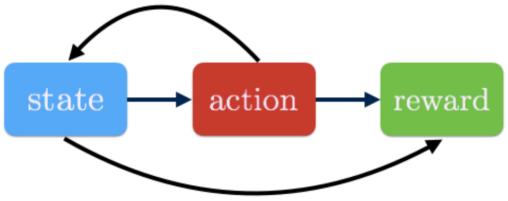
REINFORCEMENT LEARNING (RL) – POLICY GRADIENTS I

Edit

This is the first of a series of posts covering the basics of reinforcement learning (RL). There are several subsections under RL such as policy gradients, Q-learning, etc. In this post, we will specifically cover policy gradients, as it has many advantages over value and model based learning. But the reason I prefer it is due to that fact that it is all about maximizing future rewards and is end-to-end. We will first walk through a bit of the basic notation and then dive into our multi armed bandit example. Later, we will play around with OpenAI's RL gym and work on some extending our RL models.

OBJECTIVE

The three main components in our model include the state, action and reward. The state can be thought of the environment which generates an action which leads to a reward. Actions can also alter the state and often the reward may be delayed and not always an immediate response to a given action.



Let's take a took at our main objective function that we wish to minimize in order to train our policy weights, so that we choose the action that results in maximized total reward.

s	state
a	action
a^*	correct action
r	reward
π	policy
θ	policy weights
R	total reward
$\hat{A}$	advantage est.
γ	discount factor

$$\max_\theta \sum_{n=1}^{N} logP(y_n|x_n;\theta)$$

$$= \max_\theta \sum_{n=1}^{N} logP(a_n^*|s_n;\theta)$$

$$= \min_\theta [-\sum_{n=1}^{N} logP(a_n^*|s_n;\theta)]$$

But, we do not know the correct actions a^*

Since we don't have the correction actions to take, the best we can do it try some actions that may turn our good/bad and then eventually train our policy weights (theta) so that we increase the chances of the good actions. One common approach is to collect a series of states, actions and the corresponding rewards (s0, a0, r0, s1, a1, r1, …) and from this we can calculate R, the total reward – sum of all the rewards r. This will give us the policy gradient:

$$\frac{\partial J}{\partial \theta} = \frac{\partial \sum log\pi(a|s;\theta)}{\partial \theta} R$$

We will take a closer look at why this policy gradient is expressed as it is later in the section where we draw parallels with supervised learning.

When calculating the total reward R for an episode (series of events), we will have good and bad actions. But, according to our policy gradient, we will be updating the weights to favor ALL the actions in a given episode if the reward is positive and the magnitude of the update depends on the magnitude of the gradient. When we repeat this with enough episodes, our policy gradient becomes quite precise in modeling what actions to take given a state in order to maximize the reward.

There are several additions we can make to our policy gradient, such as adding on an advantage estimator to determine which specific actions were good/bad instead of just using the total

reward to judge an episode as good/bad. But we will cover these ideas and many others in future RL posts.

PARALLELS BETWEEN REINFORCEMENT LEARNING AND SUPERVISED LEARNING:

I highly reading the first section of my post on **gradients/normalization** where I cover some intuitive principles behind gradients. But I'll reiterate some of the overarching principles here and then discuss the parallels between the RL and a general supervised learning objective.

The gradient is another way of representing the partial derivative of some objective with respect to some parameter. The derivative tells us the direction to move (update) our weights in order to maximize the objective. In a typical supervised learning task, our objective may be some loss function, such as the multi-class cross-entropy loss. We can map this function and determine the gradients with respect to the weights. The gradient tells us how to update the weights in order to maximize the loss, but we want to minimize the loss, so we simply subtract the gradient. Recall that we use alpha to control the effect of the gradient.

Let's walk through a quick example to see what this looks like:

$$J(\theta) = -\sum_i ln(\hat{y}_i)$$

Negative Log Likelihood

$$\frac{\partial J}{\partial W_j} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial W_j} = -\frac{1}{y}\frac{\partial y}{\partial W_j} = -\frac{1}{\frac{e^{W_y X}}{\sum e^{XW}}}\frac{\sum e^{XW} e^{W_y X} 0 - e^{W_y X} e^{W_j X} X}{(\sum e^{XW})^2} = \frac{X e^{W_j X}}{\sum e^{XW}} = XP$$

$$\frac{\partial J}{\partial W_y} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial W_y} = -\frac{1}{y}\frac{\partial y}{\partial W_y} = -\frac{1}{\frac{e^{W_y X}}{\sum e^{XW}}}\frac{\sum e^{XW} e^{W_y X} X - e^{W_y X} e^{W_y X} X}{(\sum e^{XW})^2} = \frac{1}{P}(XP - XP^2) = X(P-1)$$

$$W_i = W_i - \alpha\frac{\partial J}{\partial W_i}$$

Above is the negative log likelihood loss function. Loss (j) is the sum of y_pred which is only the loss for the correct class. So, if our weights are good, we will get a high value for y_pred and so our loss (-ln(y_pred)) will be close to 0. However, if our weights are poor, our loss will be high. Though our loss does not account for the y_pred probabilities for the incorrect classes, they do come into play when determining the gradients, which we will use to update all of our weights. The gradient for the correct class is X(P-1) and the gradients for the incorrect classes are XP. (**Note:** X because this is a simple 2 layer MLP, otherwise, you'd see the last layer's derivative with respect to W_j instead)). This makes sense because if P is high for the correct class (which we want), X(P-1) will be a small negative number because P is always

So how does this supervised learning technique relate to reinforcement learning policy gradients? If you inspect the loss functions, you will see that they are exactly the same in principle. The negative log likelihood loss function above is the simplification of the multi-class cross entropy loss function below.

$$J(\theta) = -\sum_i y_i ln(\hat{y}_i)$$

Ex:

Computed ($\hat{y}$)	Targets (y)
[0.3, 0.3, 0.4]	[0, 0, 1]

$$J(\theta) = -\sum_i^n [0 * ln(0.3) + 0 * ln(0.3) + 1 * ln(0.4)] = -ln(0.4)$$

With the multinomial cross entropy, you can see that we only keep the loss contribution from the correct class. Usually, with neural nets, this will be case if our ouputs are sparse (just 1 true class). Therefore, we can rewrite our loss into just a sum(-log(y_hat)) where y_hat will just be the probability of the correct class. We just replace y_i (true y) with 1 and for the probabilities for the other classes, doesn't matter because their y_i is 0. This is referred to as negative log likelihood. But for drawing the parallel between supervised learning and RL, let's keep it in the explicit cross-entropy form.

	supervised	reinforcement
J	$-\sum y_i log(\hat{y}_i)$	$-\sum r log \pi(a\|s;\theta)$

In supervised learning, we have a prediction (y_hat) and a true label (y_i). In a typical case, only one label will be 1 (true) in y_i, so therefore only the log of the true class's prediction will be

taken into account. But as we saw above, the gradient will take into account the predicted probability for all the classes. In RL, we have our action (a) based on our policy (pi) which we take the log of. We multiply the action from the policy with our reward for that action. **Note:** The action is a number from the outputs (ex. chosen action is 2, so we take the 0.9 from [0.2, 0.3, 0.9, 0.1] to put into the log . he reward can be any magnitude and direction but like our supervised case, it will help determine the loss and properly adjust the weights by influencing the gradient. If the reward is positive, the weights will be altered via the gradient in order to favor the action that was made in that state. If the reward is negative, the gradient will be unfavored to make that action with that particular state. DO NOT draw parallels by saying the reward is like y_i because, as you can see, that is not the case. These parallels will be further solidified with our multi-armed bandit implementation.

MULTI-ARMED BANDIT

For those of you who are unfamiliar, an armed bandit is a slot machine. When you pull the lever, you will either win or lose. A multi-armed bandit will have multiple arms to pull, and each bandit will have a particular lever that will offer the most reward over time. Our rewards will be determined by a normal distribution, where is a particular bandit's lever's value is greater than the normal distribution's random value, then we are awarded a reward of +1. If it is less than the random normal number, we receive a -1 reward. Our RL task is to discover the proper lever to pull (action) for three different bandits (state).

```
bandits = [[-5, -1, 0, 1], [-1, -5, 1, 0], [0, 1, -1, -5]]
num_bandits = len(bandits)
num_actions = len(bandits[0])
```

We will start by assigns arms (actions) to each of our bandits (states). You can think of the arm with the higest value as the arm we should be pulling to get the maximum reward. In our example, each bandit has a different 'best arm' and let's see if our model can figure it out.

```
def pull_arm(self, bandit, action):
    """
    Pull the arm of a bandit and get a positive
    or negative result. (+/- 1)
    """

    # get random number from normal dist.
    answer = np.random.randn(1)

    # Get positive reward if bandit is higher than random
    if bandit[action] > answer:
        return 1
    else:
        return -1
```

We will define a function called pull_arm that takes in a particular bandit and an action (which lever arm in that bandit we selected). We then sample a random number from a normal distribution. Is the action value is greater than it, then the reward is +1, otherwise it's -1. As you can see, the higher the lever arm value, the better chance it has of getting a reward. But we chosen level arm values close to 0 because if we chose value from from the normal distribution mean, our model won't be able to really learn that a lever arm value of 30 is better than 60 because both are extremely far away from the mean and will almost always result in a +1 reward.

```
# Pick a random state
state = np.random.randint(0, num_bandits)
print state

# Determine the action chosen and associated reward
if np.random.rand(1) < 0.25:
    action = np.random.randint(0, num_actions)
else:
    action = np.argmax(sess.run(model.fc1, feed_dict={
        model.state: [state]}))
reward = model.pull_arm(bandits[state], action)

# Store the reward
rewards[state, action] += reward

# Update weights
W, _ = model.step(sess, [state], [action], [reward])
```

So to kick off training, we will pick a random state (bandit). We will then either choose a random arm for the bandit (happens 25% of the time) or we will choose the argmax of the fully connected outputs (happens 75% of the time). The reward will be calculate with our **pull_arm()** function and we will update the model weights based on this reward

```
# Placeholders
self.state = tf.placeholder(name='state',
    shape=[1], dtype=tf.int32)
self.action = tf.placeholder(name='action',
    shape=[1], dtype=tf.int32)
self.reward = tf.placeholder(name='reward',
    shape=[1], dtype=tf.float32)

# One hot encode the state
self.state_one_hot = tf.one_hot(indices=self.state, depth=

# Feed forward net to choose the action
with tf.variable_scope("net"):
    self.W_input = tf.Variable(tf.ones([num_bandits, num_a
z1 = tf.matmul(self.state_one_hot, self.W_input)
self.fc1 = tf.nn.sigmoid(z1)

self.chosen_weight = tf.slice(tf.reshape(self.fc1, [-1, ])
    self.action, [1])
```

```
self.loss = -(tf.log(self.chosen_weight) * self.reward)
self.train_optimizer = tf.train.GradientDescentOptimizer(
    0.001).minimize(self.loss)
```

Above is the actual model itself. It's a simple MLP that processes our input bandit that we chose (state) and determines the action. Recall that the action will be argmax of the full connected outputs (75% of the time) or randomly chosen (25% of the time). The state will be one-hot encoded to be shape [1X3]. We have a set of weights in our net of shape [3X4]. Taking the dot product gives us a [1X4] output which we apply the sigmoid nonlinearity to in order to bound the outputs between 0 and 1. We also could have used any other non-linearity such as tanh or even applied a softmax normalization (doesn't matter too much what we did) because we end up picking the argmax of the [1X4] output as our action. We determine the reward from this and choose the corresponding fully-connected output. The loss function will be the log of this chosen value (acts like our y_pred) and is multiplied by the reward (acts like our y_true).

The results of training on 1000 pulls gives us the correct arms to pull (action) depending on which bandit (state) we are pulling.

```
[[ 0.99461108  0.99542081  0.99946254  1.044976  ]
 [ 0.99434131  0.99622995  1.04705667  1.00107586]
 [ 0.99919343  1.05250132  0.99811631  0.99488103]]
```

In the next post, we will be looking a full scale RL environment where the reward will not be an immeadiete response to the action but is instead delayed.

NUANCES

You may be wondering why we randomly choose the action 25% of the time. Think about just one bandit and two arms having two very close high values (such as 3 and 4). Both of these will usually result is positive rewards since both are far from the mean 0 of the normal distribution but we know that the 4 is better. But if the first update was with the third lever, the reward is positive and the model will keep favoring the 3 without ever knowing that 4 is better. By introducing this random action once in a while, we are exposing the model to the other levers in order to prevent provincial models like this. Here are the rewards look like without any random sampling.

```
[[ -3.   -4.   -1.  258.]
 [ -7.   -5.  208.   -3.]
 [ -1.  210.   -7.   -5.]]
```

You can see that the other lever's almost never get chosen. This can be a problem if our arm's values were very close to each other and far away from the normal distribution's mean.

CODE

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

ADDITIONAL RESOURCES

- **OpenAI RL Tutorial**
- **International Conference on Machine Learning**

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

← *Recurrent Neural Network (RNN) – Part 4: Attentional Interfaces*

Reinforcement Learning (RL) – Policy Gradients II →

2 THOUGHTS ON "REINFORCEMENT LEARNING (RL) – POLICY GRADIENTS I"

Pingback: The Neural Perspective Edit

NORM October 10, 2017 at 10:21 am EDIT REPLY →

If the reward is between 0-1 instead of positive and negative, how will it make a difference? In that case, loss can be negative?

★ Like

LEAVE A REPLY

LEAVE A REPLY

Enter your comment here...