Reinforcement Learning 1: Multi-Armed Bandit

Eckel, TJHSST AI2, Spring 2024

Background & Explanation

Reinforcement learning is an immense and complex subfield of AI. The basic concepts predate deep learning (have been developed for decades). Recently, deep learning has been applied to reinforcement learning in an algorithm called Deep Q-Learning. We'll get as close to that as we can; this is a new unit – let's see what happens!

The basic purpose of reinforcement learning is to build an AI that can be put into a game, as a player, and figure out an ideal strategy from scratch based on feedback it receives while playing. In other words, it does not have previous knowledge of what moves are good and determines that itself in the process of playing the game. (This is an extremely basic definition, and it should be clear that reinforcement learning can apply to a wide variety of situations – anything that can be modelled as an environment that returns rewards after specific actions – and we'll get more technical about this later. But for now: this is the prototypical purpose of reinforcement learning.) Deep Q-Learning has recently achieved enormous results on games originally thought impossible for an AI to master, most especially Go, and truly wild results like learning how to play Atari arcade games given *only the direct input of the RGB values on the screen* and a score. We, as you might imagine, won't be able to recreate that. But we will understand the basic principles and get a sense of how a sophisticated tool like that comes together.

We're going to start with a very simple problem, often called the multi-armed bandit.

Bandits

So let's say there's a machine, we'll call it a bandit, that randomly doles out *rewards* according to a normal distribution. (The word *reward* here is a technical term; it doesn't always mean something good. A reward could be negative.)

As you probably recall, a normal distribution is defined by a mean and a standard deviation.

A straightforward statistics problem is just to try and find the mean and the standard deviation that the bandit is using from playing it over and over again. The law of large numbers says that if you keep asking it for data points, the mean and standard deviation of what you get will converge to the real values. No problem.

But let's say that there are many bandits, and you don't know any of their means or standard deviations. The game is this:

- One *move*, for you the player, is to pick one of the bandits, and get a reward from that bandit (generated according to its normal distribution).
- You get a limited, finite number of moves; your goal is to get the highest possible total reward after all those moves.
- Of course, you can keep track of the results of every move you make, and use that information to determine your next move.

What is your ideal strategy?

Policies

The basic component of a reinforcement learning algorithm is a policy. The policy determines which choice is made at each available move. And in general, there's a fundamental conflict at the heart of any reinforcement learning policy – do you want *more information* or do you want to *make the best move with the information you have*? This is referred to as *exploration* vs *exploitation*. Learn more, or exploit the knowledge you have. This conflict is at the heart of reinforcement learning algorithms, and playing with it here is the foundation we'll build on later.

In this specific case, the conflict goes like this – clearly, if we know which bandit has the highest mean, then we will just play that one every time. But if we don't know that, how do we balance playing the one that looks the best so far with experimenting to determine if another one might be better?

As we play, we'll keep track of a quality score (or Q score for short, also called Q value) for each bandit which is our current best estimate of each bandit's mean. Using these Q scores, we can define a few possible policies:

Put simply:

- A *greedy* policy just picks whatever bandit currently has the best Q value.
  - A technical point: before making any moves, each bandit has a starting Q value of 0. At that point, choosing any bandit is equally likely. Once we make at least one move, then we use that information to update that bandit's Q value, after which it will be rated against the other unchosen default 0 values. So, if the first reward given is negative, this algorithm will consider that bandit worse than all the other 0s and end up choosing one of those 0s at step two. In other words, it is *not* the case that the greedy algorithm simply plays one bandit once and then keeps playing that bandit forever because it has no information about the others.
- A *random* policy chooses the next move entirely at random (and thus generates as much information as possible, but uses none of it).
- An *epsilon-greedy* policy chooses between the previous two options with a certain probability. Epsilon is defined as the chance that the policy chooses a random move instead of a greedy one. For instance, a 0-greedy policy is the same as a purely greedy one (0% chance of choosing randomly instead), and a 1-greedy policy is the same as a purely random one (100% chance of choosing randomly instead).

We will compare epsilon values of 0, 0.001, 0.01, 0.1, 0.5, and 1 on the one-armed bandit problem and figure out which one is likely to give us the best results.

Making Bandits

The random library that comes with Python comes with a method to generate a random value from a normal distribution. It is `random.normalvariate(mean, standard_deviation)`.

We'll use a standard deviation of 1 for all of our bandits. If you want to, you can mess with that and see if it makes a difference, but it seems a reasonable simplifying assumption for now. As a result, we define a bandit with just its mean. So, if we want to make a bunch of bandits, we just want to pick a bunch of means.

We'll do this with a normal distribution too. We'll choose each bandit's mean with `random.normalvariate(0, 1)` called as many times as we need.

Compare Policies

Call the following procedure one trial:

- Generate 10 random bandits.
- For each epsilon value in [0, 0.001, 0.01, 0.1, 0.5, 1]:
    - Play 200 games of the multi-armed bandit using an epsilon-greedy strategy against the bandits you generated. (Do not regenerate the bandits each time; this is 200 different games against the same set of bandits, to avoid noise from initial conditions.)
        - Each game should last exactly 2,000 moves.
    - Average this policy's score across all 200 games.
    - Store this average policy score.
- Finally, just for comparison's sake, let's also imagine playing perfectly. Directly determine the maximum mean from the generated bandits, and multiply that by 2,000 to make an "ideal" score.

Just for good measure, let's run 10 trials, so we're not overly dependent on one particular random generation of bandits.

Average the 10 average scores that each policy has gotten.

Print each policy and its average score, along with the average of the 10 "ideal" scores.

Message me your results on Mattermost. Which policy won? How close was it to ideal performance? (Hint: if either 0 or 1 comes out on top, you've done something wrong!)

Including Uncertainty

The previous exercise should've shown you clearly the difference between exploring and exploiting, and how both are necessary to some degree.

It's not too great a leap, though, to think that we might base our exploration off of something *slightly* more complicated than just a random choice. Specifically: why not also utilize our uncertainty? That is, why not choose bandits we're *unsure of* more often, just to make sure we get some good samples off of each one?

We're storing Q-values for each bandit – the average of each bandit's results so far. Continue to keep those values. But, when it comes time to greedily choose the next move, don't just base that on the highest Q-value. Take each Q-value and add an upper confidence bound term to each, increasing each value by a measure of how uncertain we are of its accuracy. Then choose the maximum of *those* numbers, so uncertain outcomes are chosen more frequently.

There are a lot of ways to do this; here's the one I found:

$$q + \sqrt{\frac{\ln(N)}{n}}$$

…where q is a particular bandit's Q-value, n is the number of times this particular bandit has been selected, and N is the total number of moves taken so far.

Of course, we can't divide by 0, so the bottom of that fraction will need to be `(n + 1e-10)` to avoid math errors.

Compare Policies With and Without Upper Confidence Bound

Run the same 10 trials as before, but add in two more strategies:

- A greedy policy using the UCB algorithm above
- An epsilon-greedy policy using the best epsilon you found from your previous trials and the UCB algorithm above each time “greedy” is chosen

This means at the end you should output 9 different clearly labelled values:

- Random
- 0.001-greedy
- 0.01-greedy
- 0.1-greedy
- 0.5-greedy
- Greedy
- Greedy w/ UCB
- (your favorite epsilon)-greedy w/ UCB
- “Ideal”

…all averaged across 10 trials as explained above.

Once again, **send me your results on Mattermost**, all 9 values. Then, briefly analyze your values. What have you learned from this assignment?

Get Your Code Ready To Turn In

When you turn your code in, I want it to do exactly what I said above, just with **only one trial** instead of ten (takes too long to run on the grader). I’ll know to expect some variation as a result, don’t worry.

Specification

Submit **a single python script** to the link on the course website.

This assignment is **complete** if:

- You follow the instructions on the submission form to format your submission properly.
- You have **sent me your results and analysis on Mattermost** when requested.
- Your code runs **one** (1) trial of all 9 policies, and prints each policy’s average after 200 games, clearly labelled.