Mission 3

Adam Morris – Computational Social Cognition Bootcamp, July 2017
thatadammorris.com thatadammorris@gmail.com

Your agent is dominating the slot machines, and the casino is losing money fast. One day, when your agent arrives at the casino, the slot machines are mysteriously "out of order". In their place, an ominous blackjack table has appeared.

Congratulations on completing Mission 2. If you want to compare your code to mine, my solution is available in "double_slot_machine.m" (online here.)

At this point, you've implemented Q-learning on a basic task with a simple state space. Now, we're going to make the task more complicated.

In this mission, you will program your agent to play simple version of blackjack, called "solo blackjack". Here's how the game works. You get dealt two cards. Your goal is to have the sum of your cards be as close to 21 as possible. (Cards 2-9 are worth their face value; all face cards [i.e. jack, queen, king] are worth 10, and ace is always worth 11.)

You observe two initial cards, and then make a single choice. You can "stay", which means that you keep the cards you have. Or you can "hit", which means you gets dealt an extra card. (For now, assume that you can only hit a single time.)

After you make your choice, you get a reward of 10 if the sum of your cards is between 10 and 15; a reward of 20 if the sum is between 15 and 20; and a reward of 100 if the sum is exactly 21. But if your sum goes over 21, you "bust" and get a reward of -50.

Part A

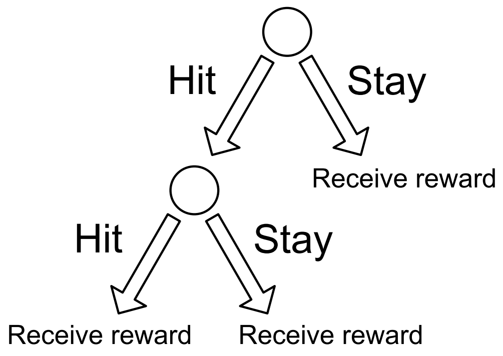
Simulate a Q-learning agent playing this game. (You should run the agent for 100,000 rounds, and set the learning rate to 0.01 and the inverse temperature to 0.1.) Think carefully about what state space you should use. What information is relevant to the agent's decision? Hint: You will have many more states in this task than you did in previous missions.

To make your job easier, don't bother to keep track of what cards are in the deck. Anytime you want to deal a card, simply call the function `getRandomCard()`, which you can download from the github (here).

When the simulations are done, plot the agents final value estimates in each state. (To do this, store your learned Q values in a numStates x numActions matrix, and call `plotResults(Qvalues, 3)`. For this to work, you should label "hit" as action 1, and "stay" as action 2. **Note that you're now sending `plotResults` a different matrix than before!**) When does the agent prefer to hit or stay?

Part B

Let's make the game more interesting. Now, the agent has the potential to make two choices. If they stay, the round ends. But if they hit, they can see their third card and then decide whether to hit again. So the decision tree looks something like the image to the right.



Simulate the agent playing this modified game with the same parameters as before. (Be careful about implementing the proper updating rules for your Q values!) Plot the final value estimates, and compare to Part A.