')
optimizer = tf.train.AdamOptimizer(alpha)
train_op = optimizer.minimize(-1.0 * tf.reduce_sum(tf.multiply(Returns, act_pi)))
```

Part 3a

Below is the performance of the cart-pole reinforce:



Below is a chart of the weights at every 500 episodes:

Episode	Weights
0	[[0.23180483, 1.15829015] [0.92399001, -0.02610886] [-0.02379543, 0.24224076] [1.32377374, 0.2862367]]
500	[[0.78310746, 0.50532198] [1.66312623, -0.2994574] [-1.54856813, 1.77980232] [-0.14629382, 1.55668616]]
1000	[[0.20310758, 0.56094581] [0.23202533, 0.89074641] [-4.48087645, 4.46624088] [-0.89624655, 2.29393148]]
1500	[[-0.08716535, 0.53853488] [-0.03441087, 1.14508462] [-5.2997508, 5.17977571] [-1.35151017, 2.4073782]]
2000	[[-0.34275436, 0.64334005] [0.25091976, 0.80668879] [-5.0533762, 4.92502642] [-1.74215651, 2.79130507]]

Part 3b

The four parameter that describe the state are x, x_dot, theta and theta_dot, representing the horizontal position of the pole, the horizontal velocity of the pole, the angle of the pole relative to the vertical line and the angular velocity of the pole.

To maximize the reward, we want to keep the pole within the screen width as long as possible which means if the pole is on the right side of the screen, the algorithm will want it to move to the left and vice versa. Intuitively, to make the pole go back to center, we want the cart to be further away from the centre than the pole.

After 2000 episodes, the reinforce learning learned [-0.34275436, 0.64334005] as weights for position, input x. Based on the how the algorithm was constructed, a positive position input(right half of the screen) will increase the chance for the cart to move to the right, and a negative one will increase the chance for the cart to move to the left.

Intuitively, to stop the pole from moving towards one direction, we want the cart to be ahead of the pole in that direction. As for the velocity input node, we observed [0.25091976, 0.80668879] as the weights. Since 0.80668879 is greater than 0.25091976, if the input(velocity) is positive(moving to right), the algorithm increases the chance for the cart to move to the right. On the other hand, if the input(velocity) is negative(moving to left), the algorithm decrease the chance for the cart to move to the right.

The third input node represent the angle of the pole relative to vertical line, positive means the pole to left of the line and negative means the pole to the left of the line. The weights, after 2000 episodes, are [-5.0533762, 4.92502642] which indicate that when the pole is tilting toward right, the algorithm would more likely to move the cart to the right, preventing bigger angular displacement and vice versa.

Similar to the third input node, the algorithm also learned a negative and a positive weights for angular velocity input (the fourth node). The weights makes sure the cart would more likely to move towards the direction that prevents the pole from further tilting.