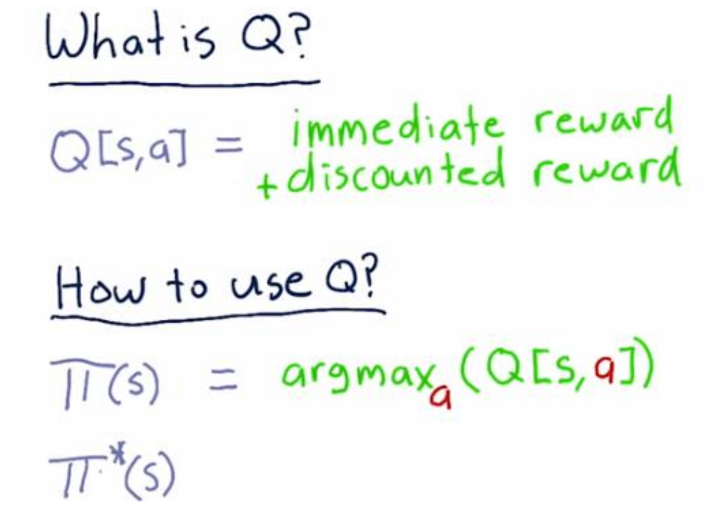
03-06 Q-Learning

Compiled by Shipra De, Fall 2016

Overview

- This lesson is about Q-learning. Recall that Q-learning is a model-free approach, meaning that it does not know about or use models of the transitions T or the rewards R.
- Instead, Q-learning builds a table of utility values as the agent interacts with the world. These Q-values can be used at each step to select the best action based on what it has learned so far.
- The fantastic thing about Q-learning is that it is guaranteed to provide an optimal policy. There is a hitch, however, that we'll cover later.

What is Q?



What is Q?

$Q[s,a] =$ immediate reward + discounted reward

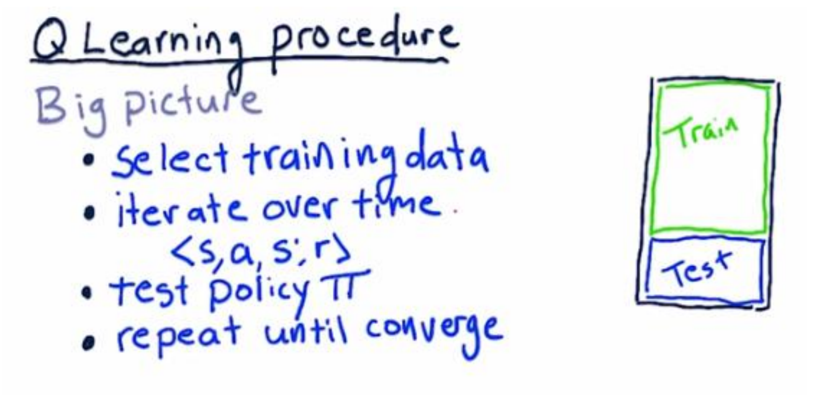
How to use Q?

$\pi(s) = \text{argmax}_a(Q[s,a])$

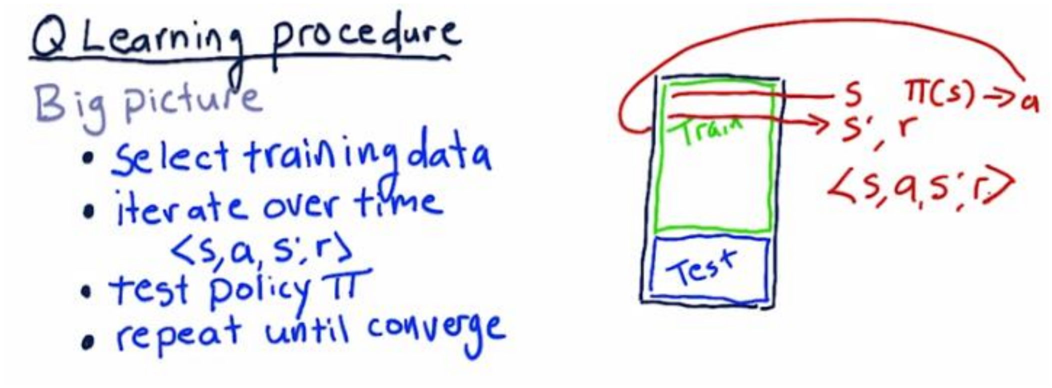
$\pi^*(s)$

- Q learning is named after the Q function. What is that? Well, let's dig in and find out. Q can be written as a function, so we might have parentheses around s and a, or you can think of it as a table.
- So in this class we're going to view Q as a table. And it's got two dimensions, s and a. So s is the state that we're looking at, and a is the action we might take. Q represents the value of taking action a in state s, and there's two components to that. The two components are the immediate reward that you get for taking action a in state s, plus the discounted reward.
- And what that is about, what the discounted reward is about, is the reward you get for future actions. So an important thing to know is that Q is not greedy, in the sense that it just represents the reward you get for acting now. It also represents the reward you get for acting in the future.
- Let's suppose we have Q already created for us. We have this table. How can we use it to figure out what to do? So what we do in any particular state is the policy. And we represent the policy with pi.
- So pi of s means, what is the action we take when we are in state s, or what is the policy for state s? And we take advantage of our Q table to figure that out. Here's how it works.
- We're in state s and we want to find out which action is the best. Well, all we need to do is look across all the potential actions and find out which value of (Q [s,a]) is maximized. So we don't change s, we just step through each value of a, and the one that is the largest is the action we should take.
- And the mathematical way to represent this is to use the function argmax, so argmax of a of this function. So what that does is it finds the a that maximizes this, and then the answer is a.
- After we run Q learning for long enough, we will eventually converge to the optimal policy. And we represent that with a little star. So the optimal policy is pi star of s. [COUGH] And similarly the optimal Q table is Q star [s, a]. Now this is how to use a Q table if you have it.
- We need now to consider how do we build that Q table in the first place.

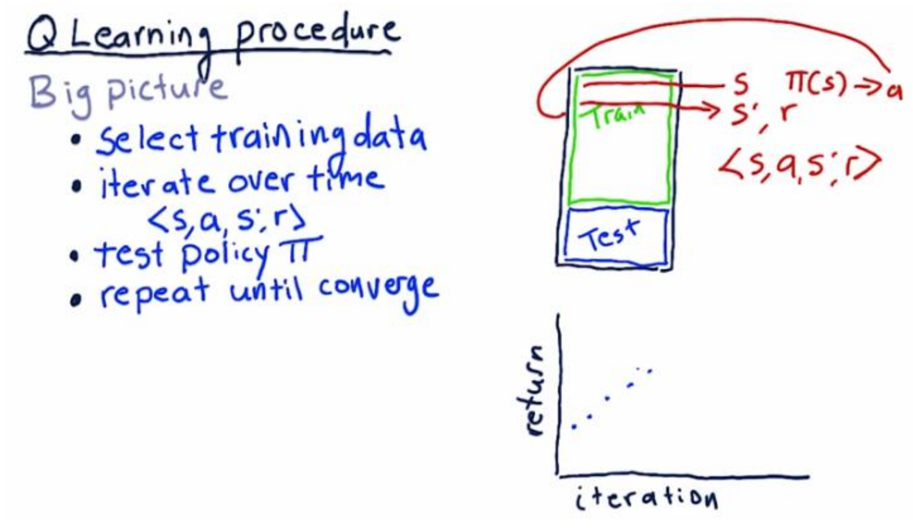
Learning Procedure



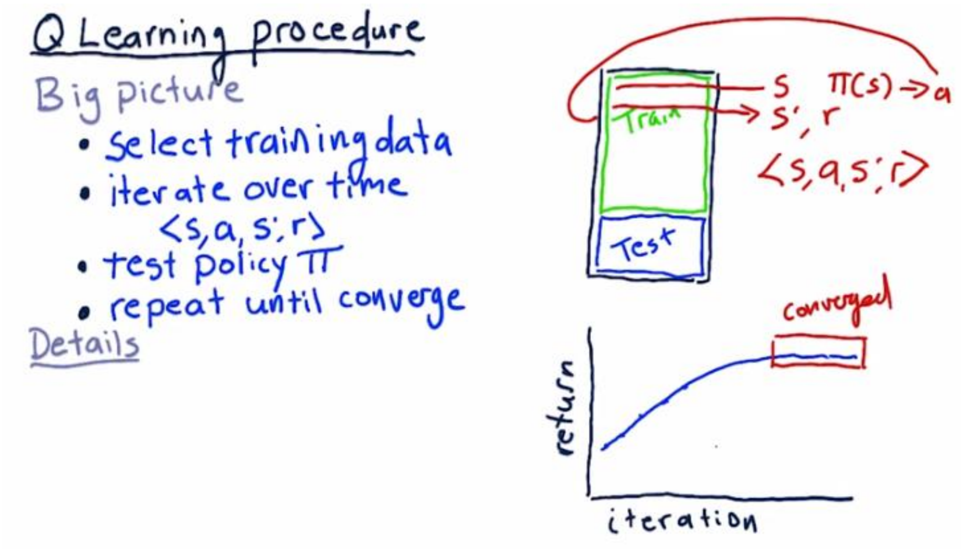
- Here's the big picture at a high level of how we train a q learner. We have our data here and we select which data we want to train on, of course this data in the case of the stock market is time series. And so it's arranged from oldest to newest vertically here.



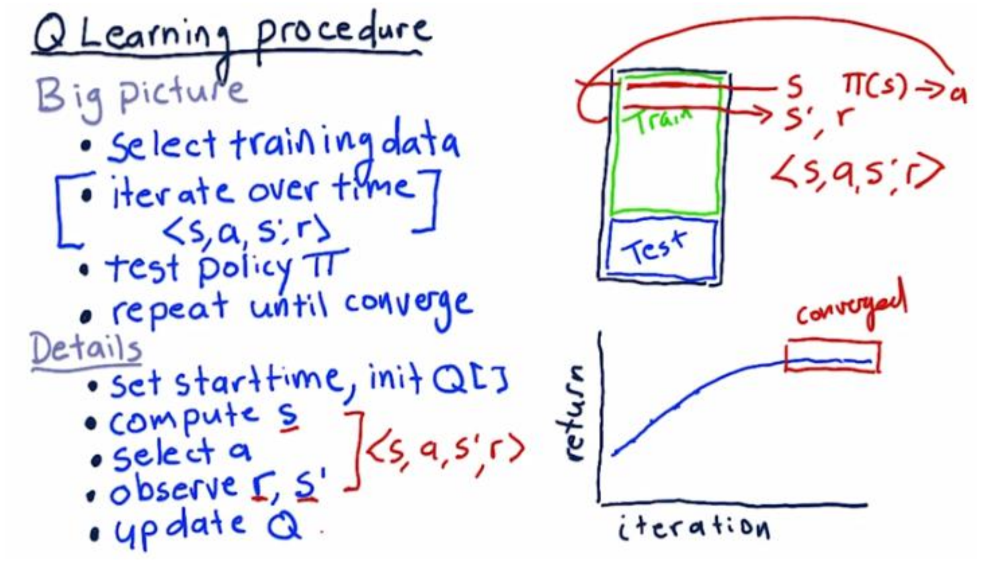
- So we select the day we want to train on. And then we iterate over this data over time. So we evaluate the situation there and for a particular stock that gives us s our state.
- We consult our policy and that gives us an action. So we take that action, plug it into our system here, evaluate the next state, and we get our s prime and our reward.
- So after one iteration here we've got an s, an action, an s prime, and an r. Or an experience tuple, and we use that experience tuple to update our Q table.
- Once we get all the way through the training data, we test our policy and we see how well it performs in a back test.



- If it's converged or it's not getting any better, then we say we're done. If not, we repeat this whole process all the way through the training data.
- So what do we mean by converge? Well, each time we cycle through the data training our Q table and then testing back across that same data, we get some performance. And we expect that each time we complete an iteration here, our performance is going to get better and better.

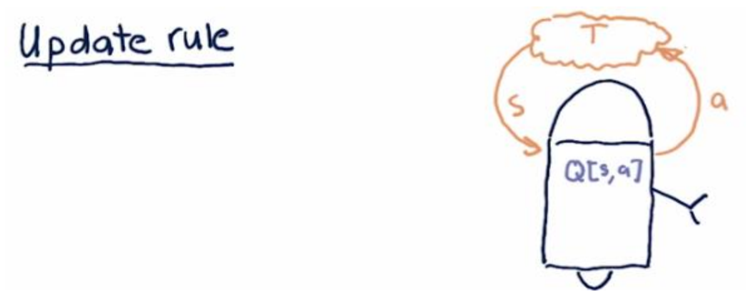


- But after a point it finally stops getting better and it converges. So overall the chart's going to look something like this. Eventually we reach this regime where more iterations doesn't make it better and we call it converged at that point.

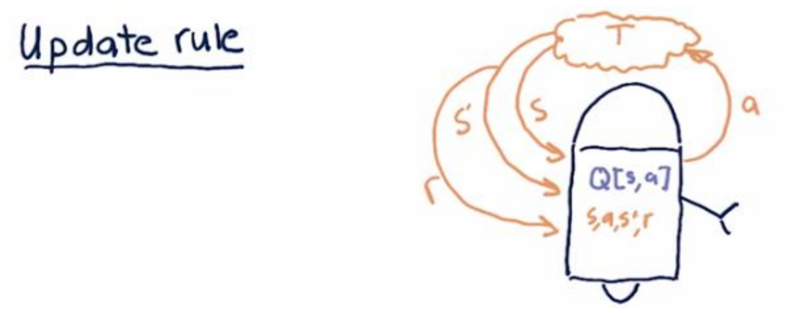


- Let's consider now in more detail what happens here when we're iterating over the data. So here are the details as we iterate over our training data. We start by setting our start time, which is right here at the beginning and we initialize our Q table.
- The usual way to initialize a Q table is with small random numbers, but variations of that are fine.
- Now we're here in time and we observe the features of our stock or stocks and from those build up together our state s. We consult our policy, or in other words we consult Q to find the best action in the current state. That gives us a.
- Then we step forward and we see what reward we get and what's our new state. We now have a complete experience tuple that we can use to update our Q table.
- So we take this information that we just learned and we improve Q based on that information. Then we step to the next point in time and the next point time and the next, next one time and so on. So these are all the details of what happens in this step of the big picture.

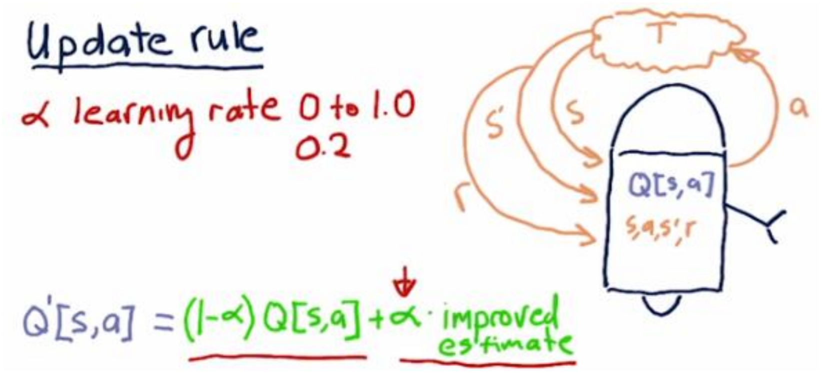
Update Rule



- Consider our robot here that's interacting with the world. The first thing it sees is some state, and as we've talked about already, the thing to do now is to take that state, go look in the Q table and find the action that corresponds to the maximum Q value. And then take that action.

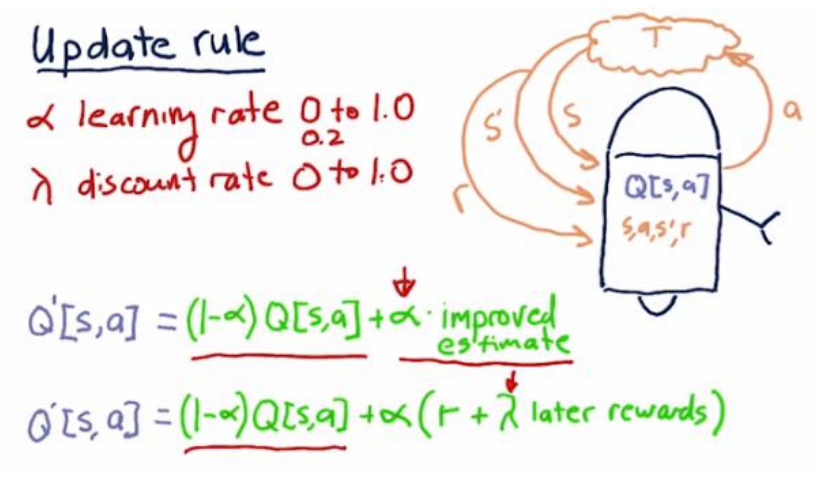


- Once that action is taken, that results in two new things. One is a new state, we call that S prime, and a reward. All that information comes into the robot, and it needs to use that information to update its Q table.
- So as a consequence of this interaction with the world, it's got an s, an a, an s prime, and an r. How does it take that information to improve this Q table?

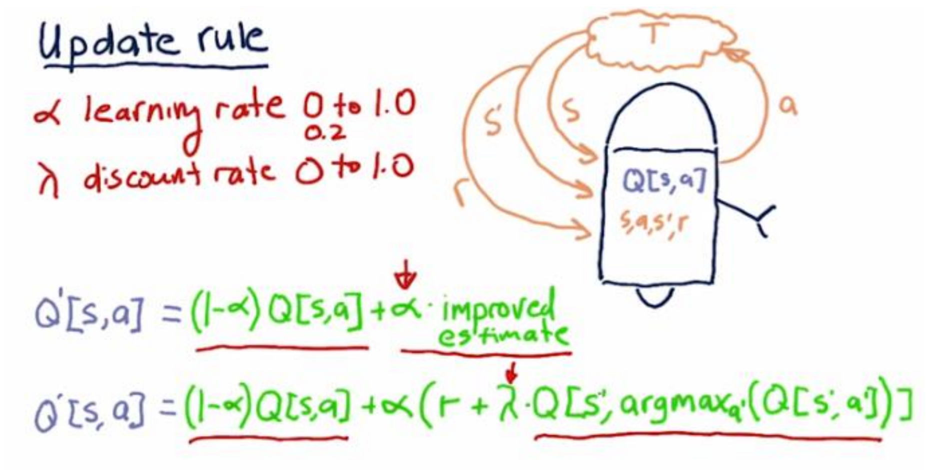


- There are two main parts to the update rule. The first is, what is the old value that we used to have? And that's Q [s, a]. And what is our improved estimate?
- And we want to blend them together. And to combine them, we introduce this new variable, alpha.

- Alpha is the learning rate. Alpha can take on any value from 0 to 1. Usually, we use about 0.2. And what that means is, in our new improved version of Q, which I indicate over here as Q prime, it's a blend of alpha times the improved estimate, plus 1 minus alpha of the old value.
- So larger values of alpha cause us to learn more quickly, lower values of alpha cause the learning to be more slow. So a low value of alpha, for instance, means that in this update rule, the previous value for Q of sa is more strongly preserved.

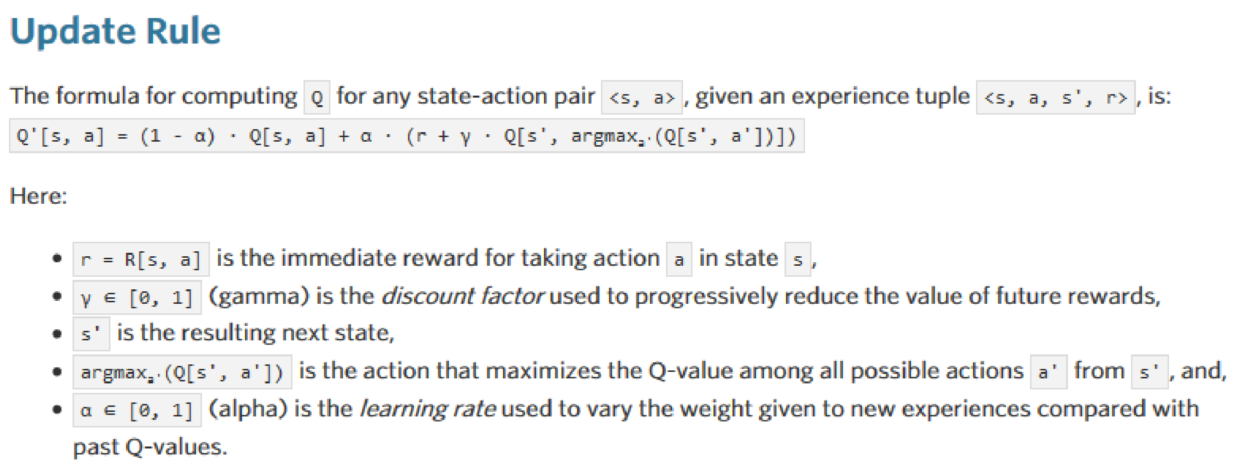

Update rule
α learning rate 0 to 1.0
0.2
λ discount rate 0 to 1.0
T
s'
s
a
r
Q[s,a]
s,a,s',r
Q'[s,a] = (1-α) Q[s,a] + α · improved estimate
Q'[s,a] = (1-α) Q[s,a] + α (r + λ later rewards)

- So stretching it out a little bit more detail. Again we have here, our current value for the Q at s,a, plus alpha times the immediate reward, plus gamma times later rewards. Now we're introducing gamma here, so I promise there's only two new parameters here that you have to worry about.
- What gamma is, is the discount rate. And similar to alpha, gamma usually ranges from 0 to 1. A low value of gamma means that we value later rewards less. Remember the discount rate when we were talking about bonds? Same thing.
- A low value of gamma equates to essentially a high discount rate. The high value of gamma, in other words a gamma near 1, means that we value later rewards very significantly. If we were to use 1.0, that means a reward 20 steps in the future is worth just as much as a reward right now. Now, we have to expand this component here in a little bit more detail.


Update rule
α learning rate 0 to 1.0
0.2
λ discount rate 0 to 1.0
T
s'
s
a
r
Q[s,a]
s,a,s',r
Q'[s,a] = (1-α) Q[s,a] + α · improved estimate
Q'[s,a] = (1-α) Q[s,a] + α (r + λ · Q[s', argmax_a'(Q[s', a'])]

- This next part here is a little bit tricky, but don't worry, we're going to step through it step by step.
- This component represents our future discounted rewards. In other words, we end up in state s prime, and from then on out, we're going to act optimally, or at least, the best that we know how to.
- And the question is, what is the value of those future rewards if we reach state S prime and we act appropriately? Well, it is simply that Q value, but we have to find out what the action is that we would have taken, so that we can reference the Q table properly.
- So, If we're in state s prime, the action that we would take that would maximize our future reward is argmax a prime, so a prime is the next action that we're going to take with regard to Q, s prime, a prime.
- So we're going to find that best a prime, that best action, that maximizes the value when we're in that state. So this collapses just to a prime, and then we look up the Q table value for Q s prime, a prime. So that allows us to bring it all together now.
- So our update rule is the following. Our new Q value in state s, action a is that old value multiplied by 1 minus alpha. So depending on how large alpha is, we valued that old value more or less, plus alpha times our new best estimate. And our new best estimate is, again, our immediate reward, plus the discounted reward for all of our future actions.
- And that's it. This is the equation you need to know to implement Q learning.

Update Rule – Notes



Update Rule

The formula for computing Q for any state-action pair <s, a>, given an experience tuple <s, a, s', r>, is:

$Q'[s, a] = (1 - \alpha) \cdot Q[s, a] + \alpha \cdot (r + \gamma \cdot Q[s', \text{argmax}_{a'}(Q[s', a'])])$

Here:

- $r = R[s, a]$ is the immediate reward for taking action a in state s,
- $\gamma \in [0, 1]$ (gamma) is the *discount factor* used to progressively reduce the value of future rewards,
- s' is the resulting next state,
- $\text{argmax}_{a'}(Q[s', a'])$ is the action that maximizes the Q-value among all possible actions a' from s', and,
- $\alpha \in [0, 1]$ (alpha) is the *learning rate* used to vary the weight given to new experiences compared with past Q-values.

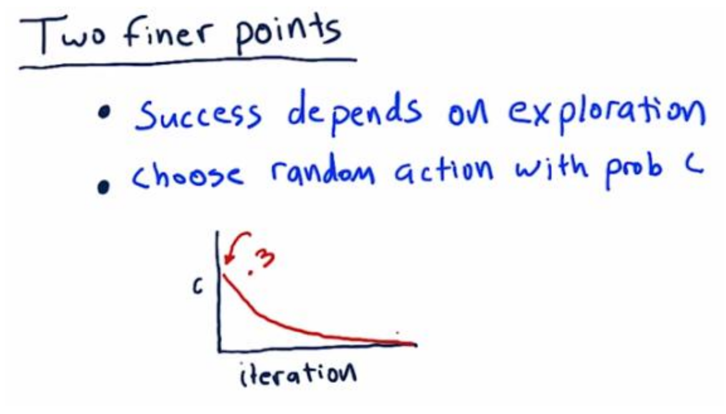
The formula for computing Q for any state-action pair <s, a>, given an experience tuple <s, a, s', r>, is:

$Q'[s, a] = (1 - \alpha) \cdot Q[s, a] + \alpha \cdot (r + \gamma \cdot Q[s', \text{argmax}_{a'}(Q[s', a'])])$

Here:

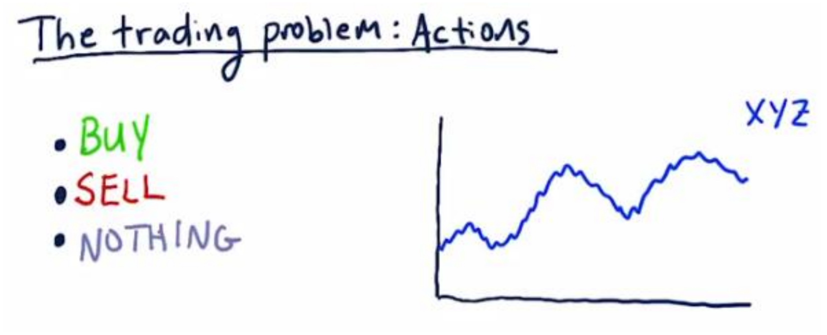
- $r = R[s, a]$ is the immediate reward for taking action a in state s,
- $\gamma \in [0, 1]$ (gamma) is the *discount factor* used to progressively reduce the value of future rewards,
- s' is the resulting next state,
- $\text{argmax}_{a'}(Q[s', a'])$ is the action that maximizes the Q-value among all possible actions a' from s', and,
- $\alpha \in [0, 1]$ (alpha) is the *learning rate* used to vary the weight given to new experiences compared with past Q-values.

Two Finer Points

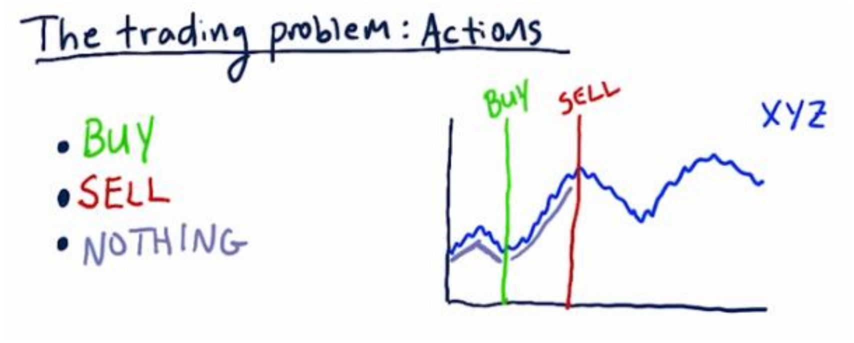

Two finer points
• Success depends on exploration
• Choose random action with prob c
c
.3
iteration

- There are two finer points that I wanted to mention about Q-learning. First is that the success of Q-learning depends to some extent, well, to a large extent on exploration.
- So we need to explore as much of the state and action space as possible. One way to accomplish this is with randomness. And the way we can interject that fairly easily in the step of Q-learning, where we are selecting an action, we flip a coin and randomly decide if we're going to randomly choose an action.
- So that means really two flips of the coin. The first is are we going to choose a random action or are we just going to pick the action with the highest Q value? Then if we decide okay, we're going to choose an action randomly, now we need to flip the coin again to choose which of those actions we're going to select.
- Typical way to implement this is to set this probability at about something at the beginning of learning. And then over each iteration to slowly make it smaller and smaller and smaller until eventually we essentially don't choose random actions at all.
- What we gain here is when we're choosing these random actions fairly frequently is we're forcing the system to explore and try different actions in different states. And it also causes us to arrive at different states that we might not otherwise arrive at if we didn't try those random actions.

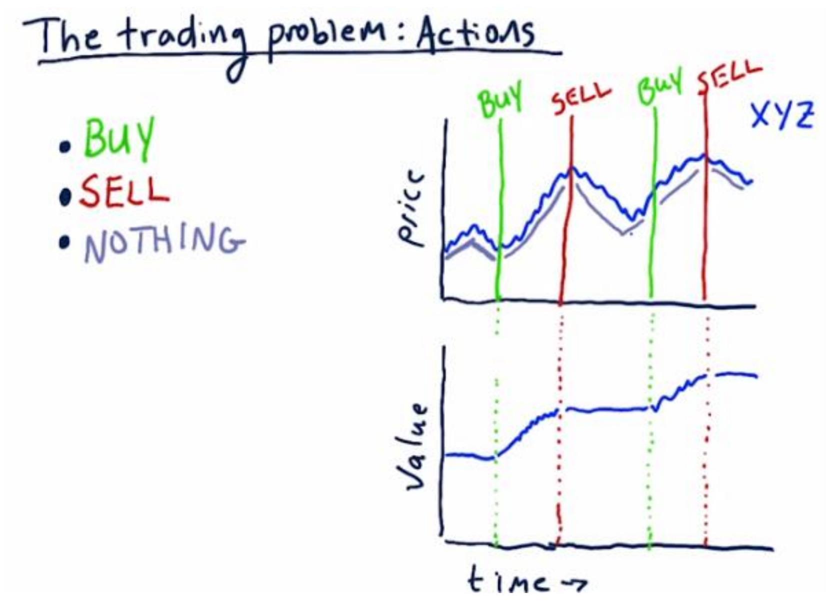
The Trading Problem: Actions



- Okay, now that you have a basic understanding of Q learning, let's consider how we can turn the stock trading problem into a problem that Q learning can solve. To do that we need to define our actions, we need to define our state, and we also need to define our rewards.
- Let's start here first with actions. It's actually pretty easy. We just have three actions, buy, sell or do nothing. So let's consider how that might play out with an actual stock, and let's suppose we've already trained our Q learner what to do.
- Let's see what's likely to happen. So usually what's going to happen most frequently is that we do nothing. There will occasionally be buys and sells, but usually nothing. Well, at least, that's my expectation, we'll have to see when we create these learners what actually happens.
- So we are evaluating the factors of the stock. In other words, we compute, say, several technical indicators. That is our state. We consider that state and we do nothing.

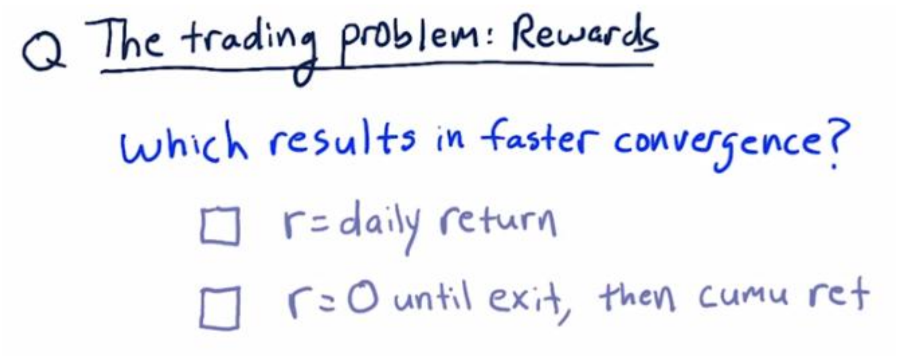


- So let's suppose we do nothing for quite a long period here, but somehow or another, after a little while, boom, something triggers and says we should buy. So we buy the stock here we're holding it.
- We do nothing, nothing, nothing, nothing, nothing, nothing, and boom, our very intelligent Q learner says, hey now is the time to sell. And we continue like that through the rest of our time series.



- Let's consider now what these buys and sells, how they affect our portfolio value. So we start out with whatever we've got in the bank and there's no trading for a while. Suddenly there's a buy, and then we see an increase in our portfolio value until we hit that sell. Then nothing. Again, another buy, and a sell. Then nothing.
- So you see this sort of stepped behavior. Now of course, because I was drawing this and I wanted us to look good, I was choosing opportunistic times to buy and sell. Your real strategy is probably not going to be so good, but ideally this is the kind of thing we'd like to see.

The Trading Problem: Rewards

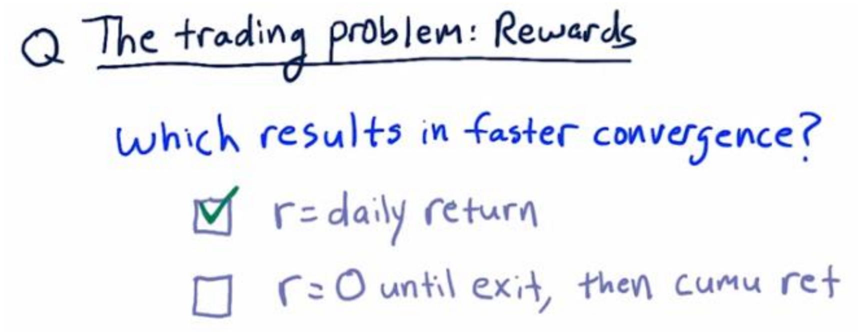

Q The trading problem: Rewards

which results in faster convergence?

☐ r = daily return

☐ r = 0 until exit, then cumu ret

- Now consider rewards for our learner. Of course it makes sense that rewards should relate in some way to the returns of our strategy.
- There are at least two approaches that we can utilize. Short-term rewards in terms of daily returns or long-term rewards that reflect the cumulative return of a trade cycle from a buy to a sell, or for shorting from a sell to a buy.
- So consider these two, daily returns, where each day we get a reward which is equal to the return over the last day, or a reward where our the reward is equal to zero, until we exit the position, then the cumulative return that we gained across that whole trade.
- Which one of these do you think will result in faster conversions?


Q The trading problem: Rewards

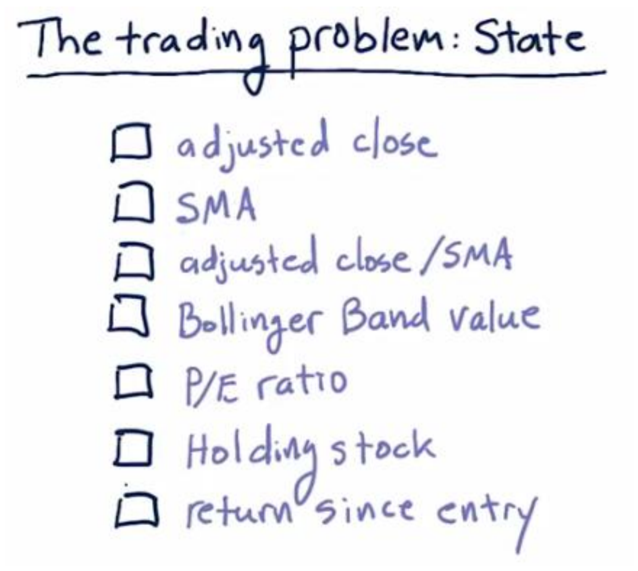
which results in faster convergence?

[✓] r = daily return

☐ r = 0 until exit, then cumu ret

- The correct answer is daily returns. The reason is, if you choose the other one where we get no reward at all until the end of a trade cycle, from a buy to a sell. The learner has to infer from that final reward all the way back, that each action in sequence there must have been accomplished in the right order to get that reward.
- If we reward a little bit each on each day, the learner is able to learn much more quickly because it gets much more frequent rewards. This by the way, is called delayed reward and this is more of an immediate reward.

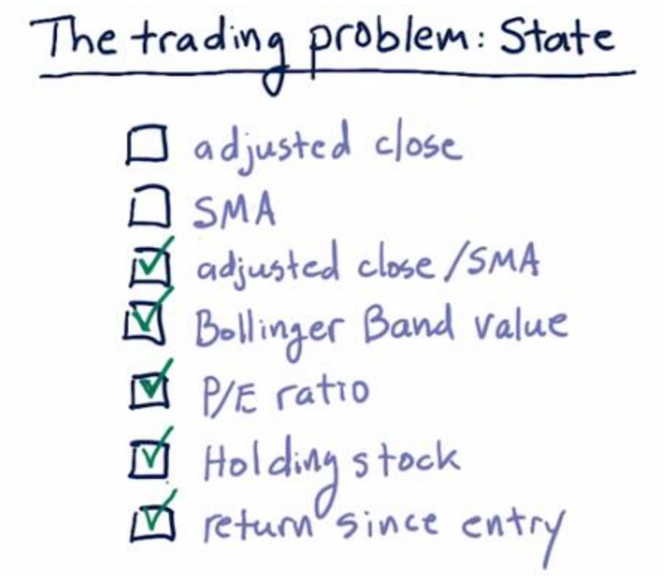
The Trading Problem: State



The trading problem: State

- ☐ adjusted close
- ☐ SMA
- ☐ adjusted close / SMA
- ☐ Bollinger Band value
- ☐ P/E ratio
- ☐ Holding stock
- ☐ return since entry

- Now we come to the last thing we need to solve in order to convert our trading problem into a q learning problem, and that is to define our state. I want you to consider these factors and let me know which ones you think make sense to be in this state.
- Okay, so here are a few factors. Look at each one and consider whether you think they make sense to be in state. And check the ones that you think do make sense.



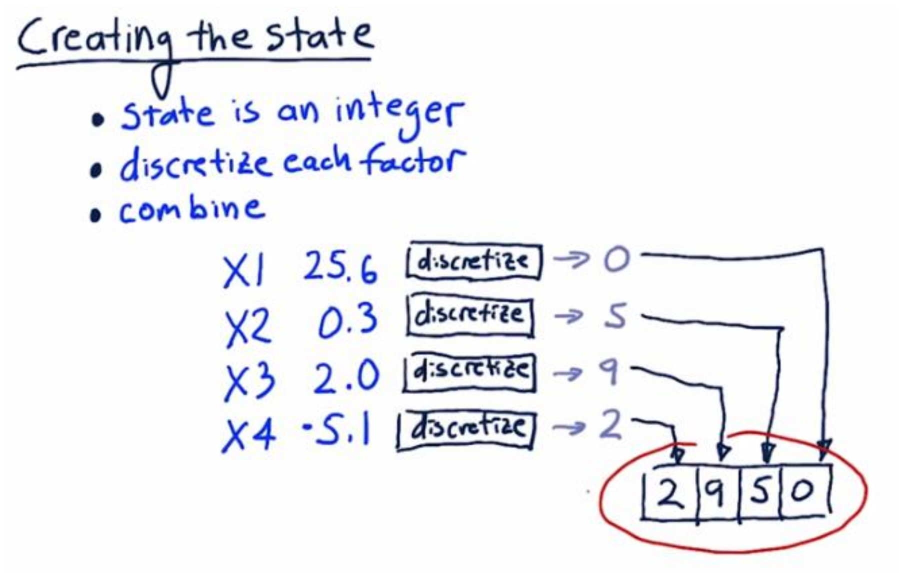
The trading problem: State

- ☐ adjusted close
- ☐ SMA
- ☑ adjusted close / SMA
- ☑ Bollinger Band value
- ☑ P/E ratio
- ☑ Holding stock
- ☑ return since entry

- Adjusted close is not a good factor for learning, because you're not able to generalize over different price regimes for when the stock was low to when it was high. Also, if you're trying to learn a model for several stocks at once and they each hold very different prices, adjusted close doesn't serve well to help you generalize.

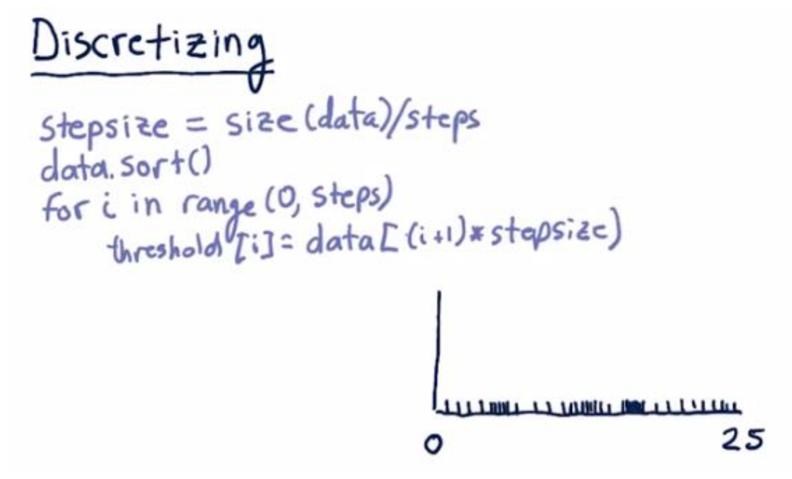
- Same thing is true for simple moving average.
- However if you combine adjusted close and simple moving average together into a ratio that makes a good factor to use in state.
- Bollinger Band value is also good. P/E ratio is good.
- And a new kind of feature that we're considering here with reinforcement learning, is whether we're holding the stock or not. That's an important state to know, in other words, if you're holding the stock it may be advantageous to get rid of it. But if you're not holding it, you might not necessarily want to sell. So this additional feature about what your situation is useful.
- Similarly, return since we entered the position might be useful. In other words, this might help us set exit points for instance, maybe we've made 10% on the stock since we bought it and we should take our winnings while we can.
- So both of these are important and good factors to have an acute learning system.

Creating the State



- So an important consideration in what we're doing here is that our state is a single number. It's an integer. That way we can address it in our Q table. It is certainly the case that some reinforcement learning methods are able to treat the state as a continuous value. But just to get started here, let's use state as an integer so we can work more easily with the data. Now we have to do a little bit of work to get our state to an integer and here is the general way to do it.
- Our first step is to discretize each factor. It's a weird word, I'll explain it in a moment. But it essentially means convert that real number into an integer.
- Next is combine all of those integers together into a single number. We're assuming that we're using a discrete state space and that means more or less that our overall state is going to be this one integer that represents at once all of our factors.
- So consider that we have four factors and each one is a real number. Now we have separately beforehand figured out how to discretize each one of these factors. So we run each of these factors through their individual discretizers and we get an integer.
- Now I've happened to select integers between 0 and 9 but you can have larger ranges, for instance 0 to 20 or 0 to 100 even. It's easy if we just go from 0 to 9 because then we can stack them together into a number. But it's not too hard to think of algorithms will enable you to combine larger ranges of integers together.
- In this case we're just able to stack them one after the other into our overall discretized state.

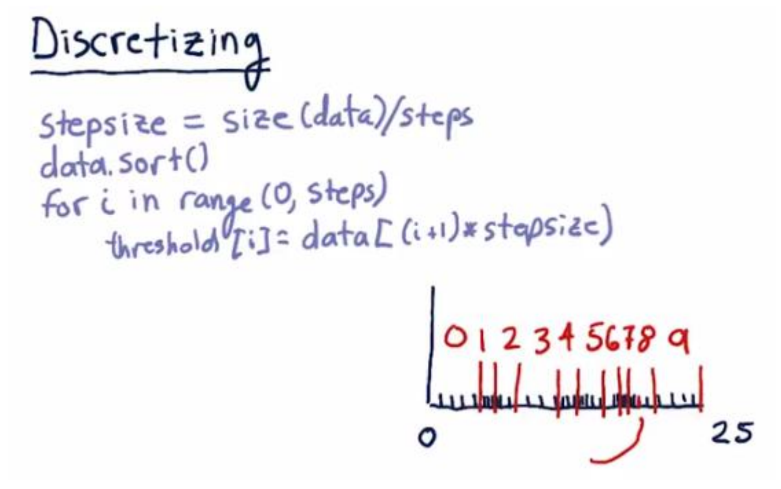
Discretizing



Discretizing

```
stepsize = size(data)/steps
data.sort()
for i in range(0, steps)
    threshold[i] = data[(i+1)*stepsize]
```

- Discretization or discretizing is an important step in this whole process. What we want to do is have a way to convert a real number into an integer across a limited scale. In other words, we might have hundreds of individual values here between 0 and 25 of a real number, and we want to convert that into an integer, say between 0 and 9.
- There's a fairly easy way to do that and I'll show you how right now. So here's what we do. First thing is we determine ahead of time how many steps we're going to have. In other words, how many groups do we want to be able to put the data into? Like I said before, to have an integer between zero and nine, we would use ten in this case.
- So we divide how many data elements we have all together by the number of steps. Then we sort the data, and then the thresholds just end up being the locations for each one of these values.
- So in other words, if we had, say, 100 data elements and we were going to have 10 steps, our step size is 10. So we just find the 10th data element and that's our first threshold, and then our 20th and 30th and so on.

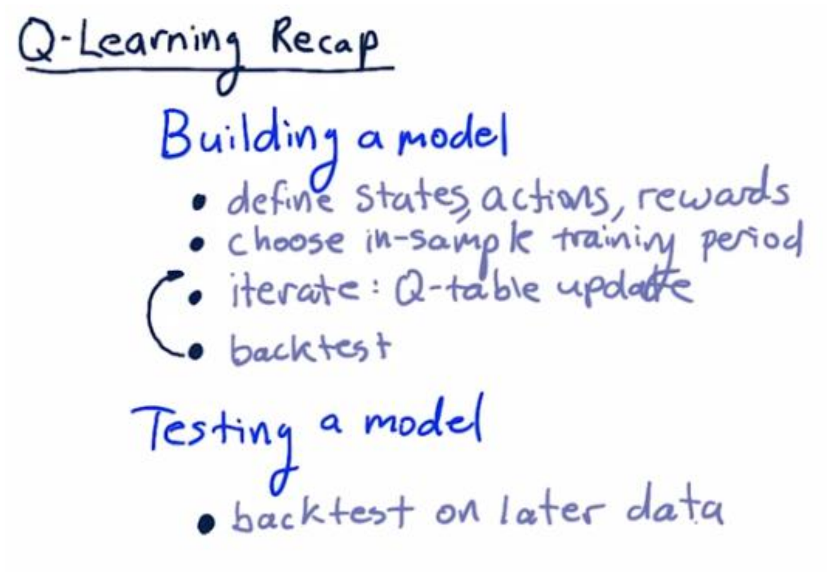


Discretizing

```
stepsize = size(data)/steps
data.sort()
for i in range(0, steps)
    threshold[i] = data[(i+1)*stepsize]
```

- It ends up looking something like this. When the data is sort of sparse, our thresholds are set far apart. When the data is not sparse, these thresholds end up being closer together.
- So in this particular example, our thresholds might end up looking something like this. And when we go to query, when we have a new value and we want to see what is its discretized value, we'll say it was a value here. Boom, it's between those two thresholds, so it would be an 8.

Q-Learning Recap



Q-Learning Recap

Building a model

- define states, actions, rewards
- choose in-sample training period
- iterate: Q-table update
- backtest

Testing a model

- backtest on later data

- Okay, let's recap what we've learned about Q-Learning. First, let's consider what it takes to build a model.
- First step is define states, actions, and rewards. So states are combinations of our features.
- Actions are buy, sell, do nothing.
- And rewards are some type of return.
- Next you choose the training period and you iterate over that training period and update your Q-table on each iteration.
- When you reach the end of that training period you backtest to see how good the model is and you go back and repeat, until the model quits getting better.
- Once it's converged you stop, you've got your model.
- Testing the model is simple you just backtest it on later data.
- Remember we're always training on one set of data and then testing on later data.
- And that's it. I hope you enjoyed the lesson.

Summary

Advantages

- The main advantage of a model-free approach like Q-Learning over model-based techniques is that it can easily be applied to domains where all states and/or transitions are not fully defined.
- As a result, we do not need additional data structures to store transitions `T(s, a, s')` or rewards `R(s, a)`.
- Also, the Q-value for any state-action pair takes into account future rewards. Thus, it encodes both the best possible *value* of a state ($\max_a Q(s, a)$) as well as the best *policy* in terms of the action that should be taken ($\text{argmax}_a Q(s, a)$).

Issues

- The biggest challenge is that the reward (e.g. for buying a stock) often comes in the future - representing that properly requires look-ahead and careful weighting.
- Another problem is that taking random actions (such as trades) just to learn a good strategy is not really feasible (you'll end up losing a lot of money!).
- In the next lesson, we will discuss an algorithm that tries to address this second problem by simulating the effect of actions based on historical data.

Resources

- CS7641 Machine Learning, taught by Charles Isbell and Michael Littman
  - Watch for free on Udacity (mini-course 3, lessons RL 1 - 4)
  - Watch for free on YouTube
  - Or take the course as part of the OMSCS program!
- RL course by David Silver (videos, slides)
- A Painless Q-Learning Tutorial