Reinforcement Learning 3: Blackjack

Eckel, TJHSST AI2, Spring 2024

Background & Explanation

Blackjack! You've probably played before. Our version will be a little bit simplified, and we'll see how well we can do against the computer. (We won't be able to win, but we'll see how little we can lose.)

In our version, the deck has an infinite number of each possible card, and at any moment any card is equally likely to be drawn. As such, we won't need to keep track of suits. We just have 13 equally likely cards – 2 through 10, J, Q, K, and A.

In Blackjack, number cards are worth their number shown, face cards are worth 10, and A is worth either 1 or 11 (more on this below).

You get dealt two cards. The dealer gets dealt two cards. You get to see one of the dealer's cards, the other is hidden. Then, you can "hit" (ask for another card) as many times as you like (including 0). If your sum goes above 21, you "go bust" and you automatically lose. Alternately, you can choose to "stick" at any point. Once you "stick", you see the dealer's hidden card, and then the dealer plays their hand according to a fixed strategy – on any total that is 16 or less, they hit, and on any total 17 or higher, they stick. If the dealer goes bust, you win. If neither of you goes bust, whoever has the highest total wins. If you have the same total, you tie. We'll say that this is a reward of 1 for winning, 0 for tying, and -1 for losing. There are no intermediate rewards; the reward for each move is 0 until the game is over and the final reward is given.

The ace thing is bit complicated – you can choose to have an ace in your hand be either 11 or 1. In practice, this means that an ace is worth 11 until you get a total above 21, in which case you subtract 10 from that total and now the ace is worth 1. This means that you need to know more than just your current sum in order to define a strategy; whether or not you have a "usable" ace makes a difference – you want to hit more often if you have an ace currently worth 11 because the risk is lower. (As a quick tip, in terms of modeling the game, I recommend setting the value of an ace at 1, and simply adding 10 to the value of any hand that contains an ace and otherwise would have a value of 11 or less.)

So, we'll define a game state as the tuple of (Boolean, int, int): (usable_ace_in_your_hand, your_sum, dealer_showing_value)

What we want is the q-value of each action at each state. For example, a single q-value might be the expected total future reward of choosing to "hit" when I have a hand worth 15 total, I have no ace, and the dealer is showing a 7. Because there are no intermediate rewards in this game, only final rewards, this means the word "expected" can be thought of strictly in the sense of "expected value" – what is the expected value of my reward at the end of the game if I choose this play?

Monte Carlo Simulation

Technically we could do this exactly the same way we did the probabilistic warps in the previous assignment. The q_value at a certain game state is the immediate reward (zero) plus the sum of the likelihood of arriving at each possible next state times the max q_value at that state. This isn't even that bad… until you consider each move to "stick". Choosing to "stick" ends the game, but each game ending final state has a different probability of winning, losing, and tying, and these are not trivial to calculate. I'm sure you could calculate the chances of winning if you stick with an 18 and the dealer is showing a 2, but that's not an easy calculation.

The good news is that actually getting our code to just *play blackjack* isn't so hard. Just use random.choice() to pick a card each time you need to pick a card, use some kind of policy for your own play, and hardcode the dealer's policy when it's their turn. Return the reward at the end.

So what we're going to do is combine our setup for grid world with our strategy for the bandits – we're going to use a 0.1-greedy strategy to explore/exploit as we play, and begin the game with each q-value being initialized to 0, updating as we play. Specifically, we'll play a game out to completion storing each (state, action) pair along the way. At the end, we'll get the reward (1, 0, or -1), then we'll add that reward to a list of outcomes we keep with each (state, action) pair. Repeat. The average of those outcomes will be our q-value for that (state, action) pair. When we choose to be greedy, we'll pick whichever action for the current state has the higher q-value at that moment. When we choose to be random, we'll pick an action randomly.

Let's build this up piece by piece.

Model Blackjack

There are certain states in the game where the move you need to make is completely obvious and should not be included in any q-value or policy list – the right move should just happen automatically. Specifically:

- If you have 21, your strategy must stick. There is no possible advantage to hitting.
- If you have 11 or less, your strategy must hit. There is no possible advantage to sticking.

The interesting part is when your sum is between 12 and 20, inclusive, so that's where your policy will determine your move.

Model the game and make it so your code can play out 1,000,000 games of blackjack where you use a random policy (ie, choose your move at random if your total is between 12 and 20 inclusive), and the dealer plays as they're supposed to.

Think of this like playing a million games of blackjack where if you win, you get a dollar, and if you lose, you lose a dollar. Total up the rewards you get and print out your total reward at the end. As a random player, what money do you lose? Send me a message on Mattermost.

Now Make q-Values

Generate a list of each state that we care about (each state where the player's total is between 12 and 20 inclusive). Remember that a state is the tuple of (Boolean, int, int): (usable_ace_in_your_hand, your_sum, dealer_showing_value).

For each state, there are two moves – "hit" or "stick". Initialize both q-values to 0.

I recommend keeping these q-values as rolling averages, not as lists of outcomes you manually average each time. To keep a rolling average, keep an average of all the outcomes so far and a count of how many outcomes there have been. When you get a new reward:

```
new_average = (old_average * count + new_reward) / (count + 1)
count = count + 1
```

If you plan to work that way, you'll also need a separate data structure storing those counts – again, a separate count for each state for each move.

Do the Monte Carlo Simulation

Now play 1,000,000 games of blackjack with a 0.1-greedy strategy. At each move, either choose randomly with probability 10% or greedily with probability 90%. Keep track of each (state, action) pair chosen throughout the game. When you get the final reward, update the averages and counts at each of those (state, action) pairs accordingly.

Also keep track of your total rewards over all 1,000,000 games as you play this way. (You should still expect this to be negative, just a lot *less* negative.)

Finally, use the q-values you have worked so hard to determine to play another new set of 1,000,000 games using a **fully greedy** strategy on those q-values. (In other words, if you play with all of that information already obtained, what advantage does that get you? On Mattermost, send me your losses over a million games *while learning* and then separately your losses over a million games *while using the strategy you've developed the whole time*.

Get Your Code Ready to Turn In

I'm interested in your final, purely greedy policy. What do your final calculated q-values say you should do in each circumstance? Here's one efficient way to output this information.

Take the situations where you **don't** have a usable ace. For each possible card that the dealer is showing, if you increase your sum from 11 (always hit) to 21 (always stick), there should come a value in between 11 and 21 where you switch from hit to stick. For instance, if the dealer is showing 10, it turns out that you should hit up to and including a hand value of 16, and stick if you have 17 or higher. So, if you have no ace and the dealer is showing 10, your **smallest stick value** is 17.

Considering the situations where you **don't** have a usable ace, output each possible card the dealer might be showing along with the **smallest stick value** for your hand in that situation.

Then, do the same thing again for all situations where you **do** have a usable ace.

(Note well: if your q-values don't fit this paradigm – if as you consider increasing values for the sum in your hand, you see that your q-values go from hit to stick and back to hit and then to stick again, you've done something wrong.)

Specification

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

It takes no command line arguments.

Run 1,000,000 games from a q-value initialization of all zeroes using a 0.1-greedy policy. Output:

- Your total reward
- The smallest stick value your policy has for each card the dealer could be showing, as two separate sets of numbers, one if you have a usable ace and one if not, as described in the previous section

This assignment is **complete** if:

- You follow the instructions on the submission form to format your submission properly.
- Your script works as described.