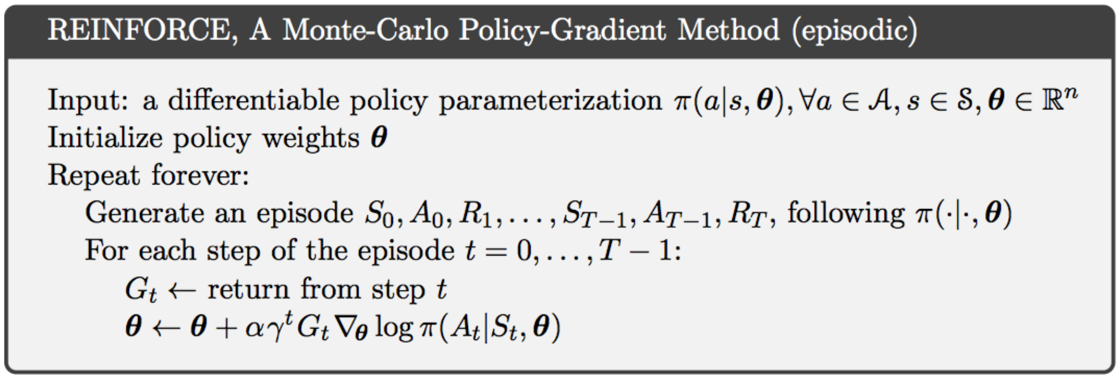
CSC411: Reinforcement Learning using Policy Gradient

Yoshiki Shoji & **Zi Mo Su**

April 4, 2017

Part 1



```
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.

The input specified in the pseudocode is the policy function which is implemented as a single hidden-layer neural network. This input can be provided as a set of arguments in *bipedal-reinforce.py lines 29-36* (see Appendix A for full *bipedal-reinforce.py* code):

```python
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'])
```

These arguments specify the initial weights and biases of the neural network. If this is not specified, these values will be randomly generated. These are the initialized policy weights.

The policy function itself is created and initialized as the neural network in *bipedal-reinforce.py lines 52-94*:

```python
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')
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)

sess = tf.Session()
sess.run(tf.global_variables_initializer())
```

The policy function can be seen to use the Gaussian model with its respective means and variances. The gradient policy is furthermore stored as a variable called train_op which uses another variable called optimizer.

After the policy function is initialized, we consider the heart of the code when it enters the for and while loops to complete reinforcement learning. Consider the code given below (*bipedal-reinforce.py lines 100-130*):

```
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
```

```
25      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),
30                                              y: np.array(ep_actions),
                                                Returns: returns})
```

The code starts by considering an agent starting at time, $t = 0$ which will repeat a total of 16384 episodes. Whenever we loop back to the beginning the for-loop, we reset the state of the agent to begin a new episode. As such, each iteration of the for-loop generates an episode following the Gaussian policy. The while-loop executes actions over the time-steps of the episode and completes when a step returns *done* $= True$ in *line 16* or if the maximum number of steps has been reached. Let us now enter the while loop which enables us to compute the total amount of discounted rewards.

$$G_t = R_t + \gamma R_{t+1} + ... + \gamma^n R_{t+n}$$

We obtain G_t by storing the discounted rewards in the array *ep_rewards* and updating the variable G in each iteration of the while-loop. G in this case is equal to G_0 at the end of the while-loop. The array of discounted cumulative sums $G_t \; \forall \; t \in 0, .., n$ is computed in *line 26* above as a vectorized equation.

After we obtain G_t, we must now update our θ variable. This is achieved by first considering the equation below which is a generalized form of updating our θ value:

$$\theta_{t+1} = \theta_t + \alpha\gamma^t G_t \nabla_\theta log \pi(A_t|S_t, \theta_t)$$

This part is achieved by considering the final three lines of code above.

By running *train_op* in our TensorFlow session, the weights $(\theta) areupdated.Wefeedin$G$_t$ which was obtained prior, as well as the input states and the actions taken, which were stored during the while-loop's execution. In our neural network, the variable $train_op$ minimizes the negative of the total reward by modifying the weights in the neural network. This is equivalent to maximizing the total reward.

Part 2

The reinforcement algorithm will utilize the pseudocode given in Part 1. We note that the policy function will be a discrete softmax policy using a Bernoulli distribution as opposed to a continuous Guassian policy:

$$actions \sim Bernoulli(f_\theta(s))$$

This enables us to mimic the actions wherein there is a force applied to the right or left of the cart. By using the Bernoulli distribution, we can compute the probability of the two actions, where $P(a = 1) = f_\theta(s)$ and $P(a = 0) = 1 - f_\theta(s)$. To realize this into code, we use a fully connected neural network where the two outputs will be mapped to a softmax activation function giving the respective probabilities.

Since the neural network contains no hidden layers, we remove the mus and sigmas initialization from the bipedal-walker code that was given. However, we will change the "hidden" variable to a variable called "out" as this will be the output given by the neural network using a softmax activation. This is given in *cartpole.py lines 31-37* (see Appendix B for full *cartpole.py* code):

```
out = 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='fc')
```

The Gaussian distribution in the bipedal-walker environment is modified into a Bernoulli distribution, which takes the two softmax outputs of the neural network as the probabilities. To obtain the samples, the *tanh* function was omitted as there was no proper evidence of it boosting performance. However, if the training stage did not give satisfactory results, the *tanh* function would be tested. The code (*cartpole.py lines 41-42*) is given below and we note the rest of the code remains the same until we reach the for-loop.

```
pi = tf.contrib.distributions.Bernoulli(p=out, name='pi')
pi_sample = pi.sample()
```

We now enter the for-loop, noting that all variables initialized prior to the while loop were not changed. Within the while loop however, only one change was made; that is, the input parameter to the *env.step* method. The code is given below:

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

For the *env.step*() method, we note that the *action* variable in the cartpole case is a vector of [1,0], [0,1], [0,0] or [1,1]. We choose to take the first element of *action*, *action*[0] as the action to be performed. This choice is arbitrary (the second element could be taken), as long as it is consistent (the first element is always taken or the second element is always taken).
The overall scheme for the network is a network that takes in environment parameters (features) and outputs two values through fully connected softmax activation. The outputs are then sent through the Bernoulli distribution. A sample is retrieved for each of the outputs at each time-step for a given episode. We take only the first sample as our action. The other sample merely supports our network. By feeding in the environment parameters x, the samples y and G_t, we update the weights in our neural network using the same method as in Part 1.

Part 3

The weights are stored in a 4×2 2D-array and trained over 2000 episodes with $\alpha = 0.0001$ and $\gamma = 0.99$:

$$w = \begin{bmatrix} w_{00} & w_{01} \\ w_{10} & w_{11} \\ w_{20} & w_{21} \\ w_{30} & w_{31} \end{bmatrix}$$

Part a

The graphs shown in Figures 2 to 11 show the change of the eight weights, mean return (calculated over the prior 25 episodes) and mean number of time-steps (calculated over the prior 25 episodes). The results reach 50 mean time-steps (calculated over the prior 25 episodes) around episode 150 and continues to improve over time as shown in Figure 11.

Part b

We observe the following matrix multiplication which is constructed by the neural network of the cartpole containing only an input layer wherein the outputs are then mapped to a softmax activation function. This is given by:

$$\begin{bmatrix} x & \dot{x} & \theta & \dot{\theta} \end{bmatrix} \begin{bmatrix} w_{00} & w_{01} \\ w_{10} & w_{11} \\ w_{20} & w_{21} \\ w_{30} & w_{31} \end{bmatrix}$$

This will then further allow us to compute the two dimensional output. Call this variable "out" such that:

$$out^{(1)} = w_{00}x + w_{10}\dot{x} + w_{20}\theta + w_{30}\dot{\theta}$$
$$out^{(2)} = w_{01}x + w_{11}\dot{x} + w_{21}\theta + w_{31}\dot{\theta}$$

We furthermore note the inputs are the position, velocity, angular position, and angular velocity respectively. Now, observing our trend in the weights it is evident that if one of the weights increases for a certain state, then the other weight decreases with the other one. To give a complete picture, we will give the weights from the first initialization and then the final weights, after 2000 iterations:

$$0.22x + 0.93\dot{x} - 0.0139\theta + 1.313\dot{\theta} \rightarrow 0.46x + 1.21\dot{x} + 0.666\theta + 3.682\dot{\theta} \ (eq1)$$
$$1.16x - 0.0368\dot{x} + 0.232\theta + 0.297\dot{\theta} \rightarrow 0.92x - 0.37\dot{x} - 0.448\theta - 2.07\dot{\theta} \ (eq2)$$

Observing the right hand side of the stated equations above, we may now observe the weights as factors which control how much the agent prioritizes each state to deduce what action to take. We will concern ourselves first with $eq1$ as this is the equation associated with the selected action. If we have a positive weighted value for angular velocity, this indicates the rod is falling towards its chosen reference point. To prevent the rod from falling, the agent must now move towards the same direction, hence also being a positive weight. The rest of the states must then be also a positive value as the chosen reference directly follows from its respective velocities. This explains why all the weights are positive in the first equation. Now, the same can be said for the second equation. If we had trained for a longer duration, we would expect all the weights to become negative (as can be seen in the graphs). The reason for this is the same as the first equation. If the angular displacement of the rod is falling in its intended negative orientation, then the agent must move towards the negative direction as well.

The equation itself can be thought of as a proportional controller, in which the weights are converging to a point where many environment parameters satisfy effective control of the cartpole.

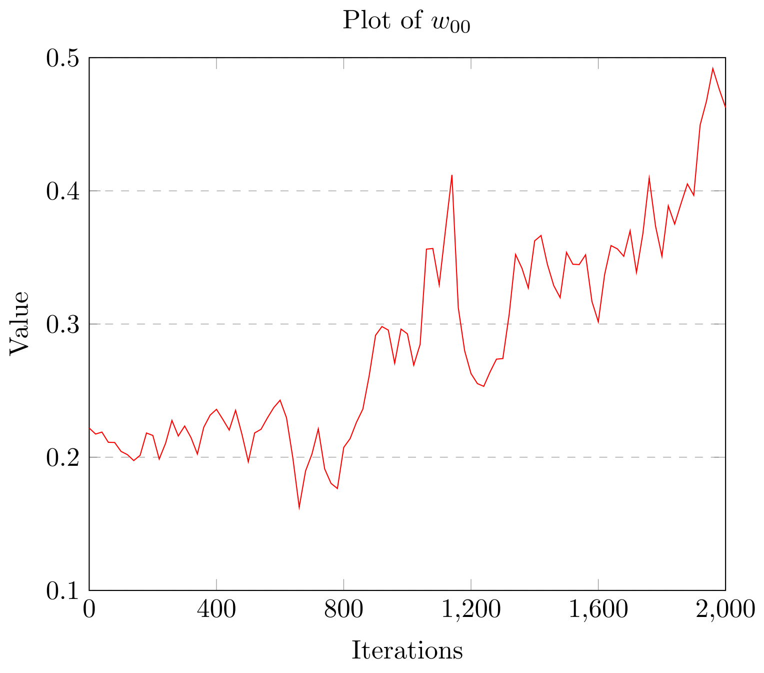


Figure 2

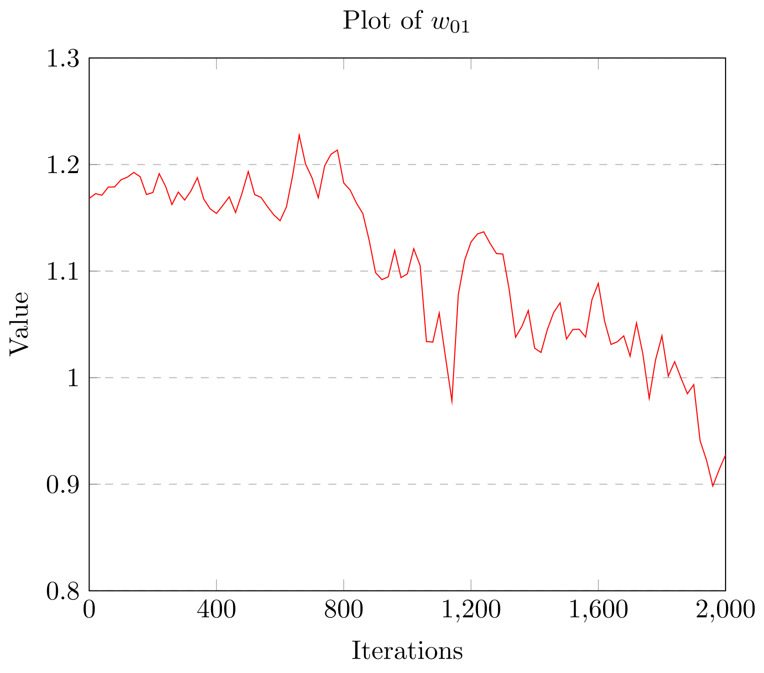


Figure 3

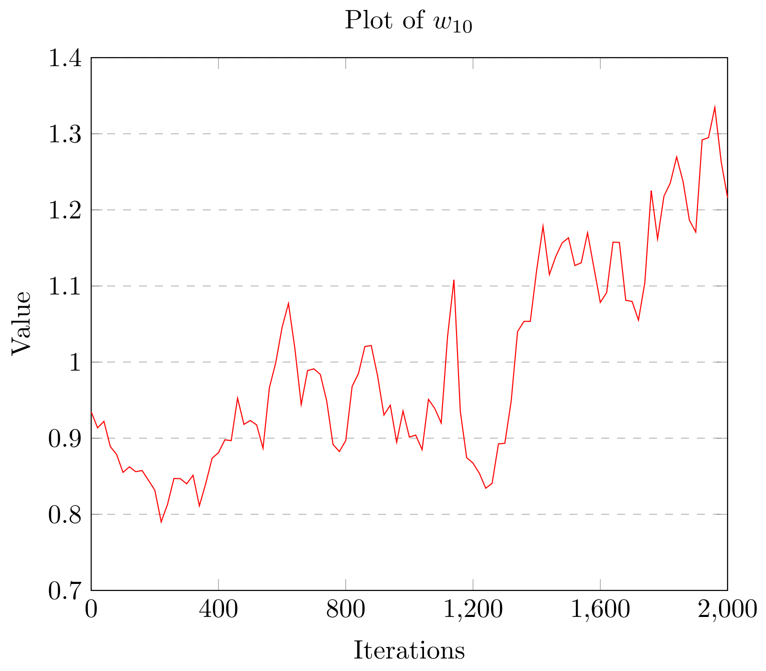


Figure 4

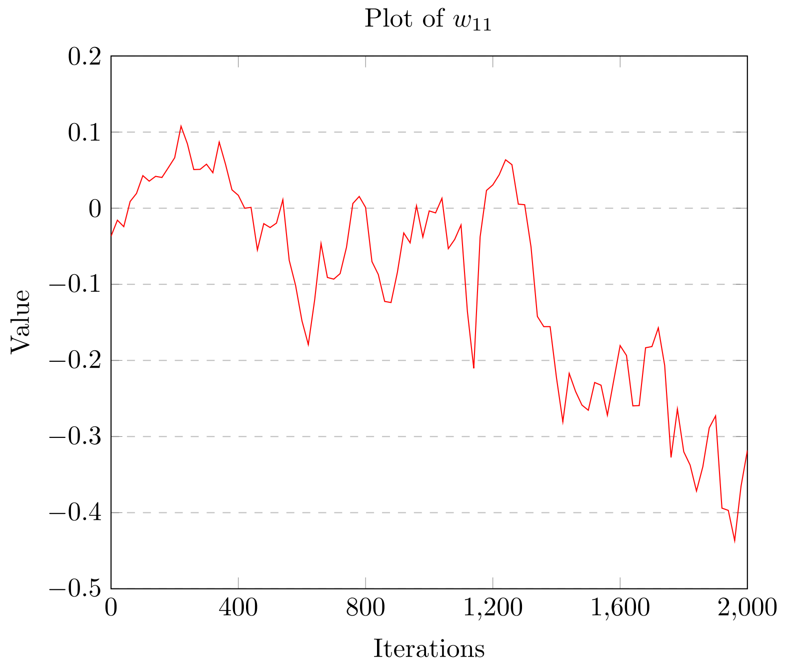


Figure 5

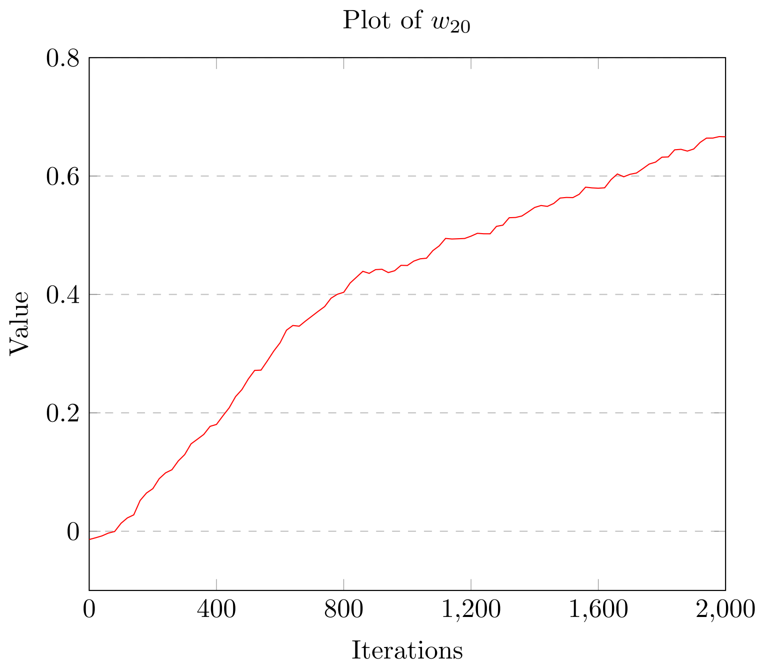


Figure 6

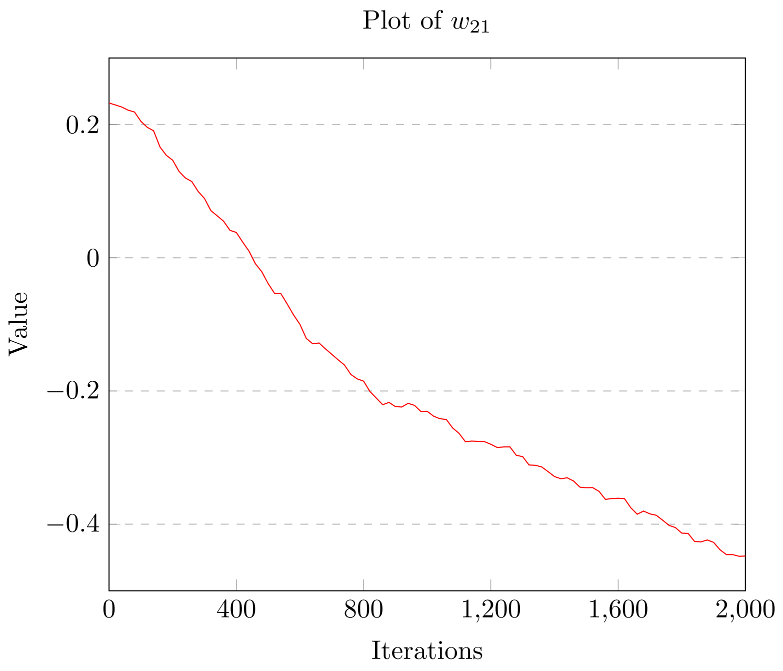


Figure 7

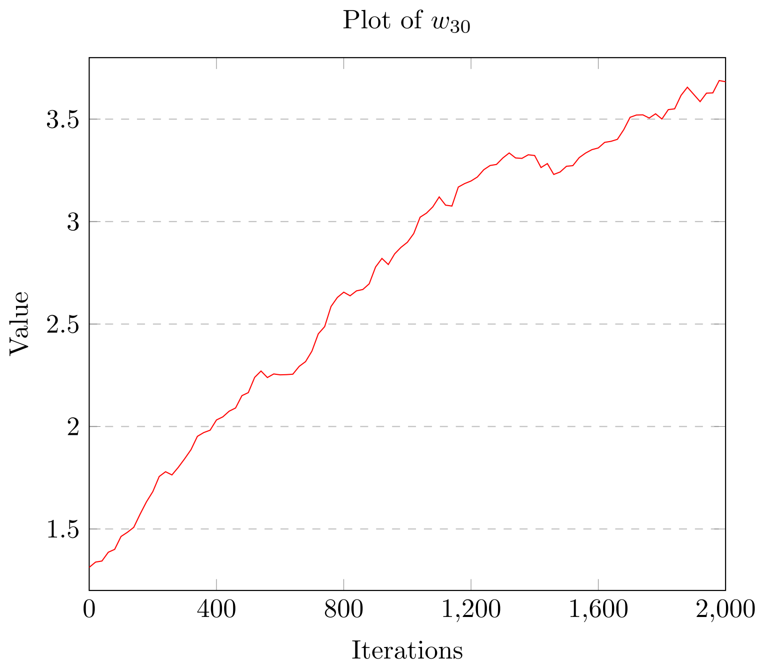


Figure 8

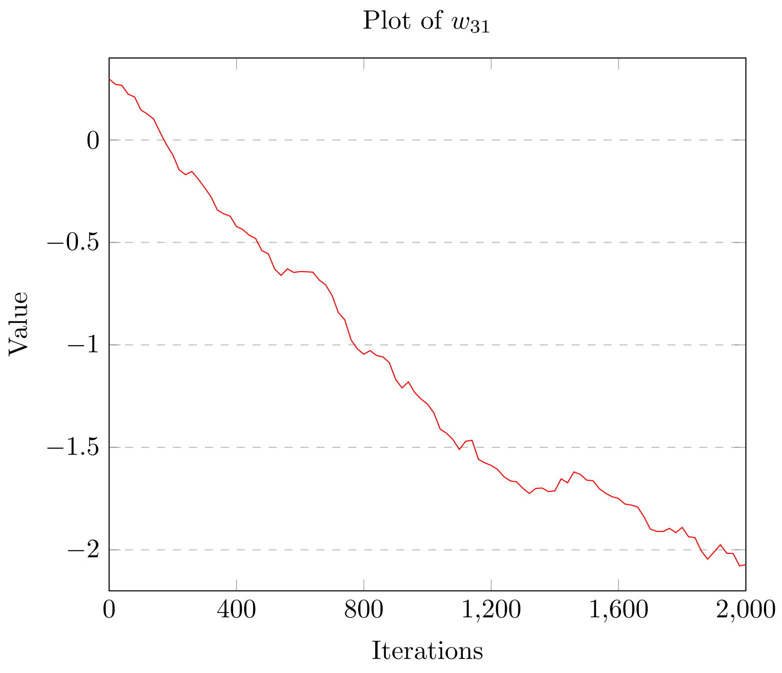


Figure 9

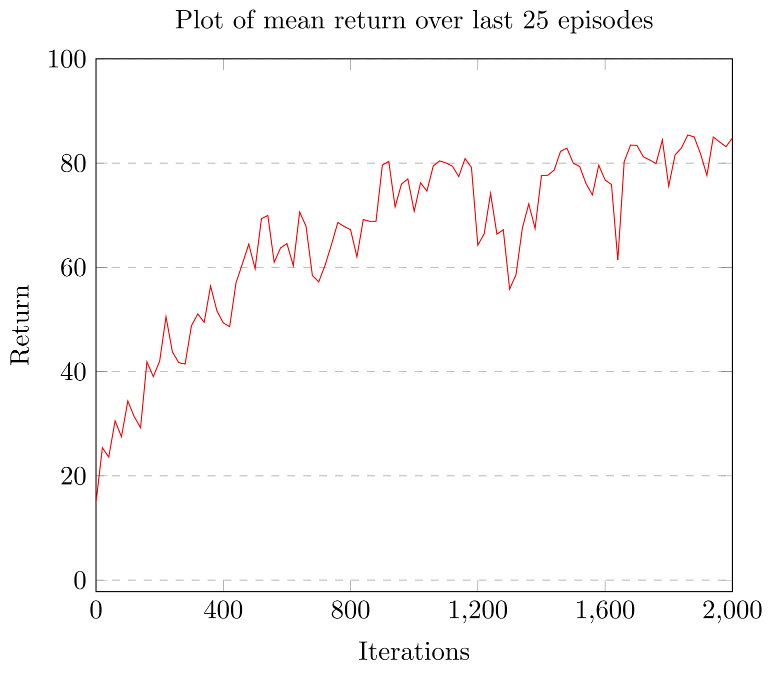


Figure 10

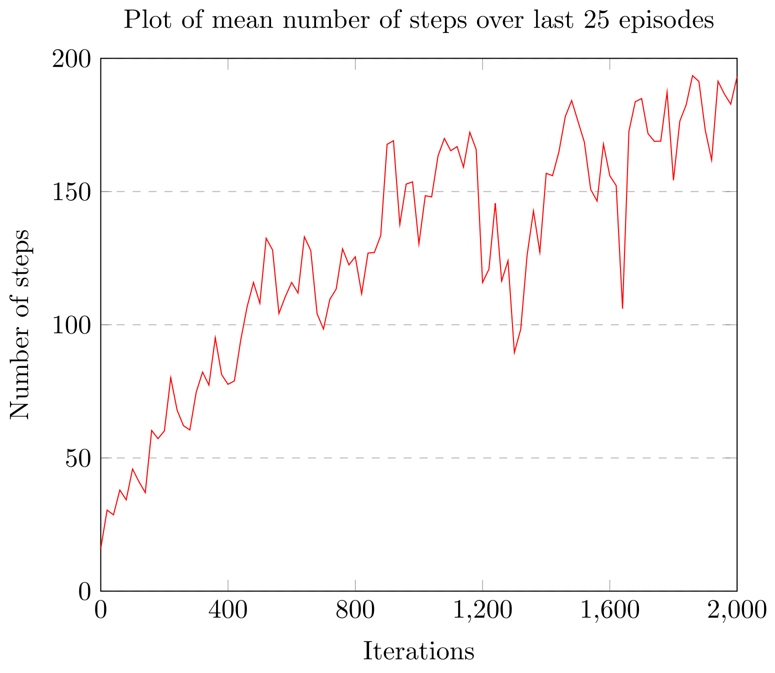


Figure 11

Appendix A

bipedal-reinforce.py (Python 3.5)

```
#!/usr/bin/env python3

from argparse import ArgumentDefaultsHelpFormatter, ArgumentParser

import gym
import numpy as np
import tensorflow as tf
from tensorflow.contrib.layers import *
import sys

parser = ArgumentParser(formatter_class=ArgumentDefaultsHelpFormatter)
parser.add_argument('-l', '--load-model', metavar='NPZ', help='NPZ file containing
    model weights/biases')
args = parser.parse_args()

env = gym.make('BipedalWalker-v2')

RNG_SEED = 1
tf.set_random_seed(RNG_SEED)
env.seed(RNG_SEED)

hidden_size = 64
alpha = 0.01
TINY = 1e-8
gamma = 0.98

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

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

```
input_shape = env.observation_space.shape[0]
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()

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)

sess = tf.Session()
sess.run(tf.global_variables_initializer())

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

track_returns = []
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})

    track_returns.append(G)
    track_returns = track_returns[-MEMORY:]
    mean_return = np.mean(track_returns)
    print("Episode {} finished after {} steps with return {}".format(ep, t, G))
    print("Mean return over the last {} episodes is {}".format(MEMORY, mean_return))

    with tf.variable_scope("mus", reuse=True):
        print("incoming weights for the mu's from the first hidden unit:", sess.run(tf
            .get_variable("weights"))[0, :])

sess.close()
```

Appendix B

cartpole.py (Python 3.5)

```
#!/usr/bin/env python3

import gym
import numpy as np
import tensorflow as tf
from tensorflow.contrib.layers import *
import sys

env = gym.make('CartPole-v0')

RNG_SEED = 1
tf.set_random_seed(RNG_SEED)
env.seed(RNG_SEED)

alpha = 0.0001
gamma = 0.99

w_init = xavier_initializer(uniform=False)
b_init = tf.constant_initializer(0.1)

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

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, output_units), name='y')

out = 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='fc')

all_vars = tf.global_variables()

pi = tf.contrib.distributions.Bernoulli(p=out, name='pi')
pi_sample = 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)

sess = tf.Session()
sess.run(tf.global_variables_initializer())
```

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


track_steps = []
track_returns = []

# For LaTeX plotting
w1_plot = ''
w2_plot = ''
w3_plot = ''
w4_plot = ''
w5_plot = ''
w6_plot = ''
w7_plot = ''
w8_plot = ''
returns_plot = ''
steps_plot = ''

for ep in range(2001):
    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[0])
        ep_rewards.append(reward * I)
        G += reward * I
        I *= gamma

        t += 1
        if t >= MAX_STEPS:
            break

    returns = np.array([G - np.cumsum(ep_rewards[:-1])]).T
    index = ep % MEMORY

    print(np.array(ep_states))

    _ = sess.run([train_op], feed_dict={x: np.array(ep_states),
                                        y: np.array(ep_actions),
                                        Returns: returns})
    track_steps.append(t)
    track_steps = track_steps[-MEMORY:]
    mean_steps = np.mean(track_steps)

    track_returns.append(G)
    track_returns = track_returns[-MEMORY:]
    mean_return = np.mean(track_returns)

    print("Episode {} finished after {} steps with return {}".format(ep, t, G))
    print("Mean return over the last {} episodes is {}".format(MEMORY, mean_return))
    print("Mean number of steps over the last {} episodes is {}".format(MEMORY,
        mean_steps))

    with tf.variable_scope('fc', reuse=True):
        weights = sess.run(tf.get_variable('weights'))
        print("Weights:")
        print(weights)

    if ep % 20 == 0:
        w1_plot += str((ep, weights[0, 0]))
        w2_plot += str((ep, weights[0, 1]))
        w3_plot += str((ep, weights[1, 0]))
        w4_plot += str((ep, weights[1, 1]))
        w5_plot += str((ep, weights[2, 0]))
        w6_plot += str((ep, weights[2, 1]))
        w7_plot += str((ep, weights[3, 0]))
        w8_plot += str((ep, weights[3, 1]))
        returns_plot += str((ep, mean_return))
        steps_plot += str((ep, mean_steps))

print('w1:', w1_plot)
print('w2:', w2_plot)
print('w3:', w3_plot)
print('w4:', w4_plot)
print('w5:', w5_plot)
print('w6:', w6_plot)
print('w7:', w7_plot)
print('w8:', w8_plot)
print('returns:', returns_plot)
print('steps:', steps_plot)

sess.close()
```