CSC411: Project 4

Due on Friday, March 31, 2017

Katie Datsenko, Loora Zhuoran Li

April 1, 2017

Part 1

...

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

List the 10 words that most strongly predict that the review is positive, and the 10 words that most strongly predict that the review is negative. State how you obtained those in terms of the the conditional probabilities used in the Naive Bayes algorithm.
For each word a_i in the training set of reviews we store the log of the conditional probability of a word with respect to the class

$$\log(P(a_i|class)) + \log(P(class))$$

in a dictionary named *wordDict* stored under the respective class (i.e. either $wordDict[word][0]$ for positive or $wordDict[word][1]$ for negative). This allows us to easily compute the bayes classification probability for a sample review consisting of multiple words by taking the sum of the log of conditional probabilities, or the sum of the corresponding entries for each word a_i in the *wordDict*.
To obtain the words that strongly predict one of the classes, we used the log odds ratio of the conditional probabilities of a_i between the positive and negative classes. For example, the ratio of the positive to the negative class may be expressed as:

$$\log(\frac{P(class = 1|a_i)}{P(class = 0|a_i)})$$

$$= \log\left(\frac{\left(\frac{P(a_i=1|class=1)P(class=1)}{P(a_i)}\right)}{\left(\frac{P(a_i=1|class=0)P(class=0)}{P(a_i)}\right)}\right)$$

$$= \log(P(a_i|class = 1)) - \log(P(a_i|class = 0)) + log(\frac{P(class = 1)}{P(class = 0)})$$

(The $log(\frac{P(class=1)}{P(class=0)})$ term may be excluded since it is the same for every word a_i):

$$\approx \log(P(a_i|class = 1)) - \log(P(a_i|class = 0))$$

The reasoning behind the log odds ratio is that it will be greater than one for features (keywords) that cause belief in the Positive Class to be greater relative to the Negative Class. The features that have the greatest impact at classification time are those with both a high probability (because they appear often in the data) and a high odds ratio (because they strongly bias one label versus another). Computing the log odds requires a simple subtraction between the *wordDict* entries, as the log function is already applied on them.

```
================== RUNNING PART 3 ===================
10 most strongly Positive word predictions:
['bold', 'spielbergs', 'ideals', 'lovingly', 'gattaca', 'wonderfully',
'dread', 'fashioned', 'astounding', 'outstanding']
10 most strongly Negative word predictions:
['turkey', 'ludicrous', 'lifeless', 'feeble', 'unimaginative', 'stupidity',
'sucks', 'nonsense', 'insipid', 'insulting']
```

The relevant code snippet is shown below:

```
def part3(wordDict):
    logOdds = []

    for word in wordDict:
        logOdds.append((wordDict[word][0] - wordDict[word][1], word))
    logOdds.sort()
    vocab_size = len(logOdds)
    print("10 most strongly Positive word predictions:")
    print([word for val, word in logOdds[(vocab_size-10):]])
    print("10 most strongly Negative word predictions:")
    print([word for val, word in logOdds[:10]])
```