Introduction to Reinforcement Learning Tutorial

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

007:

This is MI6. Hope you got out of Buenos Aires alive.

We've gotten wind of a new evil plan to conquer the world, and we know that the plan is being financed by a casino out in Budapest. We need someone to infiltrate it and beat the casino owners at their own game.

Unfortunately, due to budget cuts, we've replaced all our secret agents with reinforcement learning agents. We need you to program an RL agent that can maximize its earnings at these casino games. Good luck.

- M

This tutorial is a practical introduction to the nitty-gritty of programming reinforcement learning models. It is targeted at social scientists who ultimately want to model human decision-making, but have never simulated an RL agent before. It will employ simple models in simple tasks, of the form currently used in psychology and cognitive science. (It is not meant for, say, computer scientists in machine learning; I will brazenly ignore all technical details.)

I will assume that you have already learned the theoretical basics of RL models. For instance, you should know what Q-learning is, and be able to write out the formula for a softmax decision function.[1] If these concepts are new to you, you should read the first few chapters of Sutton & Barto's classic textbook (or its second edition here).[2]

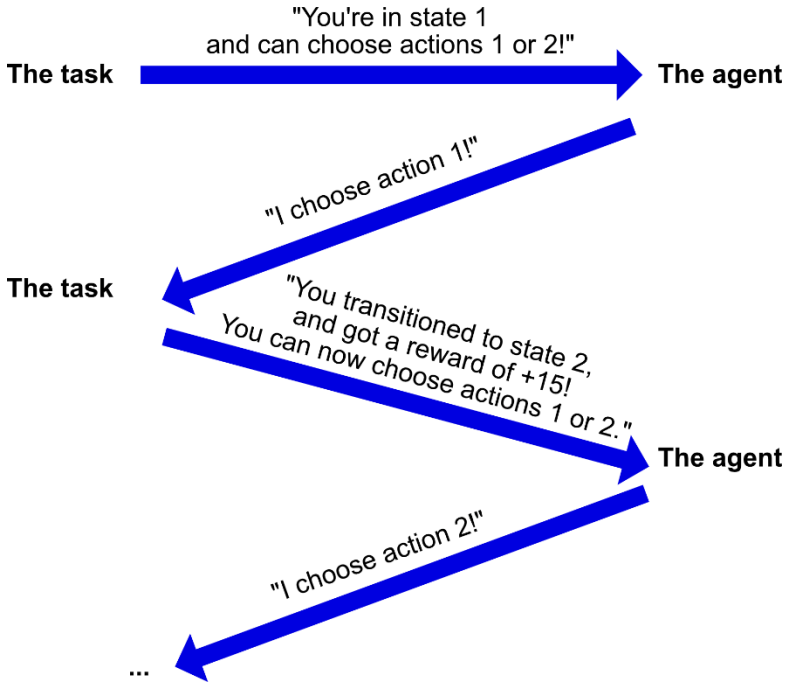
This tutorial will not teach you the theory of RL. Rather, it will assist you in translating that theory into practice. In my own experience, that translation is often the most difficult part. This tutorial is an attempt to share whatever practical wisdom I've acquired on this matter.

One other important note: This tutorial is about simulating RL agents, not fitting RL models to human choices. You should learn the former before the latter. After completing this tutorial, you can check out Nathaniel Daw's model-fitting introduction or Sam Gershman's model-fitting package.

[1] What I call softmax is also called a logistic, sigmoid, or Boltzmann function.
[2] I will also assume basic programming knowledge. All my example code is in MATLAB, but, if you want to strike out on your own, you can of course use other languages.

Two challenges

When you sit down to simulate a reinforcement learning agent, there are two things you must do. You must program the task, and you must program the agent performing that task. These are relatively separable challenges.



Communication between the task and agent. Your code will flow something like this.

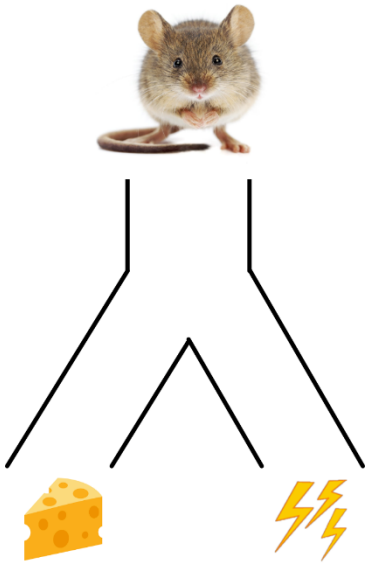
The task. Informally, you can think of the task as the "game" your agent is playing. In this tutorial, you will literally be programming casino games for your agent to play. But in your research, the game is usually whatever experiment you plan to run on humans. If you want to model how humans make a series of 10 decisions in various conditions, then your game will have 10 decisions which implement those various conditions. If you want to model how humans choose between two social partners after experiencing different rewards with each of them, then this is the game you will implement.

Formally, to program the task, you must specify four things. You must specify which states the agent can be in; which actions the agent can take in each state; what reward the agent gets after selecting an action; and which state the agent goes to next. If you've specified these four things, then you're most of the way there.

You might cry: "But Adam! My task is just a vague idea in my head. How do I translate that idea into these precise constructs?"

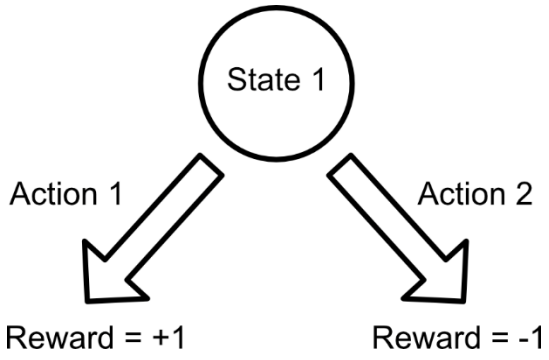
Therein lies the challenge. I'll illustrate how this is done via some examples.

Imagine that your agent is a rat, and your task is a maze. We'll start with the simplest possible maze. In this maze, there's only a single decision: go left, or go right. If the rat goes left, he gets the cheese; if he goes right, he gets shocked.



Poor rat.

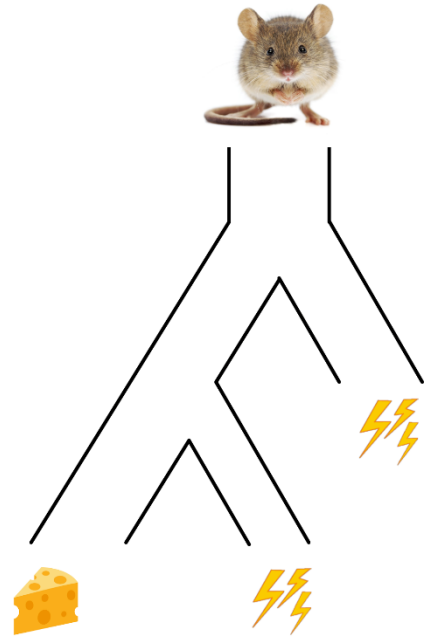
Here's what this looks like, translated into a reinforcement learning task.



States represent decision points. In this maze, there was only one decision – so there is only one state. That decision has two possible options, so there are two possible actions from the state. One action leads to something the rat likes; one action leads to something the rat dislikes. Those outcomes are captured by the reward function.[3] Finally, when the rat finishes the maze, we pull it out, plop it back in state one, and let it run the maze again.

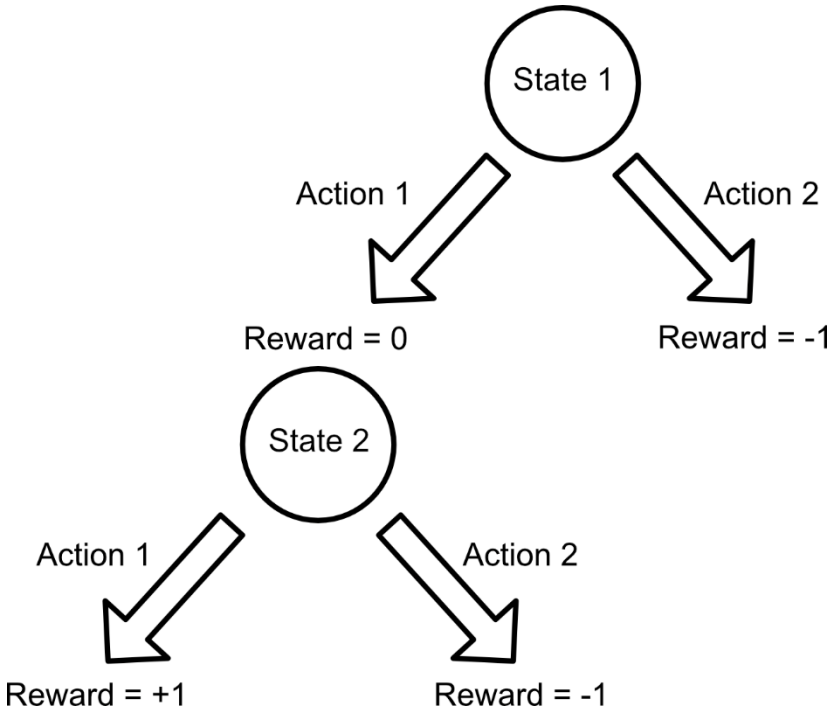
[3] You might ask, why did I choose +1 and -1? The scale of the reward doesn't really matter; it just alters the scale of the agent's parameters. (For example, if I multiplied the rewards by 10, I could just divide the agent's learning rate by 10, and it would learn at an identical speed.)

Let’s make the maze more complicated. What if we add more decision points?



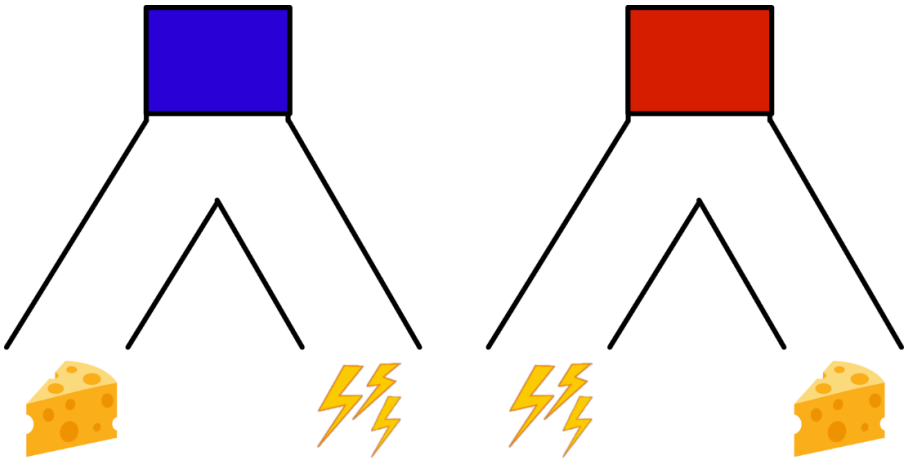
Tricked you! It’s a mouse.

Now, if the rat goes left at the first decision, it has to make a second decision. This is how we might translate the maze into an RL task:



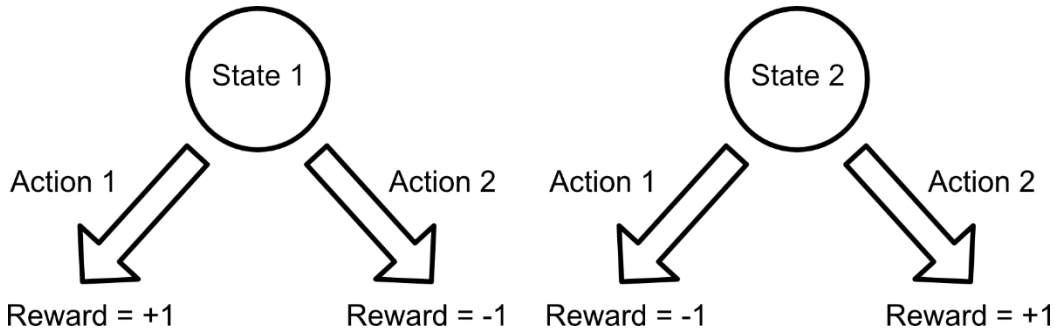
We added a second decision point, so we add another state. Make sense?[4] Notice that the agent technically gets an immediate reward after choosing to go left in state 1 – the reward is just zero. A reward function specifies a reward after every choice.

One last key point. Let's return to our original one-step maze. Now, imagine that there are two starting rooms: a blue room and a red room. In the blue room, going left leads to cheese and going right leads to shock. But in the red room, the contingencies are reversed: going right leads to cheese and going left leads to shock.



Now it's an invisible mouse.

On any given run of the maze, the rat is either placed in the blue room or the red room. How many decision points are there now? In some sense, the new maze only has one decision point – the rat makes a single choice each time it goes to the maze. But in a different sense, it has two decision points – the rat has to know what to do *if it's in the blue room*, and what to do *if it's in the red room*. It is this latter sense that you should base your states on.[5] Here's the diagram:



[4] The state labeling is totally arbitrary. I called them state 1 and 2, but I could just as easily call them Arkansas and Nebraska. (See what I did there?) Labeling the states is only important so that the agent can distinguish them.

[5] Put differently, you should have a different state for each set of information that the agent could have available to it at the decision point. Otherwise, it couldn't learn to take a different action in the two different rooms.

These diagrams are incredibly useful because they schematically capture everything you need to know about your task: what the states are, what actions are available in each state, what rewards agents get after those actions, and what state the agent transitions to next. I highly recommend drawing a similar diagram for any RL task you're trying to program.[6]

The agent. Relative to programming the task, programming the agent will be easy. There are only two things you have to program the agent to do. Your agent has to choose actions, based on how much total reward it thinks those actions will lead to (from the current state); and your agent has to update that estimate based on the rewards it experiences. There are many ways that you can do both of these things; read the Sutton & Barto textbook to get a sense of them.

In this tutorial, we will use Q-learning to update estimates of expected total reward, and a softmax decision function to (probabilistically) choose actions based on those estimates. The estimates will be stored in a "Q matrix". Each element $Q(s,a)$ of the Q matrix represents the agent's current estimate of the expected total reward after choosing action a in state s. To choose actions with softmax, you first calculate the probabilities of choosing each action a in your current state s, using this formula:

$$Prob(choosing\ action\ a\ in\ state\ s) = \frac{e^{\beta * Q(s,a)}}{\sum_{all\ possible\ actions\ a_i\ in\ state\ s} e^{\beta * Q(s,a_i)}}$$

(β is a free parameter called an "inverse temperature".) Then, you randomly sample an action using those probabilities. (To do this, check out MATLAB's *randsample* function.)

After making a choice, receiving a reward r, and transitioning to a new state s', you update your Q matrix with this formula:

$$Q(s,a) = Q(s,a) + \alpha(r + \max_{all\ possible\ actions\ a_i\ in\ state\ s'} Q(s',a_i) - Q(s,a))$$

α is a free parameter called a "learning rate". Again, if this is your first time seeing these formulas, go back and read some Sutton & Barto before attempting the Missions!

---

[6] There's one other tricky thing that I forgot to mention. How do you deal with the end of the maze? There's really two ways to think about it. You could conceptualize it as: "There's a terminal action that doesn't transition to any future state. After taking a terminal action (and getting the reward), the round ends and I start at the beginning again." That's how I've drawn the diagrams here.

But there's another possibility. You could also conceptualize it as: "There's a terminal *state* with no possible actions. After reaching a terminal state, the round ends." In that case, the diagram would have included extra states at the bottom, with no actions leading out of them. Either approach is fine; just make sure you're thinking about it. (And make sure you're thinking about how to deal with the "max" term in the last Q matrix update!)

Putting it all together. I don't expect you to have deeply understood all of this. You won't until you try it. Rather, this introduction is meant to be a reference to which you can return when you're having difficulty with the Missions.

So go start the Missions! To help you get off the ground, here is some pseudocode that puts together everything I've discussed. Your code should end up looking something like it.

```
Q_matrix = zeros(number_of_states, number_of_actions)

for current_round = 1:number_of_rounds

    what state is the agent in?
    what actions can the agent take?

    agent chooses an action (based on its Q matrix)

    what state does the agent transition to (if any)?
    what reward does the agent get?

    agent updates its Q matrix

    if the agent has transitioned to a new state,
            repeat this process at the new state;
    otherwise, if we've reached the end of the task,
            return to the starting state and go again.

end
```

Pseudocode for your simulation.