REINFORCEMENT LEARNING (RL) – POLICY GRADIENTS II

Edit

In the previous **post**, we discussed the basics of policy gradients for reinforcement learning tasks. In our multi-armed bandit implementation, the reward was immediate in terms of wether the action we took was good or bad. But it many RL tasks, the reward may be delayed where we won't know the precise impact of our action until the very end.

In this post, we will implement the classic cartpole RL task where we must balance a pole on a car for as long as possible. We will specifically be using the openAI gym in order to have a reactive environment that gives us observations (state) and reward with a given action from our model. I encourage you to check out the short OpenAI gym **tutorial** to cover the basics of the different game environments.

RANDOM PLAY

First, we will run the task using random movements. The cart can either move left or right in order to balance the pole. We will randomly choose which direction to move the cart and see the performance.

With the random movements, the car is not able to balance the poll for long and we receive (on avg) around 15 points (1 point for each time frame the pole was balanced). We need to develop a model that can determine how to move the car to balance the pole depending on the observation (state) and reward.

```
# random action sampled from action space
action = env.action_space.sample()
observation, reward, done, info = env.step(action)
total_reward += reward
num_time_steps += 1
```

AGENT

Our agent will be taking in observations, actions and rewards in order to train for the cartpole task. All three of the inputs have the same number of values because they are recorded after each action. The observation is the previous state of the environment, which was used to make the action and determine the reward for the action.

```
# Placeholders
self.observations = tf.placeholder(name="observations",
    shape=[None, FLAGS.dim_observation], dtype=tf.float32)
self.actions = tf.placeholder(name="actions",
    shape=[None, 1], dtype=tf.float32)
self.rewards = tf.placeholder(name="rewards",
    shape=[None, 1], dtype=tf.float32)

# Net
with tf.variable_scope('net'):
    W1 = tf.get_variable(name='W1',
        shape=[FLAGS.dim_observation, FLAGS.num_hidden_uni
    W2 = tf.get_variable(name='W2',
        shape=[FLAGS.num_hidden_units, 1])
    z1 = tf.matmul(self.observations, W1)
    fc1 = tf.nn.relu(z1)
    z2 = tf.matmul(fc1, W2)
    self.fc2 = tf.nn.sigmoid(z2)

# Loss
self.loss = - tf.reduce_mean((self.actions*tf.log(self.fc2
    (1-self.actions)*(tf.log(1 - self.fc2)))*self.rewards,

# Optimizing
self.train_optimizer = tf.train.AdamOptimizer(
    learning rate=FLAGS.learning rate)
```

- SEARCH -

Search …

- FOLLOW ME ON TWITTER -

Tweets by @GokuMohandas

Goku Mohandas
@GokuMohandas

Your model may be performing really well by incorrectly focusing on confounding features (extraneous influencers in the data that aren't accounted for). Check out @johnrzech's post where x-ray stickers unintentionally influenced the classifications: medium.com/@jrzech/what-a…

Embed View on Twitter

- RECENT POSTS -

Update on Embeddings (Spring 2017)

Exploring Sparsity in Recurrent Neural Networks

Question Answering from Unstructured Text by Retrieval and Comprehension

Overcoming Catastrophic Forgetting in Neural Networks

Opening the Black Box of Deep Neural Networks via Information

```
self.train_step = self.train_optimizer.minimize(self.loss)
```

We have a simple 2-layered MLP that takes in the observations and gives us 1 output per observation. This output is then used to determine which action to take. We feed this action into the environment and it is executed. We receive the state of the environment after the action was taken, the action itself, the reward from the action and wether the game has ended or not. We keep collecting obersvations, actions and rewards until the game is over and this constitues one peisode. We feed this entire episode's data into the train optimizer.

Our loss function is in the form of a simple sigmoid cross entropy (two classes) and is weighted by the sign and magnitude of the episode's rewards.

TRAINING

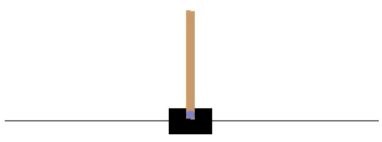
With training, we will not be adjusting our weights until the current episode is over. Until then, we perform actions using the previous observation.

```
states.append(observation)
fc2 = sess.run(model.fc2, feed_dict={
    model.observations: np.reshape(observation,
        [1, FLAGS.dim_observation])})

# Determine the action
probability = fc2[0][0]
action = int(np.random.choice(2, 1,
    p = [1-probability, probability]))

observation, reward, done, info = env.step(action)
actions.append(action)
rewards.append(reward)
```

We feed in the observation to get our probability, which we use to determine our action. We then perform this action to get our observation (state) after performing the action and the associated reward. We store these so we can use later for shifting our weights.

```
# Episode is finished (task failed)
if done:
    epr = np.vstack(
        discount_rewards(rewards, FLAGS.discount_factor))
    eps = np.vstack(states)
    epl = np.vstack(actions)

    epr -= np.mean(epr)
    epr /= np.std(epr)
    sess.run(model.train_step,
        feed_dict = {model.observations: eps,
        model.actions: epl, model.rewards: epr})

    accum_rewards[:-1] = accum_rewards[1:]
    accum_rewards[-1] = np.sum(rewards)

    # average reward for last 100 steps
    print('Running average steps:',
        np.mean(accum_rewards[accum_rewards &amp;gt; 0]),
        'Episode:', episode+1)

    break
```

Once the episode is finished, we can feed in all of our observations, actions and rewards to train our model. We are feeding in the entire episode's history and then using the reward to calcualte the loss. This means that if the reward is positive, then ALL the moves made in this episode will be favored by that magnitude. When we repeat this for enough episodes, our weights have been adjusted such a way that more and more episodes have the right actions given the observations. Eventually, we learn how to balance the pole decently well!



NUANCES

You may notice that we do an additional operation to our rewards before feeding it in for training. We do what's known as discounting the reward. The idea is that each reward will be weighted by all the rewards that follow it since the action responsible for the current reward will determine the rewards for the subsequent events. We weight each reward by the discount factor gamma^(time since reward). So each reward will be recalculated by the following expression:

$$r_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$

```
def discount_rewards(rewards, gamma):
    """
    Return discounted rewards weighed by gamma.
    Each reward will be replaced with a weight reward that
    involves itself and all the other rewards occuring aft
    The later the reward after it happens, the less effect
    has on the current rewards's discounted reward since g

    [r0, r1, r2, ..., r_N] will look someting like:
    [(r0 + r1*gamma^1 + ... r_N*gamma^N), (r1 + r2*gamma^1
    """
    return np.array([sum([gamma**t*r for t, r in enumerate
        for i in range(len(rewards))])
```

You may also notice that we do one an additional operation to the discounted rewards as well. Once we determine the discounted rewards, we standardize all the rewards to zero mean and unit variance. This is so we can control the scale at which we affect our loss and thereby, also control the gradients that back propagate to change our weights. This is a very nice and simple normalization strategy for our training. You can find out more about more normalizing techniques in my this **post**.

```
epr -= np.mean(epr)
epr /= np.std(epr)
```

EXTENSIONS

There are many more avenues to explore if you want to go deeper into reinforcement learning but this should provide a nice foundation for using policy gradients with RL tasks. We can take a look at using convolutional inputs from environements for more complicated tasks such as pong (**Karpathy**, **OpenAI**) or for the game of **GO**. All of these techniques are quite similar to our simple implementation here. For example, in Pong, we just use the game's images with the paddles and ball as the input observation. We apply convolution on the image (or on the difference between two images) and use it to determine an action. RL tasks like the Alpha GO one require pairing with more complicated tasks such as search tress, etc.

Another extension we could employ is the removal of the actual environment all together. If you think about it, even in our cartpole example, the environment is the bottle neck. We have to wait for the environments observations after each of our actions. We can speed up training but creating the environment ourselves! All we need to do is train a neural network that takes in previous observations, actions and rewards and predicts the outcome observations, actions and rewards. We initially have to use the actual environment and compare with it for loss but eventually we can train a net that closely models the actual environment. With this, we can quickly receive outputs for our inputs while training the RL model.

CODE

GitHub Repo (Updating all repos, will be back up soon!)

Posted in: *Reinforcement Learning*
Tagged: *Tutorials*

← *Reinforcement Learning (RL) – Policy Gradients I* *Convolutional Text Classification* →

ONE THOUGHT ON "REINFORCEMENT LEARNING (RL) – POLICY GRADIENTS II"

MAGESWARAN1989 August 30, 2017 at 4:41 pm EDIT REPLY →

hey…thanks for the post! could share the git link? above is not working.

★ Like

LEAVE A REPLY

Enter your comment here...