Hello!

The purpose of this assignment is to get you introduced to some basic concepts when it comes to reinforcement learning. However, this will not cover everything (which should be obvious)

What we will cover

1. What is reinforcement learning at a high level?
2. What are states, actions, rewards, and next states in a RL context and how do they relate?
3. How does RL work in a discrete environment?
4. How does RL have to be modified for a continuous environment?
5. How does changing rewards impact learning in RL?
6. What are some rules of thumb for RL that I should know?
7. What are some general ideas that improve RL performance?

What we will not cover

1. How do we go from loss calculation to updated weights?
2. How can I use options in an RL context?
3. How do I implement the state of the art in RL?
4. How does Pytorch do its insane magic?

If you want answers to the second set of questions, ROB 537 answers the first question well, and will give you the background to look into the other questions in either papers or blogs online.

1. But, what is RL?

First, a quick explanation of what reinforcement learning is. On a high level, we have an agent and an environment. The agent can affect the environment in some way and has some sensing capability. The goal is to get some desired behavior from the agent. The way we do this is by having the agent take some action (a) based on the current state (s), and then giving it a reward based on its performance (r). Based on the magnitude of the reward, it will update its policy to be more or less likely to take that action in the future.

To better illustrate this, lets look at a simple problem. Below we have an agent (a red dot), a world, and a goal (a green dot). The agent knows where it is and can move left or right by one dot. Obviously this is a trivial problem, and you could make a solution to this without RL in 5 seconds, but lets treat this as if it was a difficult RL problem.

In this world, the state is the position of the red dot on the line and the action is either left by one or right by one. Lets start by initializing an agent with no knowledge, so in any position it is likely to move left as it is to move right. Since this is a simple problem, we can express that as a table.

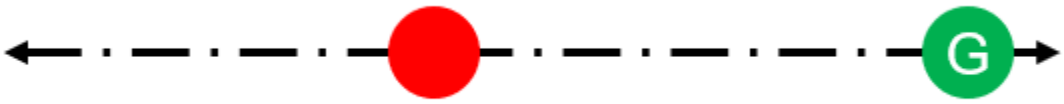
Position	1	2	3	4	5	6	7	8
Q(s,a=L)	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
Q(s,a=R)	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5

There are 8 positions on the line (not counting the goal), so we can number them 1-8 and express the Q value of both actions at that position. This form of learning where we learn the value of state-action pairs is called Q-learning (which is why we call it the Q value). For this example, our agent will pick an action with probability proportional to the Q value at that state divided by the sum of the Q values at that state. For example, at position 4 we go left with probability:

$P(L|s=4) = Q(4,a=L)/(Q(4,a=L)+Q(4,a=R)) = 0.5$

It is important to note that this table is our agent. When we need an action, we pick one by randomly selecting either left or right based on these probabilities. When we train the agent, we don't change this fundamental fact, we just change the values in this table to make one option more or less likely.

In order to train our agent, we will need a reward strategy. Since we know that we want the dot to go right, anytime the agent moves right we will encourage that action (in this example we will add 0.5 to the right action). Anytime the agent moves left we will do the opposite and decrease 0.5 to the left action. This is extreme, so you wouldn't have a reward this large in practice, but it makes the example easier. For simplicity, lets say that the Q value can't get bigger than 1 or smaller than 0. Lets say the dot moves left first because we got unlucky.



The agent gets a negative reward, so it will be less likely to take that action in the future. Importantly, this only affects the actions in position 5 (where it was before moving). In a more complex environment, we might want to go left sometimes, so we can't say anything other than "taking the left action in position 5 is a bad idea"

Position	1	2	3	4	5	6	7	8
Q(a=L)	0.5	0.5	0.5	0.5	0	0.5	0.5	0.5
Q(a=R)	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5

Lets say it takes another action and this time it goes right. We reward that action, which updates the table.

Position	1	2	3	4	5	6	7	8
Q(a=L)	0.5	0.5	0.5	0.5	0	0.5	0.5	0.5
Q(a=R)	0.5	0.5	0.5	1	1	0.5	0.5	0.5

In this instance, you can see that position 5 has a 100% chance of moving right and position 4 has a 2/3 chance to move right. In this domain, our reward function punishes the wrong choice more than the right choice. Lets say we finish the example with the agent always going right. Now we reach the goal and if we reset the environment to the same position, we have the following table.

Position	1	2	3	4	5	6	7	8
Q(a=L)	0.5	0.5	0.5	0	0	0.5	0.5	0.5
Q(a=R)	0.5	0.5	0.5	1	1	1	1	1

As you can see, it is more likely to go right at positions 6,7, and 8, but it might still try going left. If it does that, the Q value of left actions at that space would go to 0. Suppose we ran the environment 100 times starting from position 4 every time. What would happen?

Eventually, positions 6,7 and 8 would all end up with $Q(s,a=L) = 0$, so the agent would always go right from that space. We would end up with this table.

Position	1	2	3	4	5	6	7	8
Q(a=L)	0.5	0.5	0.5	0	0	0	0	0
Q(a=R)	0.5	0.5	0.5	1	1	1	1	1

This doesn't mean that the dot will not move left if it ends up in a position it hasn't visited before! If we reset the environment and place it at position 2, it still has a 50-50 chance of moving left because it doesn't know which action is better *at that state*.

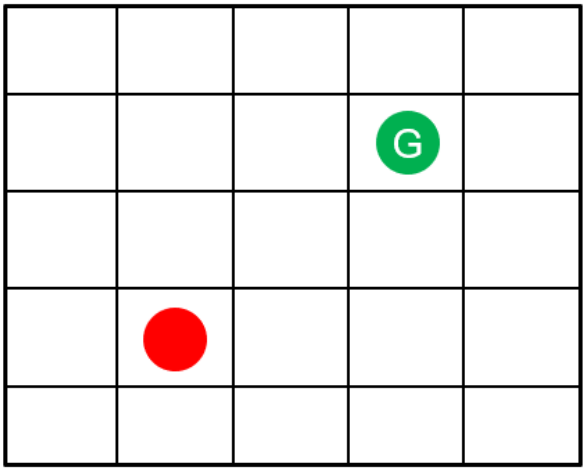
What does this trivial example teach us?

1. An RL agent reads in the state information and decides which action to take based on that state information AND values it has saved internally that represent what it "knows"
2. An RL agent updates its internal values by taking actions and receiving either positive or negative reinforcement for that action at that state
3. RL agents only "learn" the correct behaviors for the portion of the state space that they have gotten to explore

2. Ok, I get what RL is and some basic assumptions. What if we make it more challenging?

Ok, we understand the basics and got our agent to work on a toy example in 1 dimension (Or you skipped to this point because you already understood the basics). Now we can get a little more difficult with our problem.

Lets upgrade our dot example to 2D. Now we have a 5x5 grid. The goal is in the top right and agent gets initialized at some random position. Now we can move in four different directions, but we can't just reward the actions right and up because that could result in the agent missing the goal. Lets formulate this as a RL problem first, then revisit this reward problem. The state is [x,y] now, rather than just one number. The action is still only one dimensional, but it has a list of 4 options now [Left, Up, Right, Down]. From looking at the grid we have 24 possible states (Not counting the goal state) with 4 actions each (assume that taking an action that pushes against a wall just results in no state change). That means if we were to make a table of probabilities, like we did for the previous problem, we would have 96 values. Before we added this dimension, we only had 16 values to keep track of. Now, even though we shrunk the size of each dimension from 8 to 5, we still dramatically increased the number of values we need to learn. This means we need more data to tune all these values, which means running the simulator for longer. If taking an action tells us with 100% certainty if we should ever take that action again (which in practice, it doesn't) we still have to take close to 10x the number of actions that we did for the previous example.

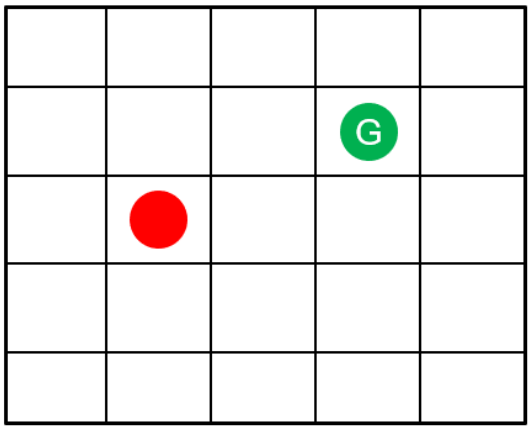


5	0	0	0	0	0
4	0	0	0	G	0
3	0	0	0	0	0
2	0	0	0	0	0
1	0	0	0	0	0
State Values	1	2	3	4	5

Since 96 is a lot of values to show on a table, what if we tried a different method of picking an action? Rather than grading the agent based on the action it takes at a specific state, we grade it on the state that it ends up at. This way, if we end up at the center of the grid, it doesn't matter which way we got there, we get the same reward. This essentially means that we divide the number of values we need to store by 4. Also, this type of learning is called, "Value Learning" because we weren't very original back in the day.

Once again, lets initialize a table with the values all the same. We will say that our agent takes the action that leads to the largest next state value. In the event of multiple next state values with the same reward, we take a random one.

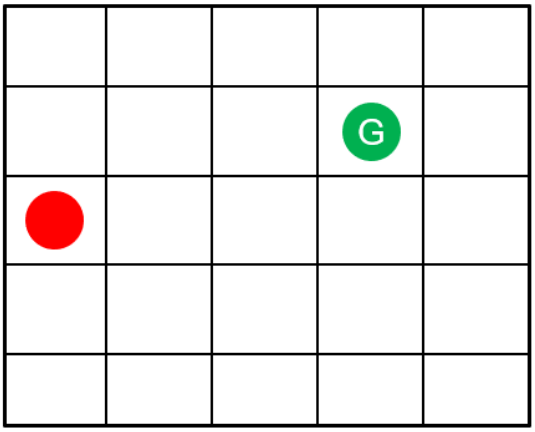
We want to encourage actions that take us closer to the goal, so we could give a reward based on how far we are from the goal after taking an action. For example, currently we are 4 steps away from the goal. If we go up or right, we are 3 steps from the goal, but if we go down or left we are 5 states from the goal. Since we want to maximize the reward, lets make our reward for any state equal to the negative distance to the goal or 10 if it reaches the goal. Lets try an example. The agent moves up at random, and it receives a reward signal of -3. For simplicity, lets say it copies that reward to the value table.



5	0	0	0	0	0
4	0	0	0	G	0
3	0	0	0	0	0
2	0	-3	0	0	0
1	0	0	0	0	0
State Values	1	2	3	4	5

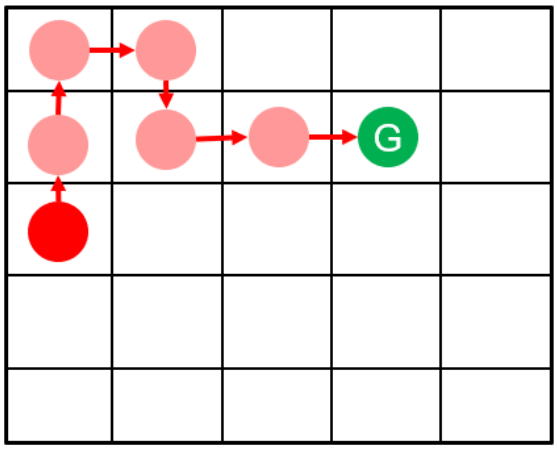
There are a few important things to note here. While the reward is based on the distance to the goal *after* taking the action, it is assigned to the state *before* taking the action. We assign the reward to the before state because we know that we can take an action at that state that leads us to that reward in the future, but the reward might not be solely dependent on the state position. (Essentially we are doing this because that is more stable for more complex problems. This problem in particular has a trivial solution so it seems like overkill)

Also, note that because the reward is negative, our agent thinks that the start state is less valuable than the other states. This will change when it visits the other states, but for now it will avoid this state in favor of exploring the other unseen states. Lets say it goes left this time, getting a reward of -4.



5	0	0	0	0	0
4	0	0	0	G	0
3	0	-4	0	0	0
2	0	-3	0	0	0
1	0	0	0	0	0
State Values	1	2	3	4	5

Now it thinks that this state is even worse than the previous one, but it still thinks the other unseen states are worth pursuing. This is not true, and it will learn that eventually, but right now it is wrong. This is an important point to consider. If we had gone right from that position instead, it would look like a better location than our start state based on our reward function. This is something to keep in mind with value learning. You can fill in the table faster, but you still haven't tried every action at every state faster. This means you could undervalue states because you happened to take a bad action the first time and you end up avoiding that location in the future. For simplicity, lets finish this trial. Lets say it goes up, up, right, down, right, right to end up at the goal.



5	-3	-2	0	0	0
4	-4	-1	10	G	0
3	-3	-4	0	0	0
2	0	-3	0	0	0
1	0	0	0	0	0
State Values	1	2	3	4	5

Now that we have some initial values, we can see that if we end up at the position (1,4), we have a straight shot to the goal. If we end up anywhere else though, we will end up exploring a bit first. Also, not every state is perfectly valued since we took good actions sometime and bad actions other times. Eventually however, assuming we are willing to explore a little bit and take what we think are sub-optimal actions, we will end up exploring all states and actions. When this happens, we will know the value of all states and we can plan perfect paths to the goal.

What did this slightly more complex example teach us?

1. We can value the state alone rather than state action pairs and still learn well.
2. While this reduces the size of our agent, it doesn't decrease the amount of data we need to explore the space.

Now, lets say that we want to increase the resolution of this model. Lets say in this first model, each square is 1x1 meter. Now unfortunately, we want to be more accurate and be able to move the robot to a 1x1 cm square. This means our little 5x5 meter room goes from 24 possible state locations to 249,999 possible state locations. Now we would need to run a prohibitively large number of trials before we could fill in a table of state values. So, what if we instead found a function that could map the state to the value? That way rather than memorizing the value of a single state using a datapoint, we tweaked the function to make the curve look better. This way learning from one state also influences how the function responds to other nearby states. In our next lesson we can look at actor-critic, a way of handling continuous domains.