Reinforcement Learning - Part 1

Part 1 - Action-Value Methods and *n*-armed bandit problems

Introduction

I'm going to begin a multipart series of posts on Reinforcement Learning (RL) that roughly follow an old 1996 textbook "Reinforcement Learning An Introduction" by Sutton and Barto. From my research, this text still seems to be the most thorough introduction to RL I could find. The Barto & Sutton text is itself a great read and is fairly approachable even for beginners, but I still think it's worth breaking down even further. It still amazes me how most of machine learning theory was established decades ago yet we've seen a huge explosion of interest and use in just the past several years largely due to dramatic improvements in computational power (i.e. GPUs) and the availibility of massive data sets ("big data"). The first implementations of neural networks date back to the early 1950s!

While really neat results have been achieved using supervised learning models (e.g. Google's DeepDream), many consider reinforcement learning to be the holy grail of machine learning. If we can build a general learning algorithm that can learn patterns and make predictions with unlabeled data, that would be a game-changer. Google DeepMind's Deep Q-learning algorithm that learned to play dozens of old Atari games with just the raw pixel data and the score is a big step in the right direction. Clearly, there is much to be done. The algorithm still struggles with long timespan rewards (i.e. taking actions that don't result in reward for a relatively long period of time), which is why it failed to learn how to play Montezuma's Revenge and similar games. Q-learning is something that was first described in 1989, and while DeepMind's specific implementation had some novelties, it's largely the same algorithm from way back then.

In this series, I will be covering major topics and algorithms in RL mostly from the Barto & Sutton text, but I will also include more recent advances and material where appropriate. My goal (as with all my posts) is to help those with limited mathematical backgrounds to grasp the concepts and be able to translate the equations into code (I'll use Python here). As a heads-up, the code presented here will (hopefully) maximize for readability and understandability often at the expense of computational efficiency and quality. I.e. my code will not be production-quality and is just for enhanced learning. My only assumumptions for this series is that you're proficient with Python and Numpy and have at least some basic knowledge of linear algebra and statistics/probability.

n-armed bandit problem

We're going to build our way up from very simple RL algorithms to much more sophisticated ones that could be used to learn to play games, for example. The theory and math builds on each preceding part, so I strongly recommend you follow this series in order even though the first parts are less exciting.

Let's consider a hypothetical problem where we're at a casino and in a section with some slot machines. Let's say we're at a section with 10 slot machines in a row and it says "Play for free! Max payout is $10!" Wow, not bad right! Let's say we ask one of the employees what's going on here, it seems too good to be true, and she says "It's really true, play as much as you want, it's free. Each slot machine is gauranteed to give you a reward between 0 and $10. Oh, by the way, keep this on the down low but those 10 slot machines each have a different average payout, so try to figure out which one gives out the most rewards on average and you'll be making tons of cash!"

What kind of casino is this?! Who knows, but it's awesome. Oh by the way, here's a joke: What's another name for a slot machine? .... A one-armed bandit! Get it? It's got one arm (a lever) and it generally steals your money! Huh, well I guess we could call our situation a 10-armed bandit problem, or an *n*-armed bandit problem more generally, where *n* is the number of slot machines.

Let me restate our problem more formally. We have *n* possible actions (here *n* = 10) and at each play (*k*) of this "game" we can choose a single lever to pull. After taking an action a we will receive a reward R_k (reward at play *k*). Each lever has a unique probability distribution of payouts (rewards). For example, if we have 10 slot machines, slot machine #3 may give out an average reward of $9 whereas slot machine \#1 only gives out an average reward of $4. Of course, since the reward at each play is probabilistic, it is possible that lever #1 will by chance give us a reward of $9 on a single play. But if we play many games, we expect on average slot machine #1 is associated with a lower reward than #3.

Thus in words, our strategy should be to play a few times, choosing different levers and observing our rewards for each action. Then we want to only choose the lever with the largest observed average reward. Thus we need a concept of *expected* reward for taking an action a based on our previous plays, we'll call this expected reward $Q_k(a)$ mathematically. $Q_k(a)$ is a function that accepts action a and returns the expected reward for that action. Formally,

$$Q_k(a) = \frac{R_1 + R_2 + \ldots + R_k}{k_a}$$

That is, the expected reward at play *k* for action a is the *arithmetic mean* of all the previous rewards we've received for taking action *a*. Thus our previous actions and observations influence our future actions, we might even say some of our previous actions *reinforce* our current and future actions. We'll come back to this later.

Some keywords for this problem are exploration and exploitation. Our strategy needs to include some amount of exploitation (simply choosing the best lever based on what we know so far) and some amount of exploration (choosing random levers so we can learn more). The proper balance of exploitation and exploration will be important to maximizing our rewards.

So how can we come up with an algorithm to figure out which slot machine has the largest average payout? Well, the simplest algorithm would be to select action a for which this equation is true:

$$Q_k(A_k) = max_a(Q_k(a))$$

This equation/rule states that the expected reward for the current play *k* for taking action A is equal to the maximum average reward of all previous actions taken. In other words, we use our above reward function $Q_k(a)$ on all the possible actions and select the one that returns the maximum average reward. Since $Q_k(a)$ depends on a record of our previous actions and their associated rewards, this method will not select actions that we haven't already explored. Thus we might have previously tried lever 1 and lever 3, and noticed that lever 3 gives us a higher reward, but with this method, we'll never think to try another lever, say #6, which, unbeknownst to us, actually gives out the highest average reward. This method of simply choosing the best lever that we know of so far is called a "greedy" method.

Obviously, we need to have some exploration of other levers (slot machines) going on to discover the true best action. One simple modification to our above algorithm is to change it to an ϵ (epsilon)-greedy algorithm, such that, with a probability ϵ, we will choose an action a at random, and the rest of the time (probability $1 - \epsilon$) we will choose the best lever based on what we currently know from past plays. So most of the time we play greedy, but sometimes we take some risks and choose a random lever and see what happens. This will of course influence our future greedy actions.

Alright, I think that's an in-depth enough discussion of the problem and how we want to try to solve it with a rudimentary RL algorithm. Let's start implementing this with Python.

In [579]:
```
#imports, nothing to see here
import numpy as np
from scipy import stats
import random
import matplotlib.pyplot as plt
%matplotlib inline
```

In [595]:
```
n = 10
arms = np.random.rand(n)
eps = 0.1
```

Per our casino example, we will be solving a 10-armed bandit problem, hence *n* = 10. I've also defined a numpy array of length *n* filled with random floats that can be understood as probabilities. The way I've chosen to implement our reward probability distributions for each arm/lever/slot machine is this: Each arm will have a probability, e.g. 0.7. The maximum reward is $10. We will setup a for loop to 10 and at each step, it will add +1 to the reward if a random float is less than the arm's probability. Thus on the first loop, it makes up a random float (e.g. 0.4). 0.4 is less than 0.7, so reward += 1. On the next iteration, it makes up another random float (e.g. 0.6) which is also less than 0.7, thus reward += 1. This continues until we complete 10 iterations and then we return the final total reward, which could be anything 0 to 10. With an arm probability of 0.7, the *average* reward of doing this to infinity would be 7, but on any single play, it could be more or less.

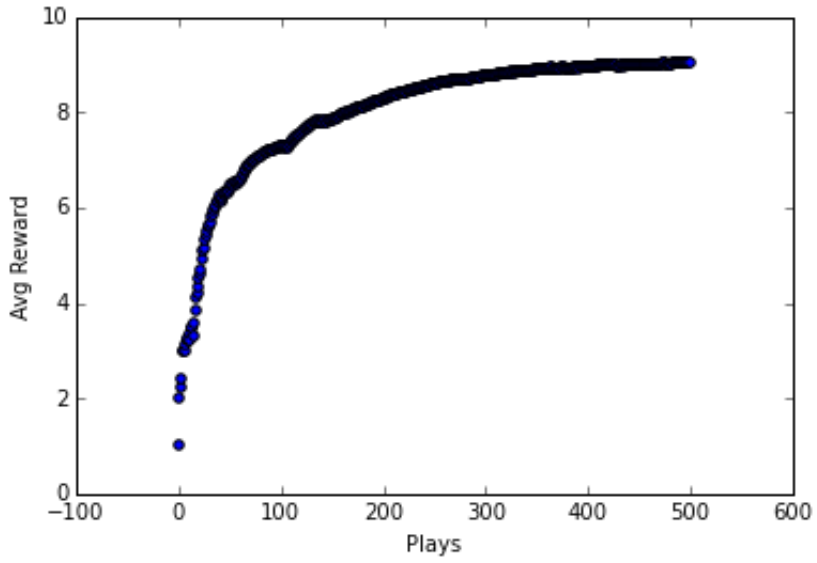
In [590]:
```
def reward(prob):
    reward = 0;
    for i in range(10):
        if random.random() < prob:
            reward += 1
    return reward
```

The next function we define is our greedy strategy of choosing the best arm so far. This function will accept a memory array that stores in a key-value sort of way the history of all actions and their rewards. It is a $2\ x\ k$ matrix where each row is an index reference to our arms array (1st element) and the reward received (2nd element). For example, if a row in our memory array is [2, 8] it means that action 2 was taken (the 3rd element in our arms array) and we received a reward of 8 for taking that action.

In [596]:
```
#initialize memory array; has 1 row defaulted to random action index
av = np.array([np.random.randint(0,(n+1)), 0]).reshape(1,2) #av = action-value

#greedy method to select best arm based on memory array (historical results)
def bestArm(a):
    bestArm = 0 #just default to 0
    bestMean = 0
    for u in a:
        avg = np.mean(a[np.where(a[:,0] == u[0])][:, 1]) #calc mean reward for each action
        if bestMean < avg:
            bestMean = avg
            bestArm = u[0]
    return bestArm
```

And here is the main loop for each play. I've set it to play 500 times and display a matplotlib scatter plot of the mean reward against plays. Hopefully we'll see that the mean reward increases as we play more times.

In [597]:
```
plt.xlabel("Plays")
plt.ylabel("Avg Reward")
for i in range(500):
    if random.random() > eps: #greedy arm selection
        choice = bestArm(av)
        thisAV = np.array([[choice, reward(arms[choice])]])
        av = np.concatenate((av, thisAV), axis=0)
    else: #random arm selection
        choice = np.where(arms == np.random.choice(arms))[0][0]
        thisAV = np.array([[choice, reward(arms[choice])]]) #ch
oice, reward
        av = np.concatenate((av, thisAV), axis=0) #add to our a
ction-value memory array
    #calculate the percentage the correct arm is chosen (you ca
n plot this instead of reward)
    percCorrect = 100*(len(av[np.where(av[:,0] == np.argmax(arm
s))])/len(av))
    #calculate the mean reward
    runningMean = np.mean(av[:,1])
    plt.scatter(i, runningMean)
```



As you can see, the average reward does indeed improve after many plays. Our algorithm is *learning*, it is getting reinforced by previous good plays! And yet it is such a simple algorithm.

I encourage you to download this notebook (scroll to bottom) and experiment with different numbers of arms and different values for ϵ.

The problem we've considered here is a *stationary* problem because the underlying reward probability distributions for each arm do not change over time. We certainly could consider a variant of this problem where this is not true, a non-stationary problem. In this case, a simple modification would be to weight more recent action-value pairs greater than distant ones, thus if things change over time, we will be able to track them. Beyond this brief mention, we will not implement this slightly more complex variant here.

Incremental Update

In our implementation we stored each action-value (action-reward) pair in a numpy array that just kept growing after each play. As you might imagine, this is not a good use of memory or computational power. Although my goal here is not to concern myself with computational efficiency, I think it's worth making our implementation more efficient in this case as it turns out to be actually simpler.

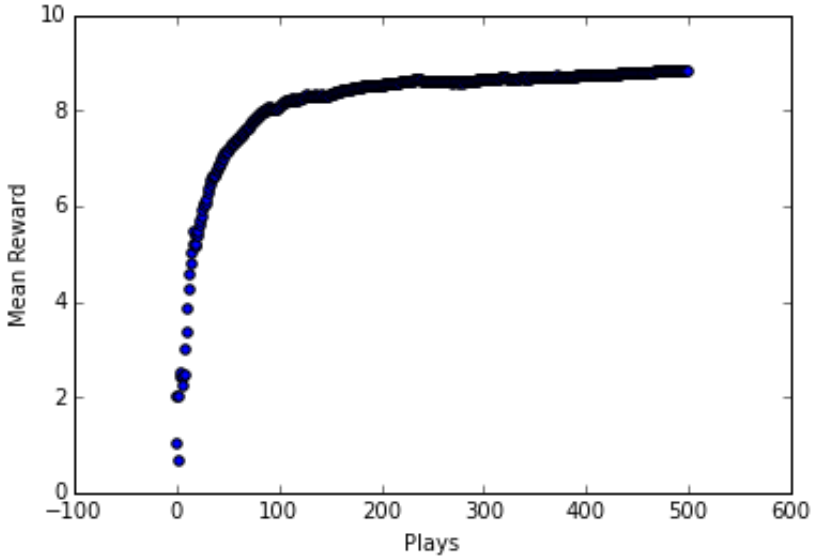
Instead of storing each action-value pair, we will simply keep a running tab of the *mean* reward for each action. Thus we reduce our memory array from virtually unlimited in size (as plays increase indefinitely) to a hard-limit of a 1-dimensional array of length *n* (n = # arms/levers). The index of each element corresponds to an action (e.g. 1st element corresponds to lever #1) and the value of each element is the running average of that action.

Then whenever we take a new action and receive a new reward, we can simply update our running average using this equation:

$$Q_{k+1} = Q_k + \frac{1}{k}[R_k - Q_k]$$

where Q_k is the running average reward for action a so far and R_k is the reward we received right now for taking action A_k, and k is the number of plays so far.

In [720]:
```
n = 10
arms = np.random.rand(n)
eps = 0.1

av = np.ones(n) #initialize action-value array
counts = np.zeros(n) #stores counts of how many times we've tak
en a particular action

def reward(prob):
    total = 0;
    for i in range(10):
        if random.random() < prob:
            total += 1
    return total

#our bestArm function is much simpler now
def bestArm(a):
    return np.argmax(a) #returns index of element with greatest
value

plt.xlabel("Plays")
plt.ylabel("Mean Reward")
for i in range(500):
    if random.random() > eps:
        choice = bestArm(av)
        counts[choice] += 1
        k = counts[choice]
        rwd =  reward(arms[choice])
        old_avg = av[choice]
        new_avg = old_avg + (1/k)*(rwd - old_avg) #update runni
ng avg
        av[choice] = new_avg
    else:
        choice = np.where(arms == np.random.choice(arms))[0][0]
#randomly choose an arm (returns index)
        counts[choice] += 1
        k = counts[choice]
        rwd =  reward(arms[choice])
        old_avg = av[choice]
        new_avg = old_avg + (1/k)*(rwd - old_avg) #update runni
ng avg
        av[choice] = new_avg
    #have to use np.average and supply the weights to get a wei
ghted average
    runningMean = np.average(av, weights=np.array([counts[i]/n
p.sum(counts) for i in range(len(counts))]))
    plt.scatter(i, runningMean)
```



This method achieves the same result, getting us better and better rewards over time as it learns which lever is the best option. I had to create a separate array `counts` to keep track of how many times each action is taken to properly recalculate the running reward averages for each action. Importantly, this implementation is simpler and more memory/computationally efficient.

Softmax Action Selection

Imagine another type of bandit problem: A newly minted doctor specializes in treating patients with heart attacks. She has 10 treatment options of which she can choose only one to treat each patient she sees. For some reason, all she knows is that these 10 treatments have different efficacies and risk-profiles for treating heart attacks, and she doesn't know which one is the best yet. We could still use our same ϵ-greedy algorithm from above, however, we might want to reconsider our ϵ policy of completely randomly choosing a treatment once in awhile. In this new problem, randomly choosing a treatment could result in patient death, not just losing some money. So we really want to make sure to not choose the worst treatment but still have some ability to explore our options to find the best one.

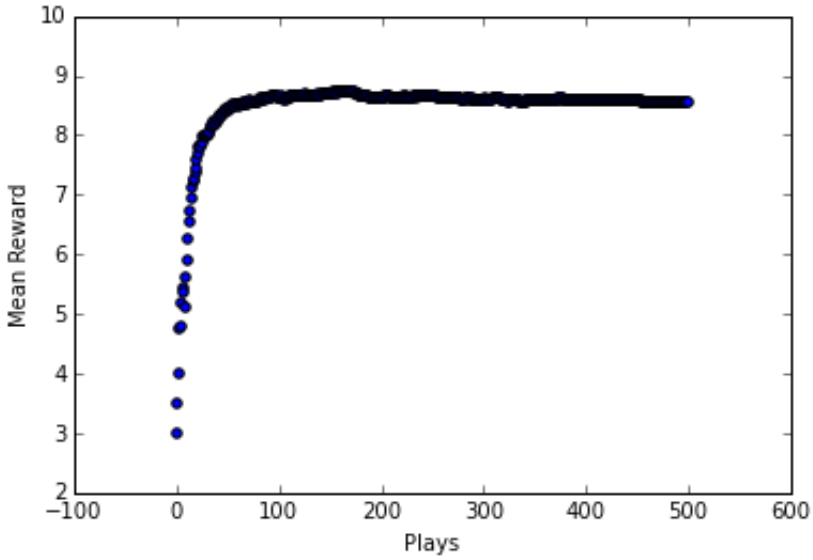
This is where a softmax selection might be the most appropriate. Instead of just choosing an action at random during exploration, softmax gives us a probability distribution across our options. The option with the largest probability would be equivalent to our best arm action from above, but then we have some idea about what are the 2nd and 3rd best actions for example. This way, we can randomly choose to explore other options while avoiding the very worst options. Here's the softmax equation:

$$\frac{e^{Q_k(a)/\tau}}{\sum_{i=1}^{n} e^{Q_k(i)/\tau}}$$

τ is a parameter called temperature the scales the probability distribution of actions. A high temperature will tend the probabilities to be very simmilar, whereas a low temperature will exaggerate differences in probabilities between actions. Selecting this parameter requires an educated guess and some trial and error.

When we implement the slot machine 10-armed bandit problem from above using softmax, we don't need our `bestArm()` function anymore. Since softmax produces a weighted probability distribution across our possible actions, we will just randomly (but weighted) select actions according to their relative probabilities. That is, our best action will get chosen more often because it will have the highest softmax probability, but other actions will be chosen at random at lesser frequency.

In [781]:
```
n = 10
arms = np.random.rand(n)

av = np.ones(n) #initialize action-value array, stores running
reward mean
counts = np.zeros(n) #stores counts of how many times we've tak
en a particular action
#stores our softmax-generated probability ranks for each action
av_softmax = np.zeros(n)
av_softmax[:] = 0.1 #initialize each action to have equal proba
bility

def reward(prob):
    total = 0;
    for i in range(10):
        if random.random() < prob:
            total += 1
    return total

tau = 1.12 #tau was selected by trial and error
def softmax(av):
    probs = np.zeros(n)
    for i in range(n):
        softm = ( np.exp(av[i] / tau) / np.sum( np.exp(av[:] /
tau) ) )
        probs[i] = softm
    return probs

plt.xlabel("Plays")
plt.ylabel("Mean Reward")
for i in range(500):
    #select random arm using weighted probability distribution
    choice = np.where(arms == np.random.choice(arms, p=av_softm
ax))[0][0]
    counts[choice] += 1
    k = counts[choice]
    rwd =  reward(arms[choice])
    old_avg = av[choice]
    new_avg = old_avg + (1/k)*(rwd - old_avg)
    av[choice] = new_avg
    av_softmax = softmax(av) #update softmax probabilities for
next play

    runningMean = np.average(av, weights=np.array([counts[i]/n
p.sum(counts) for i in range(len(counts))]))
    plt.scatter(i, runningMean)
```



Softmax action selection seems to do at least as well as epsilon-greedy, perhaps even better; it looks like it converges on an optimal policy faster. The downside to softmax is having to manually select the τ parameter. Softmax here was pretty sensitive to τ and it took awhile of playing with it to find a good value for it. Obviously with epsilon-greedy we had the parameter epsilon to set, but choosing that parameter was much more intuitive.

Conclusion

Well that concludes Part 1 of this series. While the *n*-armed bandit problem is not all that interesting, I think it does lay a good foundation for more sophisticated problems and algorithms.

Stay tuned for part 2 where I'll cover finite Markov decision processes and some associated algorithms.

Download this IPython Notebook

https://github.com/outlace/outlace.github.io/blob/master/ipython-notebooks/rlpart1.ipynb (https://github.com/outlace/outlace.github.io/blob/master/ipython-notebooks/rlpart1.ipynb)

References:

1. "Reinforcement Learning: An Introduction" Andrew Barto and Richard S. Sutton, 1996
2. https://en.wikipedia.org/wiki/Artificial_neural_network#History (https://en.wikipedia.org/wiki/Artificial_neural_network#History)
3. https://en.wikipedia.org/wiki/Q-learning (https://en.wikipedia.org/wiki/Q-learning)

Written on October 19, 2015