CSC411: Assignment # 4

Due on Tuesday, April 4, 2017

Ding, Si Chuang
1001165106

Shi, Yudian
999677862

Tuesday, April 4, 2017

1 Analysis of Bipedal Walker Algorithm

Note that REINFORCE uses the complete return from time t, which includes all future rewards up until the end of the episode. In this sense REINFORCE is a Monte Carlo algorithm and is well defined only for the episodic case with all updates made in retrospect after the episode is completed (like the Monte Carlo algorithms in Chapter 5). This is shown explicitly in the boxed pseudocode below.

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

The vector $\frac{\nabla_{\boldsymbol{\theta}}\pi(A_t|S_t,\boldsymbol{\theta})}{\pi(A_t|S_t,\boldsymbol{\theta})}$ in the REINFORCE update is the only place the policy parameterization appears in the algorithm. This vector has been given several names and notations in the literature; we will refer to it simply as the *eligibility vector*. The eligibility vector is often written in the compact form $\nabla_{\boldsymbol{\theta}} \log \pi(A_t|S_t,\boldsymbol{\theta})$, using the identity $\nabla \log x = \frac{\nabla x}{x}$. This form is used in all the boxed pseudocode in this chapter. In earlier examples in this chapter we considered exponential softmax policies (13.2) with linear action preferences (13.3). For this parameterization, the eligibility vector is

$$\nabla_{\boldsymbol{\theta}} \log \pi(a|s,\boldsymbol{\theta}) = \boldsymbol{\phi}(s,a) - \sum_b \pi(b|s,\boldsymbol{\theta})\boldsymbol{\phi}(s,b). \tag{13.7}$$

As a stochastic gradient method, REINFORCE has good theoretical convergence properties. By construction, the expected update over an episode is in the same direction as the performance gradient.[2] This assures an improvement in expected performance for sufficiently small α, and convergence to a local optimum under standard stochastic approximation conditions for decreasing α. However, as a Monte Carlo method REINFORCE may be of high variance and thus slow to learn.

We applied this algorithm to the gridworlds in Examples 13.1 and/or 13.2 to obtain the results presented earlier or here in this section.

Figure 1: p.271 of Sutton & Barto

The policy function π_θ is implemented with a single hidden layer. The actions for the bipedal walker is continuous, so we have to use Gaussian distribution on π. Here, the hidden layer is passed to μ and σ.

If there is no weight θ saved previously, it will be initialized. For hidden, μ and σ layers, there are corresponding w's and b's. The initialization codes shown below keep the gradients scale roughly the same.

```
# check whether weights and biases are saved,load them. If not, initialize them.
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


# activation function: ReLU
# 1 layer of fully connected hidden unit
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')
```

activation function :tanh
last layer of the neural network used as ϕ(a,s)(feature) $\mu = \phi\ (s,a)^T$

```
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')
```

the sigma value is clipped when it is too small or too big
activation function: softplus $g(x) = \ln(1 + e^x)$

```
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)
```

log probability of y given μ and σ. Lists of states, actions, and the returns will be passed into the training block (mention below in the for loop). Tensorflow will use the state and weights to generate new μ and σ and later compute the log probability of the actions based on the new μ and σ.

```
log_pi = pi.log_prob(y, name='log_pi')
```

returns 1 by (T-1) array for rewards (float). The cost function used here is $J(\theta) = -\sum[G_t log\pi(A_t|S_t, \theta)]$. gradient decent is used to optimize the θ in order to minimize the cost.

```
Returns = tf.placeholder(tf.float32, name='Returns')
optimizer = tf.train.GradientDescentOptimizer(alpha)
train_op = optimizer.minimize(-1.0 * Returns * log_pi)
```

Here is where training data starts. In each iteration, the environment will be reset. Then, states, actions and rewards are generated from time 0 to time T. Randomly sample actions from the π normal distribution. Use pi.sample() to generate the corresponding action *pi_sample* and use it to generate new action, state and reward. Lists of *ep_actions*,*ep_states*, *ep_rewards* are used to store all corresponding information. The total discounted rewards is stored in G. Then we obtain G_t, the program calls a cumulative sum function on *ep_reward* and minus it from G. starting from time t. Finally, returns store the total discounted rewards for each iterations from time 0 to T-1.

```
for ep in range(16384):
    # reset the environment
    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()
        # pi_sample() is a list store randomly generated probabilities
        # assign pi_sample to action
        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 is total discounted reward
        G += reward * I
        I *= gamma
        t += 1
        if t >= MAX_STEPS:
            break
    if not args.load_model:
        # returns = G_t = total discounted reward - cumulative up until time t
        returns = np.array([G - np.cumsum(ep_rewards[:-1])]).T
        index = ep % MEMORY
        _ = sess.run([train_op],
                     # ep_states includes all states from S_0 to S_{T-1}
                     # ep_actions includes all actions from A_0 to A_{T-1}
                     feed_dict={x:np.array(ep_states),
                                y:np.array(ep_actions),
                                # returns include all returns(G_t) from t = 0 until t
    = T
                                Returns:returns })
```

2 Part 2, Changes To Code

In part 1, we used multiple layers: hidden layer which computes the feature vector x. It is then been passed to the other layer which computes μ and the other computes σ. Based on the outputs we get to generate π and further use π found to compute action.

In part 2, it is the simplified version based on the method used in part 1. Instead of using multi-layers, we only used one layer here. We first create the environment 'CartPole-v0'. The activation function used is "sigmoid". An output array of size 2 is generated, of 0 and 1, sampling from a Bernoulli distribution with the probability determined by the observation and weights. Each of the two elements from the policy output represent weather the cart should go left or right. The cart will receive a input force to the left if the network sampled a 0 for moving right and a 1 for moving left. Otherwise the cart will receive a pump to the right. This is implemented as a argmax on the output vector.

2.1 Detailed Changes to Policy Related Code

This section lists the relevant changes to the example we required to make the policy network function. Some other less relevant code, such as plotting and book keeping, that are does not influence funtion of the policy network are omitted.

Changed the learning rate to 0.001 and gamma to 0.99, as per instructed in the assignment. We also found that these values did produce much better result than others.

```
# Parameters for single layer NN as policy network
hidden_size = 8
alpha = 0.001
TINY = 1e-8
gamma = 0.99
```

Changed input and output nodes to the correct dimension

```
# Tensorflow input and output initialization
# Input features: 4, output features 2
x = tf.placeholder(tf.float32, shape=(None, 4), name='x')
y = tf.placeholder(tf.float32, shape=(None, 2), name='y')
```

Modified the nerual net structure to only a single softmax layer, with sigmoid activation function. The assignemnt suggested that we use this structure for the policy network. The sigmoid activation function layer is equivlent to softmax.

```
# Only one lay network is required, softmax
sigmas = tf.clip_by_value(fully_connected(
    inputs=x,
    num_outputs=2,
    activation_fn=tf.nn.sigmoid,
    weights_initializer=sw_init,
    weights_regularizer=None,
    biases_initializer=sb_init,
    scope='sigmas'),
    TINY, 5)
```

Modified the probability distribution for pi. Since the action here is descrete the bernulie distribution is more approiate space to sample from. Sampling from this distribution will give us a

element output, representing if the network decide if we should push left, right, or undecided. In this case of this implimentation, the cart force act to the right by default if the sampled outputs are both 1 or 0.

```
# Layers of network
# Use bernulie distribution here instead since actions are descrete
pi = tf.contrib.distributions.Bernoulli(p = sigmas, name='pi')
pi_sample = pi.sample()
log_pi = pi.log_prob(y, name='log_pi')
```

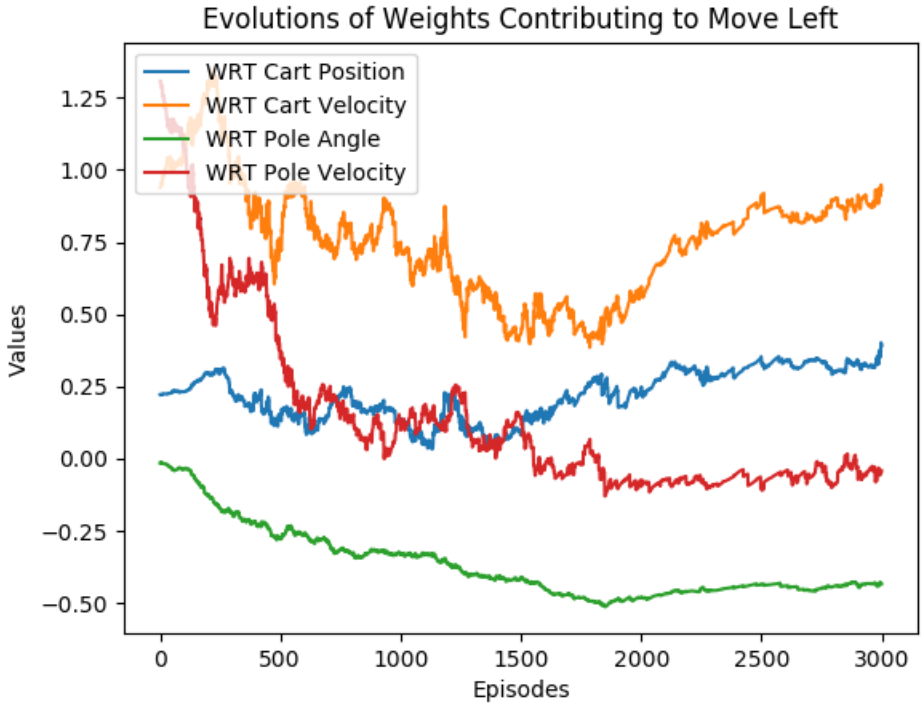
The while loop running the enviorment is modified to handle the previous changes made to the policy network.

```
    while not done:
        ep_states.append(obs)
        #env.render()
        action =  sess.run([pi_sample], feed_dict={x:[obs]})[0][0]

        obs, reward, done, info = env.step(np.argmax(list(action)))

        ep_actions.append(action)
        ep_rewards.append(reward * I)
        G += reward * I
        I *= gamma

        t += 1
        if t >= MAX_STEPS:
            break
```

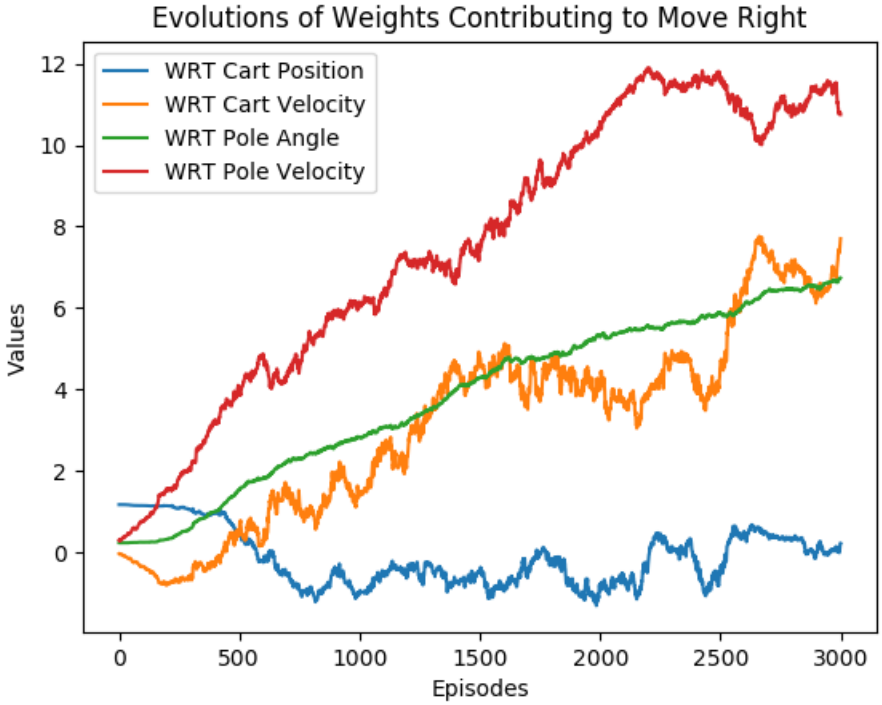
3 Training Cartpole

3.1 part 3a

Figure 2: Steps Survived and Returns Over Training Episodes



(a) Weights for Moving Left



(b) Weights for Moving Right

Figure 3: Evolution of Weights Over Episodes

3.2 Part 3b, Why Weights Work

By the nature of how cartpole is implemented, there is no "No Action" move. i.e. the cart have to neither move left or move right. Also, from a controls perspective, we know that since we only have the position and velocity information from cartpole, we would expect some kind of PD controller to be implemented by the policy network. Looking at the weights, it makes a huge amount of sense:

The weights for moving left is very small compared to the ones for moving right. This makes a lot of sense since the cart can only do one of two actions. If it is not moving right then it will automatically be moving left. So only one of the output nodes need to have large weights that are trained to the system and the other one can essentially be passive. In this case, the decision ended up being neither one of "move right" or "not move right".
Focusing on the weights toward direction the cart to move right, we noticed the properties that we expect out of a PD controller.

- The weight is largest with the pole velocity, the cart is most likely to move right if it system sees the pole is falling over toward the right. Weights for the pole velocity if about double compared to cart velocity and pole position. This makes sense because the cart need to move twice as much to influence the pole velocity as the center mass of the pole is half way up, accelerating the cart by a unit velocity will accelerate the pole by half that much.

- Also, the weights are positively correlated to pole position, so if the pole is leaning toward the right side of the cart, the cart must move to the right.

- Weights are also positively correlated to the care velocity, since if the cart is moving to the right with some velocity, the cart need to be accelerated more to correct a pole leading right.

- Finally, the most telling and intuitive weight is the weight corresponding to the cart position. Its obvious from the setup of the environment, that the absolute position of the cart have no effect on balancing the pole, as long as the cart do not run off the edge of its rail. We can clearly see that the policy network was able to infer this fact, and adjusted the weights corresponding to the car's position to essentially be zero.