CSC411: Project 4

Due on Friday, March 31, 2017

Katie Datsenko, Loora Zhuoran Li

April 3, 2017

Part 1

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 θ
Repeat forever:

Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \theta)$
For each step of the episode t = 0, . . . , T − 1:

$G_t \leftarrow$ return from step t
$\theta \leftarrow \theta + \alpha\gamma^t G_t \bigtriangledown\theta \log \pi(A_t|S_t,\theta)$

We would like to show the code in bipedal-reinforce.py corresponds with the pseudocode above.

1. **Input and initialization, policy learning**

   A policy function π takes in a current state **s**, and outputs the action **a** the agent should take. For stochastic policy, $\pi(a|s) = P(A_t = a\ |S_t = s)$
   In bipedal-reinforce.py, a single-hidden-layer neural network is used to implement the policy function.

   x – Input for this neural network

   a state vector of dimension 24 (24 features for one state **s**)

   y – Output for this neural network

   probability for taking each possible action given current state **s**, number of actions = number of output units

```
input_shape = env.observation_space.shape[0]
NUM_INPUT_FEATURES = 24 #the state vector
x = tf.placeholder(tf.float32, shape=(None, NUM_INPUT_FEATURES), name='x')
y = tf.placeholder(tf.float32, shape=(None, output_units), name='y')
```

   Referring to the first two lines of the pseudocode.

   Before implementing the network structure, the policy weights and bias terms are initialized randomly.

```
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
```

In terms of structure for the network,
— "hidden" is a fully connected layer, with observation features for states as inputs, hidden units as outputs, relu function as the activation function, weights initialized using xavier initializer/given constant, bias terms initialized using relu initializer/given constant.

In this case, the actions are continuous, thus it is reasonable to use Gaussian distribution for the actions, centered around$\phi(s,a)^T\theta$, which is a $\sim \mathrm{N}(\phi(s,a)^T\theta, \sigma^2)$
— "mus" is a fully connected layer, with hidden units as inputs, $\phi(s,a)^T\theta$ as outputs, tanh function as the activation function, weights initialized using xavier initializer/given constant, bias terms initialized using relu initializer/given constant.
— "sigma" is a fully connected layer, also with hidden units as inputs, standard deviation σ as outputs, softplus function as the activation function, weights initialized using xavier initializer/given constant, bias terms initialized using relu initializer/given constant.

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

#Gaussian distribution policy for the actions
# mus = phi(s, a)^T times theta
mus = fully_connected(
    inputs=hidden, #hidden layer
    num_outputs=output_units,
    activation_fn=tf.tanh,
    weights_initializer=mw_init,
    weights_regularizer=None,
    biases_initializer=mb_init,
    scope='mus')

#the sigmas of the gaussian distribution,
sigmas = tf.clip_by_value(fully_connected(
    inputs=hidden, #units of hidden layer
    num_outputs=output_units,
    activation_fn=tf.nn.softplus,
    weights_initializer=sw_init,
    weights_regularizer=None,
    biases_initializer=sb_init,
    scope='sigmas'),
    TINY, 5)
```

a $\sim$ N($\phi(s,a)^T\theta$, σ^2)

For a certain state, probabilities π are computed for continuous actions based on the state input x with 24 features. pi-sample is a randomly generated action based on probabilities in pi, this is called later in the loop.

```
all_vars = tf.global_variables()
pi = tf.contrib.distributions.Normal(mus, sigmas, name='pi')
pi_sample = tf.tanh(pi.sample(), name='pi_sample')
```

2. **Repeat Forever: loop for infinite number of episodes**

Reset the environment for a new episode, following the policy with current θ, generate all the states and actions S_0, A_0, R_1, . . . , S_{T-1}, A_{T-1}, R_T.

G is the total (discounted) reward, starting from time t=0

t is the timestep

I is the coefficient for reward at timestep t (this is a discounted reward system)

```
    # reset the environment
    obs = env.reset() #root state
    # following the policy with current theta
    # generate all the states and actions and rewards
    G = 0
    ep_states = []
    ep_actions = []
    ep_rewards = [0]
    done = False
    t = 0
    I = 1
```

Now we would like to analyze how G_t and θ are updated

If this episode has not finished yet, add observation vectors(x inputs) to the state list. Based on state observations from input x, pick a single random action at time step t. Retrieve information on observations, reward, whether episode finished or not and info for taking this action. Add discounted reward for taking this action **a** to the reward list. **Update total discounted reward G by adding the discounted reward at the current time step to its original value, G = G +** $reward_t$. Update the reward discount coefficient by multiplying γ to its original value. Update time step by adding 1. Repeat above steps until the average number of time steps per episode exceeds the constant value MAX STEP.

```
    while not done:
        ep_states.append(obs)
        env.render()
        # pick a single random action for time step t, based on state obs -- input x
        # pi_sample is a randomly generated action based on probablities in pi
        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) #R_t - reward for taking action a
        # G is the total discounted reward, starting from time t=0
        #R0 + gamma*R1 + gamma^2*R2 + ... + gamma^(MAX_STEPS - 1)*R(N-1)
```

```
        G += reward * I
        I *= gamma #rewards farther away in time count less (greedy approach)
        #gamma between 0 and 1
        t += 1
        if t >= MAX_STEPS:
            break
```

For each step of the episode, if not done, θ is also updated. This corresponds to the last line in the pseudocode, $\theta \leftarrow \theta + \alpha\gamma^t G_t \bigtriangledown\theta \log \pi(A_t|S_t,\theta)$
The codes in the next box show details of how the stochastic gradient is updated.

Take the log probability of the output π for each state in this current episode, y is the output of neural network ie. the probability for taking each possible action given current state **s**.
Initialize Returns in order to record all G_t from t = 1 to T
Multiply each reward value G_t with the log of probability of the action A_t taken at state S_t for time step t, then take summation over all values for each time step t within an episode.
The goal is to maximize expected returns (Returns × log(pi)), this represents rewards from policy with current θ. Comparing with the pseudocode, instead of doing an update for every time step within an episode, it computes the cumulative gradients for all time steps together (the gradient of a sum is equal to the sum of the gradients of the individual terms). Then, multiply the expected returns by -1, thus we need to find the minimum of this function(-1 × Returns × log(pi)). Apply gradient descent technique with Tensorflow to compute the gradient corresponding to the θ for updates.

```
# log of prob of actions from A_0 to A_T-1
log_pi = pi.log_prob(y, name='log_pi') #log prob(p16)

#Returns contains all the G_t's from t=1 to t=T
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())
```

If the model not loaded: for each index i of ep rewards, take a cumulative sum of entry at index i with all the entries before it. G_t is the total discounted reward starting from time step t. G_t = total rewards value - cumulative sum up towards time step t. Returns is a set of all G_t from t = 1 to T Run tensonflow Session, use gradient descent to maximize expected rewards thus update θ; record all the states from s_0 to s_{T-1}; recored all the actions from A_0 to A_{T-1}; record all the discounted returns G_t from t = 1 to T.

```
    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 })
```

Part 2

1. Environment Setup

   We added our code right after the class *CartPoleEnv*, and set up the environment with these two lines:

```
env = gym.make('CartPole-v0')
cartpole = CartPoleEnv()
```

   We added the line

```
np.random.seed(0)
```

   as part of our random seed settings because we use np.random further in the code (when generating a action via bernoulli sample).

2. Policy Function

   We removed the previous policy function model which was a single-hidden-layer network and replaced it with a single fully-connected softmax layer.

   We have variables x, *params* with dimensions $(T \times 4)$ and (4×2) respectively. Variable x represents the list of States S of the cartpole, for a single episode. Each individual state is encoded by 4 features (thus, NUM_INPUT_FEATURES is updated to 4), relevant to the internal state representation as dictated by the *CartPoleEnv* class. Variable *params* represents our theta parameters for the policy function, they are the weights of the fully connected single layer. In the Cart Pole task, only two actions are possible, applying a force pushing left, and applying a force pushing right. Thus our policy function, which takes a state as input and outputs the probability of an action, has two outputs: one for action left and one for action right. Our corresponding single layer network has two output units, which are passed through a softmax activation function, which will eventually learn to compute the "probabilities" of an action, through our policy reinforcement training algorithm. This justifies the (4×2) dimension of our theta parameter matrix *params*.

```
NUM_INPUT_FEATURES = 4 #the number of features of the state vector
x = tf.placeholder(tf.float32, shape=(None,NUM_INPUT_FEATURES), name='x') #state
#y - A_t (actions selected from A_0 to A_T-1 via bernoulli)
y = tf.placeholder(tf.int32, shape=(None), name='y')

params = tf.get_variable("thetas",[NUM_INPUT_FEATURES,2])
linear_layer = tf.matmul(x,params)
pi = tf.nn.softmax(linear_layer)
```

3. Stochastic gradient update

   We also compute each stochastic estimation of the total expected reward based on an episode of the policy in the following way: we take the log of the softmax probabilities computed for every state in the episode (stored in Variable *pi*). Variable y stores the list of actions (one action per time step in the episode), 0 is assigned as the left action, 1 as the right. We keep the probabilities of only the actions that actually occurred in the generated episode by converting y to a one-hot representation and doing elementwise multiplication (Variable *act_pi* with modifications to the line from the handout). Then we multiply each reward value G_t (stored in Returns) with the log of the probability of the action A_t taken at the state S_t for time step t and sum over all values for every time step t of an episode.

This corresponds to the pseudocode on the slides, with the exception that we are trying to do the updates for an episode all at once, versus for each time step individually in an iteration of the loop and combining updates as we iterate (the gradient of a sum is equal to the sum of the gradients of the individual terms). To turn our Reward Expectation Maximization problem into a minimization problem, we multiply the expected episode reward by -1. The Tensorflow gradient descent algorithm will take care of the computation of the gradient with respect to theta for this update.

```
# log of prob of actions A_0 and A_1 for every state for episode
log_pi = tf.log(pi) #2 prob values for every state in vector of time steps

#We make a one-hot vector of 0/1 actions stored in y
#with a one at the action we want to increase the probability of.

#1. log_pi - should be (T x 2), 2 action probabilities for every time step
#2. tf.one_hot(y, 2, axis=1) - every row corresponds to one time step T,
#columns are left & right probabilities - (T x 2)

#tf.multiply - Returns x * y elementwise, will set the prob of action != At to 0
#tf.reduce_sum - squeezes vector values across two rows into flat vector,
#act_pi is probability of action A_t for every time step t

act_pi = tf.reduce_sum(tf.multiply(log_pi, tf.one_hot(y, 2, axis=1)),
reduction_indices=[1])

#Returns contains all the G_t's from t=1 to t=T
Returns = tf.placeholder(tf.float32, name='Returns')
optimizer = tf.train.GradientDescentOptimizer(alpha)

vect = act_pi * Returns #elementwise multiplication
#for gradient update for entire episode we take the sum
#Multiply by -1 because we want to maximize JavV reinforcement reward by
#following policy theta
loss = -tf.reduce_sum(vect)
train_op = optimizer.minimize(loss)
```

4. Training Parameters

   We changed the values of α and γ to be 0.0001 and 0.99 respectively, and compute the average number of time-steps per episode, stopping the reinforcement algorithm when the average is at 50 or higher. MAX_STEPS (max limit of time steps possible in an episode) is given by the environment constant which is set at 200 by default.

```
alpha = 0.0001
gamma = 0.99
MAX_STEPS = env.spec.tags.get('wrapper_config.TimeLimit.max_episode_steps')
```

5. Sampling action A_t

   The last important change to the code is how we sample actions for each time step of an episode from our policy function with the current theta. For Bipedal reinforcement, the actions were continuous values; thus the probabilities of actions were modelled using a normal distribution, and a sample action for a state came from a Normal Distribution with a particular μ and σ (computed using a network

trained on state input). In the Cartpole problem, the actions are discrete; there are only two. Thus this corresponds to sampling actions from a Bernoulli distribution, where the probabilities of the left or right action come from the two softmax units.

```
ep_states.append(obs) #record the state
# pick a single random action for time step t, based on state obs
action_probs = sess.run(pi, feed_dict={x:[obs]})
#Get action sample using Bernoulli, probs already provided thanks to policy
if np.random.uniform(0,1) < action_probs[0][0]:
    action = 0
else:
    action = 1

ep_actions.append(action)
```

Part 3

(a)
printout:

(b)
According to the setup codes below,

The input vector has a dimension of 4. (x, x', θ, θ')
x – position of cart along the horizontal axis, (with x=0 being the middle point. x takes positive and negative values for the cart on different sides of the middle point)
x' – velocity of cart along the horizontal axis
θ – angle between the pole and its balanced position (the pole is considered balanced if it is perpendicular to the the horizontal axis)
θ**'** – angular velocity of the pole

Observing our output, we found that

```
def _step(self, action):
        assert self.action_space.contains(action), "%r (%s) invalid"%(action, type(action))
        state = self.state
        x, x_dot, theta, theta_dot = state
        force = self.force_mag if action==1 else -self.force_mag
        costheta = math.cos(theta)
        sintheta = math.sin(theta)
        temp = (force + self.polemass_length * theta_dot * theta_dot * sintheta) /
               self.total_mass
        thetaacc = (self.gravity * sintheta - costheta* temp) /(self.length *
                   (4.0/3.0 - self.masspole * costheta * costheta / self.total_mass))
        xacc  = temp - self.polemass_length * thetaacc * costheta / self.total_mass
        x  = x + self.tau * x_dot
        x_dot = x_dot + self.tau * xacc
        theta = theta + self.tau * theta_dot
        theta_dot = theta_dot + self.tau * thetaacc
        self.state = (x,x_dot,theta,theta_dot)
        done =  x < -self.x_threshold \
                or x > self.x_threshold \
                or theta < -self.theta_threshold_radians \
                or theta > self.theta_threshold_radians
        done = bool(done)

        return np.array(self.state), reward, done, {}
```