# CSC411: Project 4
# Reinforcement Learning with Policy Gradients

Due on Tuesday, April 4, 2017

**Michelle Leung and Tianyi Yu**

February 23, 2018

# Part 1

*Reinforce Algorithm for BipedalWalker-v2*
The handout code is a direct implementation of the pseudo-code from Sutton & Barto. The except is shown below, following it are python snippets that demonstrate the correspondence of the BipedalWalker code to the reference algorithm:

REINFORCE, A Monte-Carlo Policy-Gradient Method (episodic)
Input: a differentiable policy parameterization $\pi(a|s,\theta), \forall a \in A, s \in S, \theta \in R^n$
Initialize policy weights $\theta$
Repeat forever:
    Generate an episode $S_t, A_t, R_t$, following $\pi(a|s,\theta)$
    For each step of the episode t = 0, ... T-1:
        $G_t \leftarrow$ return from step t
        $\theta \leftarrow \theta + \alpha\gamma^t G_t \nabla_\theta log\pi(A_t|S_t,\theta)$

**Input: a differentiable policy parameterization:**
Code (Lines 57-93):

```
NUM_INPUT_FEATURES = 24
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()

# policy pi computed by constructing normal distribution from output of neural network, one distribu
pi = tf.contrib.distributions.Normal(mus, sigmas, name='pi')
# action performed by sampling pi
pi_sample = tf.tanh(pi.sample(), name='pi_sample')
# training done from log of the policy
log_pi = pi.log_prob(y, name='log_pi')
```

**Initialize policy weights:**
Code (Line32):

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

**Repeat forever:** Code (Line 106):

```
# repeat for 16384 episodes instead of forever
for ep in range(16384):
```

**Generate an episode:** Code (Lines 107-115):

```
# reset the environment
obs = env.reset()

# initialize total returns
G = 0
# states
ep_states = []
# actions taken
ep_actions = []
# culmulative rewards at each timestep
ep_rewards = [0]
# done flag
done = False
# timestep
t = 0
# discount rate
I = 1
```

**For each step of the episode:**
Code (Lines 116-138):

```
while not done:
    ep_states.append(obs)
    env.render()
    # get an action following policy pi
    action = sess.run([pi_sample], feed_dict={x:[obs]})[0][0]
    ep_actions.append(action)
    # take the action, track the reward and observations
```

```
    obs, reward, done, info = env.step(action)
    ep_rewards.append(reward * I)
    # discount reward
    G += reward * I
    I *= gamma

    t += 1
    if t >= MAX_STEPS:
        break

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

    # train using observations, actions taken, log policy and discounted rewards
    _ = sess.run([train_op],
                 feed_dict={x:np.array(ep_states),
                            y:np.array(ep_actions),
                            Returns:returns })
```

**Calculate returns:**
Code (Lines 109, 121-123, 131):

```
# initialize rewards at start of episode
G = 0

# run a step, retrieve rewards
obs, reward, done, info = env.step(action)
ep_rewards.append(reward * I)
G += reward * I

# feed returns into optimizer
returns = np.array([G - np.cumsum(ep_rewards[:-1])]).T
```

**Update theta:**
Code (Lines 96, 123-124, 97):

```
# alpha = 0.000001
optimizer = tf.train.GradientDescentOptimizer(alpha)

# gamma^t G_t
G += reward * I
I *= gamma

# grad_theta (log pi(A|S, theta)), multiply by discounted rewards
train_op = optimizer.minimize(-1.0 * Returns * log_pi)
```

# Part 2

*CartPole-v0 REINFORCE*

Using the handout code for Bipedal Walker REINFORCE, the following modifications were made:

To make the code run `CartPole-v0` instead of `BipedalWalker-v2`, the following line of code was changed

```
env = gym.make('CartPole-v0')    #previously env = gym.make('BipedalWalker-v2')
```

The number of hidden units had to be changed to match the number of actions (due to the policy function being one-hot), thus, the following change was made

```
hidden_size = 2 #previously hidden_size = 64
```

Then, to accommodate for the changed observation dictionary size (corresponding to the position, velocity, angle, and rotation speed of the pole), the number of input features had to be changed to four in order to feed a value of shape (1,4) into Tensor 'x:[obs]' which originally had shape (?,24)

```
NUM_INPUT_FEATURES = 4  #previously NUM_INPUT_FEATURES = 24
```

Next, a single softmax layer replaced the relu layer in Bipeddal Walker, and all mention of the mus and sigmas were discarded

```
hidden = fully_connected(
    inputs=x,
    num_outputs=hidden_size,
    activation_fn=tf.nn.softmax,    #previously activation_fn=tf.nn.relu
    weights_initializer=hw_init,
    weights_regularizer=None,
    biases_initializer=hb_init,
    scope='hidden')
```

As a normal distribution using mus and sigmas is no longer in use, the following lines were changed in order to accommodate for a Bernoulli distribution

```
pi = tf.contrib.distributions.Bernoulli(p=hidden, name='pi')
pi_sample = pi.sample()
#previously     pi = tf.contrib.distributions.Normal(mus, sigmas, name='pi')
#and            pi_sample = tf.tanh(pi.sample(), name='pi_sample')
```

Since the policy function has two outputs for 'left' and 'right', represented by the probability of each occurring, the following line was changed in order to determine the action based only on the left set of weights; if the action was a 0, then a left movement would occur, but if the action was a 1, then a right movement would occur

```
obs, reward, done, info = env.step(action[0])
#previously     obs, reward, done, info = env.step(action)
```

Using the above modifications, with `alpha=0.0001` and `gamma=0.99`, CartPole REINFORCE reaches a mean of 50 time-steps (39.5 steps with the discount rate of 99%) in 150 episodes.

# Part 3

*Weights of the Policy Function*

(a) The following is a printout of the weights of the policy function and the average number of time-steps changed for a selection of episodes:

```
Episode 0 finished after 16 steps with return 14.854222890512432
Mean return over the last 25 episodes is 14.854222890512432
incoming weights:
 [[ 0.22191776  1.16817725]
 [ 0.93468541 -0.03680427]
 [-0.01397642  0.23242176]
 [ 1.31265676  0.29735366]]
Episode 200 finished after 37 steps with return 31.055091413092182
Mean return over the last 25 episodes is 41.99719966269536
incoming weights:
 [[ 0.21634713  1.1737479 ]
 [ 0.83159864  0.06628263]
 [ 0.07192611  0.14651923]
 [ 1.68246746 -0.07245685]]
Episode 400 finished after 119 steps with return 69.75955643309779
Mean return over the last 25 episodes is 49.36020773018882
incoming weights:
 [[ 0.23595829  1.15413654]
 [ 0.8809585   0.01692276]
 [ 0.18039188  0.03805341]
 [ 2.03197455 -0.42196399]]
Episode 600 finished after 106 steps with return 65.53878166524818
Mean return over the last 25 episodes is 64.55923018939211
incoming weights:
 [[ 0.24281125  1.14728427]
 [ 1.04575849 -0.14787683]
 [ 0.31841075 -0.09996534]
 [ 2.25216007 -0.64215058]]
 Episode 800 finished after 36 steps with return 30.358678195042607
Mean return over the last 25 episodes is 67.22769193261021
incoming weights:
 [[  2.07403407e-01   1.18269324e+00]
 [  8.96835864e-01   1.04615046e-03]
 [  4.03632075e-01  -1.85186595e-01]
 [  2.65552306e+00  -1.04551244e+00]]
```

Notice that the last weight in the left column increases as the mean return time-steps increases as CartPole is being trained by the REINFORCE Algorithm, but the first and third weight in the left-hand column remain small compared to the second weight of the left side.

(b) The final weights are of a 4x2 matrix, where each of the four rows represents position, velocity, angle, and rotation speed of the pole, respectively. When these weights are matrix multiplied by the observation matrix, which is of 1x4 dimension, thus producing 1x2 matrix, for which we take the value of the first element, 0 or 1, representing the inclination to move the cart in either the left or right directions.
If we look at the left weights of an episode early into training:

```
Episode 50 finished after 25 steps with return 22.21786406008531
Mean return over the last 25 episodes is 26.917420909182493
incoming weights:
 [[ 0.21322857]
 [ 0.90245253]
 [-0.00677486]
 [ 1.36480558]]
```

Versus the left weights of an episode far into training:

```
Episode 4350 finished after 193 steps with return 85.6255062846374
Mean return over the last 25 episodes is 81.6416210835355
incoming weights:
 [[ 0.04391694]
 [ 1.52089691]
 [ 1.11899269]
 [ 4.24980736]]
```

We see that the REINFORCE algorithm puts the rotation speed (the fourth weight) of the pole as a much higher priority above all the other input parameters, which makes sense as the faster the pole rotates in either direction, the more out of control the system is and the harder it will be to correct it. It is almost not concerned with the position of the pole, indicated by the first weight, and slightly more concerned about the velocity over the angle of the pole, which all make sense as the position and angle are not as hard to correct as the velocity and rotational speed once the system has been set in motion.