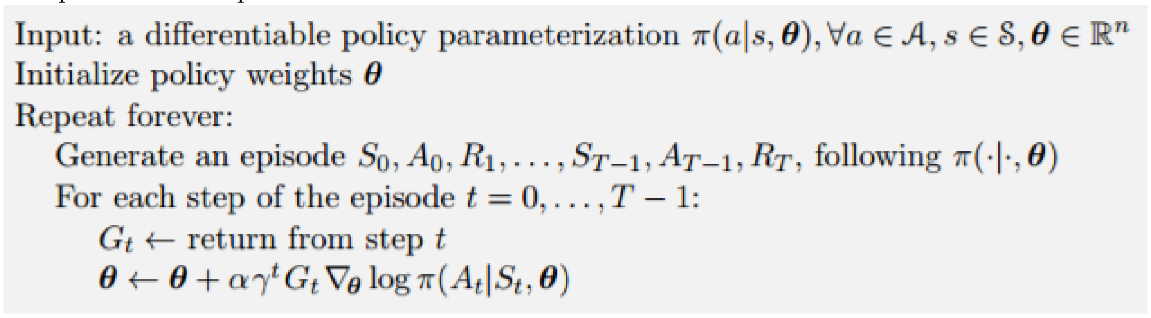
CSC411 Project 4: Reinforcement Learning with Policy Gradients

Dawn Duan and Shamama Khattak

April 6, 2017

1 Part 1

The pseudocode on p. 271 of Sutton Barto is as follows:



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

The precise explanation of why the code corresponds to the pseudocode is as follows.

1. **Declare all the input variables, i.e. the policy variable π, action variable a, state variable s.**

   $\pi_\theta(a|s)$ takes state s at time t, or at the current state, as input features and outputs the action the agent should pursue at the next state, s_{t+1}

   ```
   action = sess.run([pi sample], feed dict={x:[obs]})[0][0]
   ```

   Here, a continuous Gaussian policy is adopted here because the metion of the bipedal walker is continuous.

   ```
   pi = tf.contrib.distributions.Normal(mus, sigmas, name='pi')
   ```

2. **Initialization policy weights θ.**

   ```
   weights_init = xavier_initializer(uniform=False)

   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

   ```

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

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)
```

3. **Start the loop.**

```
for ep in range(16384):
```

The range, 16384 is the large arbitrary value which we want to consider as the last episode.

4. **Generate an Episode $S_0, A_0, R_1, ..., S_{T-1}, A_{T-1}, R_T$, following $\pi(.|., \theta)$.** The declaration of the terms is presented below, where the subscript means the time-step.

```
# initilization
    obs = env.reset()

    G = 0
    ep_states = []
    ep_actions = []
    ep_rewards = [0]
    done = False
    t = 0
    I = 1
```

5. **For each step of the episode $t = 0, ..., T-1$.**

The declaration of the variable, time-step t is shown below. The last time-step, i.ei. T in the pseudocode, corresponds to the MAX STEPS defined from the *env*. That is, when the done flag is set to True, whichever comes first:

```
MEMORY=25
MAX_STEPS = env.spec.tags.get('wrapper_config.TimeLimit.max_episode_steps')
```

```

t = 0
while not done:
    ...
    t += 1
    if t >= MAX_STEPS:
        break
```

6. **Get G_t, is the actual total reward in each episode starting from time t, i.e. discounted.**

$$G_t = \sum_{i=t} \gamma^i r_i$$

In this case, starting from time t = 0

```
gamma = 0.98
'''iterate through all the episode'''
G = 0
t = 0
I = 1 # discounted rate

obs, reward, done, info = nev.step(action) # gives out the reward value at each action step, tell you weather goal stae
G += reward * I #update the actual reward at each step
I *= gamma #update discounted rate
t += 1 #update time step
```

Here a difference between the pseudo code and the real implementation is presented. That is, the actual reward G is supposed to get updated at each time-step for each episode. In the actual implementation, the actual reward is only override in each episode. For the actual reward in each time-step, it is implemented using np.cumsum() to compute the reward $ep_rewards$, for time-step 0 up to time-step T.

```
    # Updates for the rewards
    if not args.load_model:
        returns = np.array([G - np.cumsum(ep_rewards[:-1])]).T
```

7. **Update θ using gradient ascend, i.e. the following formula:**

$$\theta \leftarrow \theta + \alpha\gamma^t G_t \nabla_\theta log(\pi_\theta(A_t|S_t))$$

The learning rate α is specified as 0.01 here.

```
alpha = 0.01

optimizer = tf.train.GradientDescentOptimizer(alpha)
train_op = optimizer.minimize(-1.0 * Returns * log_pi) # the update to theta
# -1.0 because it's gonna be adding for gradient descend

# the relevant values are feed into the tensor per episode
    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 })
```

The gradient is computed from Policy Gradient Theorem. It consists three part:

(1) G_t is computed from the previous step, Step5
(2) $\nabla_\theta log(\pi_\theta(A_t|S_t))$ is presented as follows:

```python
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')#insert tensor y

train_op = optimizer.minimize(-1.0 * Returns * log_pi) # the update to theta

# loop through eps
action = sess.run([pi_sample], feed_dict={x:[obs]})[0][0]
obs, reward, done, info = env.step(action) #obs are the states
```

2 Part 2

In this section, we are required to write a reinforcement learning algorithm for the "CartPole-v0" environment. In order to accomplish this task, our group made the modifications detailed below to the REINFORCE reinforcement learning algorithm used for the "BipedalWalker-v2" environment.

Previously, in the "BipedalWalker-v2" env, we computed the policy function, π_θ, as a single hidden layer neural network. Since the "CartPole-v0" environment defines two discrete actions (move left or move right), to obtain a good policy function and make REINFORCE work, we had to do the following:

1. **Change from Normal to Bernoulli Distribution**

   Since we are using reinforcement learning, we want to use a stochastic policy that, given an action a and a state s, gives us the probability of taking action a from state s at time step t. That is we aim to have a policy, $\pi_(a|s) = P(A_t = a, S_t = s)$

   Previously, since we had a continuous set of actions, we used the gaussian policy. Now that we have a discrete set of actions, we must now use a softmax probability.

   More formally, we aim to change from: $a \sim N(f_\theta(s))$ to $a \sim Bernoulli(f_\theta(s))$. That is, we need to change from a Normal distribution to a bernoulli distribution.

   In code this change corresponds to (and resulted in the change of):

```python
# Define policy function to be a Bernoulli distribution
pi = tf.contrib.distributions.Bernoulli(p=dir_probabilities, name="pi")

pi_sample = pi.sample()
```

2. **Change the function $f_\theta(s)$**

   We also changed the function $f_\theta(s)$ so that instead of being a neural network with one hidden layer, it is now a fully connected network with no hidden layers. It uses the softmax activation function and has 2 output units which correspond to the actions that we can take. The outputs represent the probability of taking action left or taking action right. Our network takes in x which is a NONE (variable) X 4 matrix. We chose to use a NONE X 4 matrix because we have 4 numbers in our observation space as found by executing the command: env.observation_space.shape[0]. The code corresponding to the change is shown below:

```python
# Gets observation space set size
Y_UNITS = 2
input_shape = env.observation_space.shape[0]
```

```
NUM_INPUT_FEATURES = 4
x = tf.placeholder(tf.float32, shape=(None, NUM_INPUT_FEATURES), name='x')
y = tf.placeholder(tf.float32, shape=(None, Y_UNITS), name='y')

# Obtain probabilities of taking action left and right
dir_probabilities = fully_connected(
    inputs=x,
    num_outputs=output_units,
    activation_fn=tf.nn.softmax,
    weights_initializer=w_init,
    weights_regularizer=None,
    biases_initializer=b_init,
    scope='dir_probabilities')
```

3. **Other changed lines:**

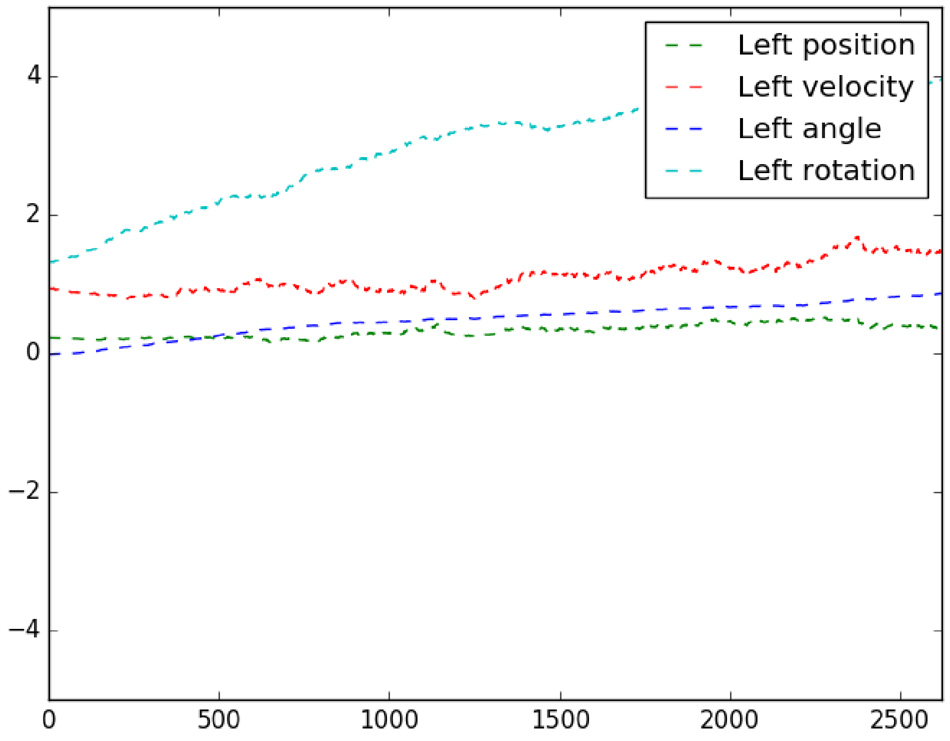
   The following lines were changes so that the training part would work with the changes we made above:

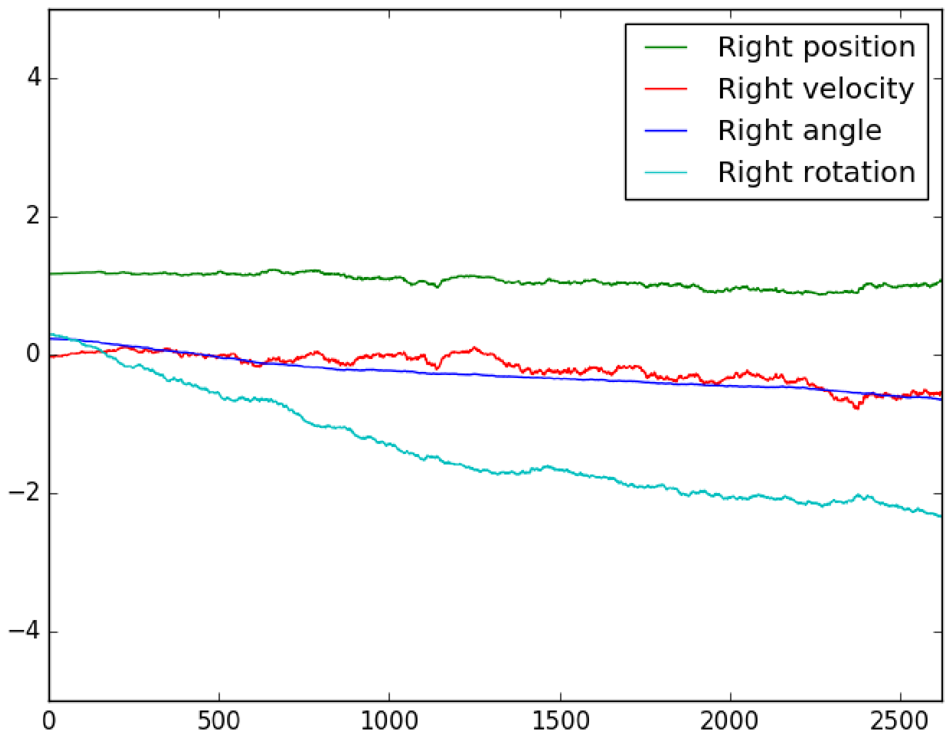
```
1. MAX_STEPS = 200
2. obs, reward, done, info = env.step(action[0])
3. _ = sess.run([train_op],
                    feed_dict={x:np.array(ep_states),
                                y:np.vstack(np.array(ep_actions)),
                                Returns:returns })
```

3 Part 3

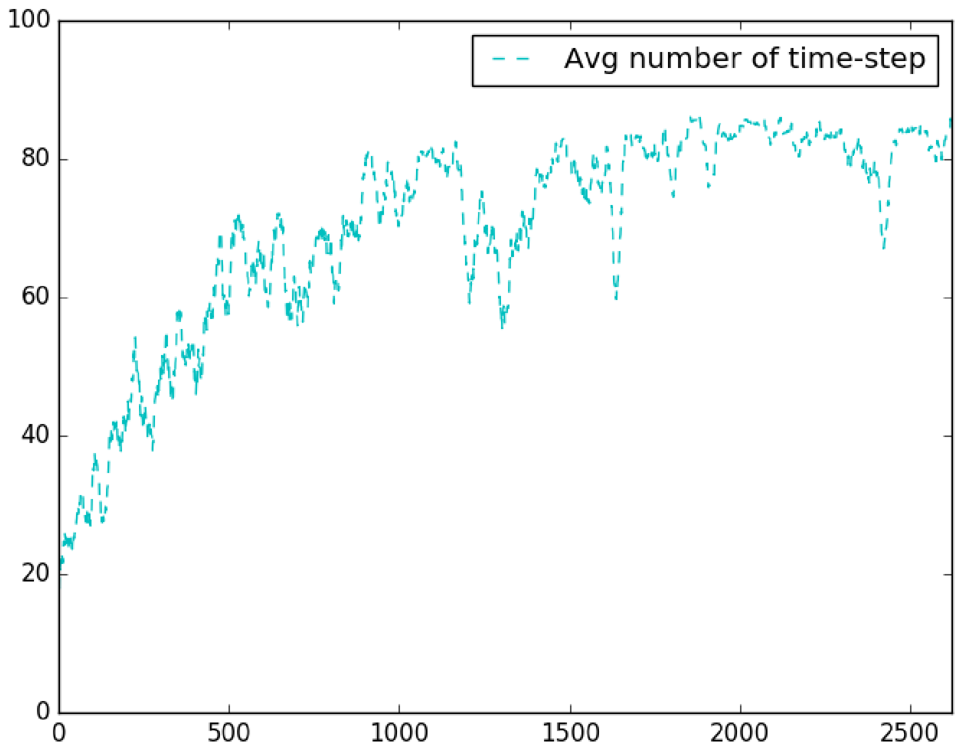
3.1 Print-out of weights

The printout that shows how the weights of the policy function changed for both output units are as follows:





The printout that shows how the average number of time-steps per eipsode (over e.g. the 25 last episodes) changed as we trained the agent is as follows:



3.2 Explanation of weights

The four input dimensions correspond to:

1. The position of the cart
2. The velocity of the cart
3. The angle of the pole
4. The rotation rate of the pole

The final weights that we obtain are:

```
[[ 0.20637971,   1.18371701],
 [ 0.95413405, -0.05625207],
 [ 0.37281665, -0.15437123],
 [ 2.46482682, -0.85481673]]
```

In order to understand why the weights listed above make sense, we must first consider what each component of the 2 x 4 matrix corresponds to. Each of the 2 columns represents the weights for each of the output units. Each row corresponds to a feature of our observation state space. Then, using this line of reasoning, it can be inferred that the weights at index 1, 1 is the weight that corresponds to the position of the cart for the first output unit.

Let's consider each of the features of the observation space individually in order to deduce why these values make sense:

1. **Feature: Position of the cart**

   The weights for the position of the cart is: [0.20637971   1.18371701]

   Consider that the position on the cart is on a scale from -5 to 5, where the negative axis indicates that the cart is on the left of the center and positive implies that we are to the right of the center. Then the position weight **multiplied** by the position state for output 1 will be higher than the position state times the position weight for output 2 when the cart is to the left of the center. When the weight is to the left of the center, the position described by the input is negative. The output 1 weight is smaller (0.20637971) than the output 2 weight (1.18371701) and the effect of multiplying a large negative number by a small positive number results in a smaller outcome than multiplying a large negative number by a large positive number.

   Thus, we can say that output unit 1 is favoured (i.e., larger than output unit 2) when the cart is to the left and output unit 2 is favoured when the cart is to the right.

2. **Feature: Velocity of the cart**

   The weights for the velocity of the cart is: [0.95413405, -0.05625207]

   These weights imply that the velocity of the cart will be higher in output unit 1 when the input state velocity is positive (since the weight is positive and a positive number multiplied by a positive number is greater than a positive number multiplied by a negative number). Conversly, output unit 2 will be favoured when the input state velocity of the cart is negative (implying that the cart is moving towards the left).

3. **Feature: Angle of the pole**

   The weights for the angle of rotation is: [0.37281665, -0.15437123]

   When the input state angle of the pole is positive (tilting to the right) we will favour output unit 1 (by similar logic to the weights above). When the the input state angle of the pole is negative (i.e., tilting to the left), we will favour output unit 2.

4. **Feature: Rotation rate of the pole**

   The weights for the angle of rotation is: [2.46482682, -0.85481673]

> When the input state's rotation rate of the pole is negative (i.e., the pole is falling to the left) that means we are likely to favour output unit 2. When the input state's rotation rate of the pole is positive (falling to the right), we are very strongly likely to favour output unit 1.

Putting all of this together, what we find is that output unit 1 is more likely to be favoured (greater than output unit 2) when the input for our weight is as follows: the cart is to the left of the center, moving with a positive velocity to the right with the pole leaning to the right, and falling to the right. Intuitively, the combination of these parameters as weights for output unit 1 makes sense. In the steady state, we want to occasionally try to oppose the action given so that we can stabilize ourselves. Using similar logic, we can come up with a similar deduction for output unit 2.