CSC411 project4

Xiaoxue Xing

April 6, 2017

1 Part1

For π, the code use normal distribution, because the action is continuous. We use the hidden layer to generate the μ and σ.
For the weight, we will first load the model. If no such file, we will initialize it using tf. constantinitializer.
The cost function is J(θ) = - $\sum$(Gtlogπ (At |(st,θ)). We use gradient descent to minimize cost function.
Then we will start training. Firstly we reset the environment. Then , we initialize G, the states, action and reward. Then we keep track of this. We use total - cumulative to get Gt, and epreward to get cumulative. Then we store the total reward. Then tensorflow use this value to generate the new σ and μ

```
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')
```

The code has a fully connected hidden layer with activation function ReLU.

```
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')
```

This activation is tanh. This is the μ in the normal distribution.

```
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)
```

The activation is softplus. This is the σ in the normal distribution. clip by value means it will be clipped if it is too large or small.

```
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')
```

π use the normal distribution. Then calculate the log probability.

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

Minimize cost function.

```
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)
```

Training code. $G+ = reward * I$ shows that G is the total discounted reward. $returns = np.array([G - np.cumsum(ep_rewards[: -1])]).T$ shows that Gt is total minus culmulative until t.ep_states has all the states, $ep_actions$ has all the actions.$returns$ has all the Gt.

2 Part2

This time we don't need to use hidden layer. The policy function simply need a fully-connected layer. So I delete the sigma and mu and change the hidden code and the output code to:

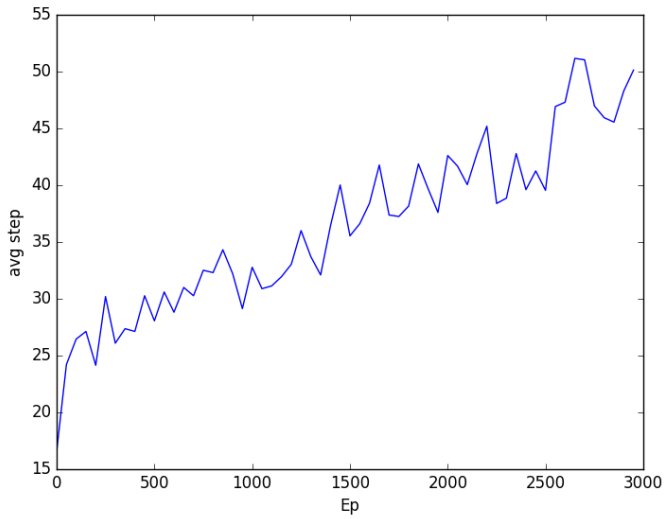
```
layer = fully_connected(
    inputs=x,
    num_outputs=output_units,
    activation_fn=None,
    weights_initializer=hw_init,
    weights_regularizer=None,
    biases_initializer=hb_init,
    scope='layer')
output = softmax(hidden, scope="output")
```

We set α as 0.000001 and γ 0.99. π instead of being normal distribution in part one, it is bernoulli distribution.

```
pi = tf.contrib.distributions.Bernoulli(p=output, name='pi')
pi_sample = pi.sample()
log_pi = pi.log_prob(y, name='log_pi')
```

Change π to bernoulli. Other are similar to part1 code.

3 Part3



And the printout that shows how the weights of the policy function changed (the 50 last eipsode) is :

The average step change is (too much, only put part of it):

Mean return over the last 50 episodes is 54.26
Episode 2450 finished after 20 steps with return 18.209306240276906
Mean return over the last 50 episodes is 60.44
Episode 2500 finished after 30 steps with return 26.029962661171943
Mean return over the last 50 episodes is 55.66
Episode 2550 finished after 37 steps with return 31.055091413092182
Mean return over the last 50 episodes is 71.18
Episode 2600 finished after 123 steps with return 70.95115056900356
Mean return over the last 50 episodes is 71.56
Episode 2650 finished after 40 steps with return 33.102824143031924
Mean return over the last 50 episodes is 80.96
Episode 2700 finished after 108 steps with return 66.22455991010975
Mean return over the last 50 episodes is 80.96
Episode 2750 finished after 121 steps with return 70.36134126007913
Mean return over the last 50 episodes is 71.6
Episode 2800 finished after 89 steps with return 59.117982557745
Mean return over the last 50 episodes is 69.28
Episode 2850 finished after 18 steps with return 16.548623854991234
Mean return over the last 50 episodes is 66.74
Episode 2900 finished after 62 steps with return 46.373177479281445
Mean return over the last 50 episodes is 73.12

Episode 2950 finished after 26 steps with return 22.99568541948446
Mean return over the last 50 episodes is 78.42

The weight change is (too much, only put the beginning part and end part):
array([[-0.75574315, 0.4998965], [-1.31953847, -0.39813197], [0.1250218 , 0.21801835], [-0.41616151, 0.00855926]], dtype=float32), array([[-0.75577581, 0.49992913], [-1.32076287, -0.3969076], [0.12464707, 0.21839309], [-0.4143247 , 0.00672247]], dtype=float32), array([[-0.75435668, 0.49850997], [-1.30136895, -0.41630155], [0.12403393, 0.21900623], [-0.43764147, 0.03003925]], dtype=float32), array([[-0.75461262, 0.49876592], [-1.2995472 , -0.41812333], [0.12380921, 0.21923093], [-0.44031 , 0.03270779]], dtype=float32), array([[-0.75438666, 0.49853998], [-1.30219722, -0.41547328], [0.12394034, 0.2190998], [-0.43626705, 0.02866484]], dtype=float32), array([[-0.75175202, 0.49590537], [-1.30162263, -0.41604787], [0.12239845, 0.22064169], [-0.43845141, 0.0308492]], dtype=float32), array([[-0.75094521, 0.49509859], [-1.30086184, -0.41680861], [0.12172437, 0.22131577], [-0.4389466 , 0.03134438]], dtype=float32), array([[-0.75017786, 0.49433121], [-1.29782641, -0.41984397], [0.12214455, 0.22089559], [-0.44212371, 0.03452148]], dtype=float32), array([[-0.7499904 , 0.49414375], [-1.29230511, -0.4253653], [0.1218283 , 0.22121184], [-0.44994566, 0.04234343]], dtype=float32), array([[-0.74996156, 0.49411491], [-1.29488635, -0.42278403], [0.1218445 , 0.22119564], [-0.4460367 , 0.03843446]], dtype=float32), array([[-0.74940145, 0.49355483], [-1.30132222, -0.41634819], [0.1216594 , 0.22138074], [-0.43642867, 0.02882642]], dtype=float32), array([[-0.74918139, 0.49333477], [-1.3168838 , -0.40078664], [0.12219573, 0.2208444], [-0.4098891 , 0.00228686]], dtype=float32), array([[-0.74970812, 0.4938615], [-1.32291007, -0.3947604], [0.12281796, 0.22022218], [-0.3998045 , -0.00779773]], dtype=float32), array([[-0.74928337, 0.49343678], [-1.31480181, -0.40286863], [0.12201135, 0.22102879], [-0.41140008, 0.00379785]], dtype=float32),

The weight at the end part:
array([[-0.80824262, 0.55239618], [-1.59070671, -0.12696318], [0.43482947, -0.0917893], [0.86067581, -1.2682786]], dtype=float32), array([[-0.8089053 , 0.55305886], [-1.58722591, -0.13044393], [0.43583861, -0.09279845], [0.85633886, -1.26394165]], dtype=float32), array([[-0.80937845, 0.553532], [-1.59501088, -0.12265899], [0.43551219, -0.09247203], [0.8656708 , -1.27327359]], dtype=float32), array([[-0.80893731, 0.55309087], [-1.59161341, -0.12605646], [0.43503469, -0.09199455], [0.85843498, -1.26603782]], dtype=float32), array([[-0.8123039 , 0.55645746], [-1.5986954 , -0.11897445], [0.43415251, -0.09111236], [0.86877471, -1.27637756]], dtype=float32), array([[-0.81357592, 0.55772948], [-1.59990501, -0.11776485], [0.43330622, -0.09026606], [0.87258184, -1.28018463]], dtype=float32), array([[-0.81339258, 0.55754614], [-1.59908843, -0.11858141], [0.43286633, -0.08982619], [0.86942059, -1.27702343]], dtype=float32), array([[-0.81501144, 0.559165], [-1.6022948 , -0.11537497], [0.43454793, -0.0915078], [0.87482452, -1.28242731]], dtype=float32), array([[-0.813977 , 0.55813056], [-1.60154021, -0.11612953], [0.43310541, -0.09006527], [0.87163436, -1.27923715]], dtype=float32), array([[-0.81384301, 0.55799657], [-1.61203408, -0.1056356], [0.43462506, -0.09158491],

[0.89068633, -1.29828918]], dtype=float32), array([[-0.81374246, 0.55789602], [-1.61779797, -0.0998717], [0.43540648, -0.09236634], [0.90061784, -1.30822074]], dtype=float32),

4 Part3b

The input vector is a 4 dimension vector with its location, velocity, angel and angular velocity. ([x,xdot,theta,thetadot]) The weight will balance these four elements. So the weights are always has a different sign. If the car move to the left, the velocity is negative. xdot*weight for xdotleft will be negative and xdot*weight for xdotright will bw positive. Therefore, the cart to push to right. Similiar for the location, angular velocity and angel. The system want to keep these elements near 0, and therefore keep balance. As we can wee, the weight first didn't follws one negative and one positive at begining. However, after training, the weight gradually becomes one negative and one positive. The reason, there are still one pair is both negative is because we didn't tarin enough. We can see in this pair one negative number is decreasing in absolute value, one negative is increasing in absolute value. If we continuous to train it, it will also become one positive and one negative.