Lecture 19
Markov Decision Process, Part 2

Alice Gao

July 27, 2021

Contents

1 Learning Goals

By the end of the lecture, you should be able to

- Trace the execution of and implement the value iteration algorithm for solving a Markov decision process.
- Trace the execution of and implement the policy iteration algorithm for solving a Markov decision process.
- Describe the steps of policy iteration algorithm.

2 Value Iteration

Now, we will begin talking about the value iteration algorithm, which we can use to solve for the optimal policy of a Markov decision process. First, we will introduce the Bellman equations, which form a key component of the value iteration algorithm.

2.1 Solving for $V^*(s)$

In the previous lecture, we defined V, the expected long-term total discounted reward of entering a state, and Q, the expected utility of taking an action once we are in a state. V and Q have a very natural relationship and are actually defined recursively in terms of each other:

$$V^*(s) = R(s) + \gamma \max_a Q^*(s,a) \tag{1}$$

$$Q^*(s,a) = \sum_{s'} P(s'|s,a)V^*(s') \tag{2}$$

You've already seen equation 2. As for equation 1, we can measure V as the immediate reward of entering s plus the discounted reward we expect from the best action taken afterward.

Our goal is to derive an algorithm to solve for the V-values, so let's combine these equations to eliminate the Q's. What we get are the Bellman equations (a system, not just one equation):

$$V^*(s) = R(s) + \gamma \max_a \sum_{s'} P(s'|s,a)V^*(s') \tag{3}$$

$V^*(s)$ are the unique solutions to the Bellman equations.

Problem: Write down the Bellman equation for $V^*(s_{11})$.

Solution:

$$V^*(s_{11}) = -0.04 + \gamma \max[0.8 * V^*(s_{12}) + 0.1 * V^*(s_{21}) + 0.1 * V^*(s_{11}),$$
$$0.9 * V^*(s_{11}) + 0.1 * V^*(s_{12}),$$
$$0.9 * V^*(s_{11}) + 0.1 * V^*(s_{21}),$$
$$0.8 * V^*(s_{21}) + 0.1 * V^*(s_{12}) + 0.1 * V^*(s_{11})]$$

The four arguments of the max[ ] function are the expected utilities of trying to go right, up, left, and down, in order.

For example, when we try to go right, we'll move right with probability 0.8, but we may also move down with probability 0.1 or try to move up and bump into the wall with probability 0.1.

In the up and left expressions, the $V^*(s_{11})$ terms are combined since there are two directions which leave us in s_{11}. For instance, if we try to go up, we may bump into the top wall with probability 0.8 or bump into the left wall with probability 0.1.

The point of this example is to make the Bellman equations more concrete for you, rather than just being symbols. You can also see we'll end up plugging in many values to these equations. Here, specifically, we have three variables: $V^*(s_{11})$, $V^*(s_{12})$, and $V^*(s_{21})$.

In the grid world example, we have nine non-goal states. Therefore, we'll have nine Bellman equations in nine variables. Given this, the solution sounds quite simple: we can take the nine equations and solve for the nine unknowns; once we have the V-values, we can solve for the optimal policy.

But as computer scientists, we not only care about being able to solve a problem, we also care about being able to solve a problem efficiently.

Problem: Can we solve the system of Bellman equations efficiently?

(A) Yes

(B) No

(C) I don't know

As an example, here is the Bellman equation for $V^*(s_{11})$:

$$V^*(s_{11}) = -0.04 + \gamma \max[0.8 * V^*(s_{12}) + 0.1 * V^*(s_{21}) + 0.1 * V^*(s_{11}),$$
$$0.9 * V^*(s_{11}) + 0.1 * V^*(s_{12}),$$
$$0.9 * V^*(s_{11}) + 0.1 * V^*(s_{21}),$$
$$0.8 * V^*(s_{21}) + 0.1 * V^*(s_{12}) + 0.1 * V^*(s_{11})]$$

This is a tricky question to answer. As a hint, first try to determine whether the equations are linear or non-linear. Then, try to recall algorithms or techniques you know for solving linear and/or non-linear systems of equations.

Solution: Because of the max[ ] in the equations, the system is non-linear.

Hopefully, you have taken some courses in linear algebra and/or linear programming and know there are many techniques for solving linear systems.

However, our system is non-linear, and it turns out there are no efficient techniques for solving a non-linear system in general.

Given this, the correct answer is (B).

What can we do instead? We'll solve this system of equations by approximation or numerical methods. Specifically, we will solve it iteratively. This leads to the value iteration algorithm.

2.2 Solving for $V^*(s)$ iteratively

Recall the Bellman equations:

$$V^*(s) = R(s) + \gamma \max_a \sum_{s'} P(s'|s,a)V^*(s').$$

The value iteration algorithm is as given:

Let $V_i(s)$ be the values for the i^{th} iteration.

1. Start with arbitrary initial values for $V_0(s)$; zeroes are a good choice.

2. At the i^{th} iteration, compute $V_{i+1}(s)$ as follows:

$$V_{i+1}(s) \leftarrow R(s) + \gamma \max_a \sum_{s'} P(s'|s,a) V_i(s').$$

   Note that the right-hand side will always use values from the previous iteration.

3. Terminate when $\max_s |V_i(s) - V_{i+1}(s)|$ is small enough.

If we apply the Bellman update infinitely often, the $V_i(s)$'s are guaranteed to converge to the optimal values.

2.3 Applying value iteration

Let's apply the value iteration algorithm to our grid world. Assume that

- the discount factor is $\gamma = 1$, and
- $R(s) = -0.04, \forall s \neq s_{24}, s \neq s_{34}$.

Start with $V_0(s) = 0, \forall s \neq s_{23}, s \neq s_{34}$.

Problem: Where is $V_1(s_{23})$?

(A) $(-\infty, 0)$ (B) $[0, 0.25)$ (C) $[0.25, 0.5)$
(D) $[0.5, 0.75)$ (E) $[0.75, 1]$

$V_0(s)$:

	1	2	3	4
1	0	0	0	0
2	0	X	0	-1
3	0	0	0	$+1$

Solution: The calculation is as follows:

$$\begin{aligned} V_1(s_{23}) &= -0.04 + 1 \cdot \max[0.8 * 0 + 0.1 * (-1) + 0.1 * 0, && \text{(up)} \\ &\qquad 0.8 * (-1) + 0.1 * 0 + 0.1 * 0, && \text{(right)} \\ &\qquad 0.8 * 0 + 0.1 * (-1) + 0.1 * 0, && \text{(down)} \\ &\qquad 0.8 * 0 + 0.1 * 0 + 0.1 * 0] && \text{(left)} \\ &= -0.04 + 0 \\ &= -0.04. \end{aligned}$$

We see that $V_1(s_{23}) \in (-\infty, 0)$, so the correct answer is (A).

Does this make sense in our grid world? We can see that going right gives the worst expected utility, since we most likely end up moving to the -1 state. Going up or

down is better, but still has a chance of falling into the -1 state. Going left gives the maximum expected utility of 0 since there is no chance of falling into the -1 state, and this is reflected in the calculation.

Problem: Where is $V_1(s_{33})$?

(A) 0.26 (B) 0.36 (C) 0.46
(D) 0.56 (E) 0.76

$V_0(s)$:

	1	2	3	4
1	0	0	0	0
2	0	X	0	-1
3	0	0	0	$+1$

Solution: The calculation is as follows:

$$\begin{aligned} V_1(s_{33}) = -0.04 + 1 \cdot \max[&0.8 * 1 + 0.1 * 0 + 0.1 * 0, && \text{(right)} \\ &0.8 * 0 + 0.1 * 1 + 0.1 * 0, && \text{(up)} \\ &0.8 * 0 + 0.1 * 0 + 0.1 * 0, && \text{(left)} \\ &0.8 * 0 + 0.1 * 1 + 0.1 * 0] && \text{(down)} \\ = &-0.04 + 0.8 \\ = &\ 0.76. \end{aligned}$$

The correct answer is (E).

Be careful that you do not use $V_1(s_{23})$ in your calculations instead of $V_0(s_{23})$. Remember that the values of the next iteration depend only on the values of the previous iteration.

Again, the answer here makes sense. The best choice is to go right, which gives an expected utility of 0.8. Going up and down give a 10% chance of ending up in the $+1$ state, while going left gives a 0% chance. The max function will choose going right in this case.

At this point, you may wish to fill in the rest of the grid for $V_1(s)$ yourself. You may realize that this is actually quite easy. The only state with a positive expected utility will be s_{33} since it is the only state adjacent to the $+1$ state, so it is the only state from which taking one step can bring us to the $+1$ state. For all the other states, we can only hope to get closer to s_{33}, which only incurs the immediate reward of -0.04.

$V_0(s)$:

	1	2	3	4
1	0	0	0	0
2	0	X	0	-1
3	0	0	0	$+1$

$V_1(s)$:

	1	2	3	4
1	-0.04	-0.04	-0.04	-0.04
2	-0.04	X	-0.04	-1
3	-0.04	-0.04	0.76	$+1$

Intuitively, you can interpret the value updates as follows. If we are calculating $V_i(s)$, only the states within i steps of the $+1$ state will have positive estimates $V_i(s)$. If a state is unable to reach the $+1$ state within i steps, it can only accumulate negative reward at that point. In this case for $V_1(s)$, only s_{33} can reach the $+1$ state within $i = 1$ step, so only s_{33} has a positive V_1-value.

Problem: What is $V_2(s_{33})$?

(A) 0.822

(B) 0.832

(C) 0.842

(D) 0.852

(E) 0.862

Solution: The correct answer is (B).

Problem: What is $V_2(s_{23})$?

(A) 0.464

(B) 0.466

(C) 0.468

(D) 0.470

(E) 0.472

Solution: The correct answer is (A).

Problem: What is $V_2(s_{32})$?

(A) 0.16

(B) 0.36

(C) 0.56

(D) 0.76

(E) 0.96

Solution: The correct answer is (C).

Finally, here are the values for $V_2(s)$:

	1	2	3	4
1	−0.08	−0.08	−0.08	−0.08
2	−0.08	X	0.464	−1
3	−0.08	0.56	0.832	+1

These three examples are included because they are the only states with positive estimates $V_2(s)$. Again, this is because these are the only three states within $i = 2$ steps of the +1 state.

2.4 Observations from value iteration

Each state accumulates negative rewards until the algorithm finds a path to the +1 goal state. This is a bit specific to our grid world, but it should still help you understand what the value iteration algorithm is doing. The algorithm is trying to find a path to the +1 goal state, and

until it does that, it will accumulate negative rewards (the penalty of exploration). However, once it finds the +1 goal state, that +1 will add on to the reward, turning it positive.

Finally, how should we update $V^*(s)$ for all states s? There are two ways:

- synchronous: store and use $V_i(s)$ to calculate $V_{i+1}(s)$.

  This is what we used. In code, you would store two identical tables/data structures: one for the old iteration, and one for the new iteration. We fill the new iteration's table first, then replace the old iteration's table by it.

- asynchronous: store $V_i(s)$ and update the values one at a time, in any order.

  This version suggests that if certain states are more promising, you may want to update their estimates more often so that they converge faster. However, you will still want to update each state a sufficient amount of times so that the values of all states converge.

3 Policy Iteration

3.1 Introduction

Policy iteration is one of the two main approaches to solving a Markov decision process. The other one is the value iteration algorithm. Policy iteration is also useful for understanding reinforcement learning algorithms. In particular, the policy evaluation part of policy iteration is an idea that will appear again and again in reinforcement learning algorithms.

First, recall how value iteration works. The main idea of value iteration is that we iteratively estimate the V values, which are the expected utilities of the optimal policies starting from a particular state. Once we've derived accurate estimates of these expected utilities, we derive an optimal policy based on this.

Sometimes, when we're executing the value iteration algorithm, we observe that **the optimal policy stops changing even before the utility values have converged**. This tells us that sometimes deriving the optimal policy does not necessarily require accurate estimates of the utility values. As long as the estimates are reasonably accurate, we can derive the optimal policy.

This idea inspires the policy iteration algorithm.

The main idea of the policy iteration algorithm is alternating between improving the utility values and improving the policy. We start with an arbitrary initial policy. Based on this policy, we perform a step called **policy evaluation** to calculate the updated expected utility values. This is the utility of each state if the current policy were to be executed.

With the updated utility values, we perform a step called **policy improvement**, where we calculate a new policy based on the updated utility values from the previous policy evaluation. So, the steps of policy iteration are:

1. **Policy evaluation:** Given a policy π_i, calculate $V^{\pi_i}(s)$, which is the utility of each state if π_i were to be executed.

2. **Policy improvement:** Calculate a new policy π_{i+1} using V^{π_i}.

We can compare the steps of value iteration with the steps of policy iteration:

Value Iteration: $V_1(s) \to V_2(s) \to \cdots \to V^*(s) \to \pi^*(s)$
Policy Iteration: $\pi_1(s) \to V^{\pi_1}(s) \to \pi_2(s) \to V^{\pi_2}(s) \to \cdots \pi^*(s) \to V^*(s) \to \pi^*(s)$

For policy iteration we start with some arbitrary initial policy $\pi_1(s)$ and perform policy evaluation to get the updated utility values $V^{\pi_1}(s)$. Based on $V^{\pi_1}(s)$, we derive a new policy $\pi_2(s)$ through policy improvement, and we again perform policy evaluation to get $V^{\pi_2}(s)$.

We repeat this until the point where the policy does not change anymore. This happens when we reach policy $\pi^*(s)$, we perform policy evaluation to get $V^*(s)$, and then we perform policy improvement to get $\pi^*(s)$ again. Notice that this policy is the same as the previous policy, so we stop.

Observe here that it does not matter whether the utility values are optimal or not. We stop when the policies we derive stops changing. This differs from value iteration, where the goal is to derive the most accurate estimate of the utility values and then derive a policy based on that estimate.

How do we perform policy evaluation and policy improvement?

For policy improvement, we go from the utility values to an updated policy: $V^{\pi_i}(s) \to \pi_{i+1}(s)$. We already know how to do this because at the end of the value iteration algorithm when the utility values have converged, we perform this exact algorithm:

$$\pi_{i+1}(s) = \arg\max_a \sum_{s'} P(s'|s,a)V^{\pi_i}(s')$$

We take the utility values and calculate

$$Q(s,a) = \sum_{s'} P(s'|s,a)V^{\pi_i}(s')$$

which are the expected utilities of taking a particular action in a particular state. After calculating theses values, we need to figure out the best action in each state based on the Q values. This is where we find the action that achieves the maximum expected utility:

$$\pi_{i+1}(s) = \arg\max_a Q(s,a)$$

So, the steps are exactly the same.

For policy evaluation, we go from a policy to updated utility values: $\pi_i(s) \to V^{\pi_{i+1}}(s)$. To solve this, we solve the system of equations below:

$$V_i(s) = R(s) + \gamma \sum_{s'} P(s'|s,\pi_i(s))V_i(s').$$

This is similar to the bellman equations, but have some differences. In fact, they are a simplified version of the bellman equations.

Looking at the equation, we already know the reward function $R(s)$, and we know the transition probabilities that tell us the probabilities of getting to each target state based on the action we take.

We also know the action that we're taking in each particular state from policy $\pi_i(s)$. Therefore, we don't need the max function in front of the equation since we already know the action we're going to take.

The only unknowns in this system of equations are the utility values $V_i(s')$ and $V_i(s)$.

3.2 Performing Policy Evaluation

Performing policy evaluation seems to be a bit more complicated than performing policy improvement, since it seems we would need to solve a system of equations which looks similar to the bellman equations. However, this system of equations is actually much simpler to solve.

We can compare the equations for policy evaluation and the bellman equations to focus on the parts that are different:

Policy evaluation:

$$V(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi(s))V(s').$$

Bellman equations:

$$V(s) = R(s) + \gamma \max_a \sum_{s'} P(s'|s, a)V(s').$$

The utility values ($V(s)$ and $V(s')$), the reward ($R(s)$), and the discount factor (γ) are the same in both equations. The main difference between these two equations is that we are restricted to a particular policy $\pi(s)$ in policy evaluation, whereas there is a maximum over all possible actions in the bellman equations.

Turning both of these into concrete equations will show why solving the equations for policy evaluation is much easier than solving the bellman equations. We'll write down both equations when the state we're considering is the starting state; $s = s_{11}$. We also assume that the policy π says we should go downward when we're in state s_{11}:

Policy evaluation:

$$V(s_{11}) = -0.04 + \gamma(0.8 * V(S_{21}) + 0.1 * V(s_{12}) + 0.1 * V(s_{11}))$$

Bellman equations:

$$\begin{aligned} V(s_{11}) = -0.04 + \gamma \max[&0.8 * V(s_{12}) + 0.1 * V(s_{21}) + 0.1 * V(s_{11}), \\ &0.9 * V(s_{11}) + 0.1 * V(s_{12}), \\ &0.9 * V(s_{11}) + 0.1 * V(s_{21}), \\ &0.8 * V(s_{21}) + 0.1 * V(s_{12}) + 0.1 * V(s_{11})]. \end{aligned}$$

For policy evaluation, we are fixing the policy we're using, so we only have to consider one of the four possible actions. For the bellman equations, there is a maximization over four different terms representing the four possible actions.

Since the policy evaluation equations doesn't have a maximization, the equation is linear. Solving a system of linear equations allows us to use many standard linear algebra techniques. This is not possible with bellman equations as they are not linear and there's no general efficient algorithm to solve a system of nonlinear equations.

This is why it's easier to perform policy evaluation than it is to solve bellman equations. Given this, we can solve the policy evaluation equations in two ways.

Firstly, we can perform the calculation exactly using standard linear algebra techniques. The details of this is not going to be covered here since they belong to a linear algebra or linear programming course. However, in general, if there are n states, the standard technique will take $O(n^3)$ to solve these equations.

This is alright when n is small, meaning that there aren't a lot of states. However, the cubic time might be too much as n gets larger. As such, we might still want to solve problems with a large n by approximation instead.

We can use something similar to the idea of value iteration since the set of equations for policy evaluation is similar to the bellman equations. We can take the policy evaluation equations and convert them into an iterative update rule:

$$V(s) \leftarrow R(s) + \gamma \sum_{s'} P(s'|s, \pi(s))V(s').$$

Similar to value iterations, we plug in estimates of $V(s')$ on the right, then calculate the right side to give us the new estimates of $V(s)$.

We do this for a few iterations until it somewhat converges. The number of iterations depends on how accurate we want the estimates to be. This allows us to get reasonably accurate estimates of the utility values without spending a lot of time.

To summarize, the key idea of policy iteration is to iteratively improve both the policy and the estimates of the utility values. There are two steps in this process. The first is policy evaluation where we start with a policy and use it to estimate the utility values. The second is policy iteration where we start with some estimates of the utility values and derive a new best policy.

3.3 An example of policy iteration

In the previous sections, we introduced the high level ideas of the policy iteration algorithm. We will now run through an entire example executing the algorithm. Obviously, we can't work a very large example, because all the calculations are to be done by hand.

Here is a tiny grid world, a two by two world, where we have two goal states, +1 and -1 with reward +1 and -1:

	+1
	-1

So there are really only two states that we're concerned with: s_{11} and s_{21} are the two states we want to determine the policy as well as the utility values.

The setting is pretty much the same as the more complicated example in previous sections. So the reward of entering a non-goal state is -0.04, so same as before. And also we'll keep the transition probabilities to be the same.

So whenever we are trying to go in and direct a direction, there is 80% chance that we will end up going in the intended direction. Then 10% chance we will go to the left of the intended direction, and 10% chance to the right of the intended direction. And same as before, if we ever hit a wall, then we will remain in the same location.

So let's execute the policy iteration algorithm on this tiny grid world to find out what are the utility values for these two states, and also what should be the optimal policy.

Now, if you remember for the policy iteration algorithm we are going to alternate between two steps, one is evaluating the policy, and the other one is improving the policy.

Let's start with an initial policy where we're going right in both states. We set this so that our example is substantial: we do have to go through multiple steps, but there won't be too many steps.

So starting with this initial policy, we will perform a policy evaluation, which means given this policy, we need to figure out the corresponding utility values for the two states.

$\rightarrow$	+1
$\rightarrow$	-1

Essentially, we are going to take the Bellman equations and write them down for this particular policy, and if you recall the Bellman equations basically gives us a relationship between the utility values for neighboring states.

Because there are two states we're interested in, we will write down two equations. One for the V value of s_{11} and the other one for the V value of s_{21}.

Bellman equations dictates that we have some immediate reward of entering a state. In this case, either state is a non-goal state so we'll get -0.04, and then there is also some long-term utility, or long-term discounted reward of getting into the next state.

So based on our transition probabilities, for example, our current policy for s_{11} is going to our right, so with 80% chance we'll go to our right and end up in the +1 reward state. That's s_{12}, so we'll get +1.

With 10% chance we're going to go up, which is left of the intended direction. There's a wall, so we're going to come back. And that's why we get $V(s_{11})$ as the second term.

And finally the third possible direction is going down, we're going to the right of the intended direction, which gets us to s_{21}. So the third term is $0.1V(s_{21})$.

$$V(s_{11}) = -0.04 + 0.8(+1) + 0.1V(s_{11}) + 0.1V(s_{21})$$

Notice here that this is different from the original Bellman equations, because the original Bellman equation says we're going to take the maximum over all possible policies, and there are four possible policies for each location. However, here we have a fixed policy already, so we don't need to take the maximum — we can simply follow that fixed policy and see what happens.

Also notice the equation's linear because of this. If we had the maximum term in the equation it would not be linear.

The equation for s_{21} is very similar. We get the immediate reward again and because our policy is also going right, with 80% chance we'll get into the -1 reward state. With 10% percent chance we'll go up and get into s_{11}, and with another 10% chance we'll go down and get into s_{21} because we hit a wall.

$$V(s_{21}) = -0.04 + 0.8(-1) + 0.1V(s_{11}) + 0.1V(s_{21})$$

Now we have the two equations, each of which have two unknowns, $V(s_{11})$ and $V(s_{21})$. They're both linear equations, so we basically have a linear system of equations, with two equations and two unknowns. We can solve this exactly, by hand or using linear programming. Here it's simple enough so we can do it by hand.

The following steps are the simplification of the equations:

$$\begin{cases} V(s_{11}) = -0.04 + 0.8(+1) + 0.1V(s_{11}) + 0.1V(s_{21}) \\ V(s_{21}) = -0.04 + 0.8(-1) + 0.1V(s_{11}) + 0.1V(s_{21}) \end{cases}$$

$$\Rightarrow \begin{cases} 0.9V(s_{11}) - 0.1V(s_{21}) = 0.76 \\ -0.1V(s_{11}) + 0.9V(s_{21}) = -0.84 \end{cases}$$

$$\Rightarrow \begin{cases} 8V(s_{11}) = 6 \Rightarrow V(s_{11}) = 0.75 \\ 0.9V(s_{21}) = -0.765 \Rightarrow V(s_{21}) = -0.85 \end{cases}$$

The final result is that $V(s_{11})$ is 0.75 and $V(s_{11})$ is -0.85. These two numbers make sense, because both policies are going right. For the top square going right, we will probably get into the +1 state, which is great. So that's why $V(s_{11})$ has a positive value and it's close to 1. Whereas for s_{21} we will probably get into the -1 state. So that's why the number is negative and close to -1 as well.

So this is the first policy evaluation step. Starting from the initial policy, which is going right in both squares, we derive the utility values for the two squares. Next, we will solve for the best policy given the current estimates of the utility values. This is the policy improvement step.

What is the best policy for s_{11} here? We need to consider all possible policies, therefore going right, going up, going left, and going down. For each of the policies, we are going to calculate my expected reward, or utility.

We will ignore the immediate reward because the immediate reward is always the same. Instead, we calculate the utility if we went in the intended direction, the left of the intended direction, and the right of the intended direction. So, the expected long-term reward if the policy is going right will be:

$$\text{right } 0.8(+1) + 0.1(0.75) + 0.1(-0.85) = 0.79$$

Right, so having done the four calculations then finally we just have to compare all the numbers and pick the policy that gives us the largest number.

$$\begin{aligned}
&\text{up } 0.8(0.75) + 0.1(0.75) + 0.1(+1) = 0.775 \\
&\text{left } 0.8(0.75) + 0.1(-0.85) + 0.1(0.75) = 0.59 \\
&\text{down } 0.8(-0.85) + 0.1(+1) + 0.1(0.75) = 0.505
\end{aligned}$$

In this case, the policy of going right gives us the largest expected reward, which is 0.79. So given the current utility values, the optimal policy for s_{11} is going right.

Similarly, we can do this for s_{21}, and similar calculations you can take a look on your own time, but in the end we found that the best policy is going up. This makes sense as you look at s_{21}, going right will most likely give us a reward of -1. Whereas going up, we have a positive reward of 0.75 there.

This is it for our policy improvement step. Before the policy improvement step, our policy was going right for both squares. After the policy improvement step, we found a better one so the better policy is going right in s_{11} and going up for s_{21}.

$\rightarrow$	+1
$\uparrow$	-1

Since the policy changed, we have to keep executing the algorithm. Remember for policy iteration, as long as the policy changes, we have to keep going, and we can only terminate when the policy stays the same, which means we cannot improve the policy anymore.

So our current policy is right and up, and we will follow a very similar procedure to repeat the policy evaluation. These two equations should look different now. It's just that for s_{11}, the equation for s_{11} would look similar if not identical because our policy did not change. The equation for s_{21} certainly changed because our policy changed from going right to going up. Writing down the two equations, again we have two equations two unknowns they're

both linear.

$$\begin{cases} V(s_{11}) = -0.04 + 0.8(+1) + 0.1V(s_{11}) + 0.1V(s_{21}) \\ V(s_{21}) = -0.04 + 0.8V(s_{11}) + 0.1V(s_{21}) + 0.1(-1) \end{cases}$$
$$\Rightarrow \begin{cases} 0.9V(s_{11}) - 0.1V(s_{21}) = 0.76 \\ -0.8V(s_{11}) + 0.9V(s_{21}) = -0.14 \end{cases}$$
$$\Rightarrow \begin{cases} 7.3V(s_{11}) = 6.7 \Rightarrow V(s_{11}) = 0.918 \\ 0.9V(s_{21}) = 0.5944 \Rightarrow V(s_{21}) = 0.660 \end{cases}$$

We have that the two utility values as 0.918 and 0.660. This is another policy evaluation step where we take the current policy and figure out the utility values given the policy. Next, we have to carry out another policy improvement step.

0.918	+1
0.660	-1

Given our current utility values which are 0.918 and 0.660, can we derive a better policy? Is the policy going to change? So the process is exactly the same as before. We are going to consider all four possible policies, and for each policy we'll calculate our expected long-term reward. The key is thinking about where we're going next. With 80% chance where are we going up, and so on and so forth.

$$\text{right } 0.8(+1) + 0.1(0.918) + 0.1(0.660) = 0.9578$$
$$\text{up } 0.8(0.918) + 0.1(0.918) + 0.1(+1) = 0.9262$$
$$\text{left } 0.8(0.918) + 0.1(0.660) + 0.1(0.918) = 0.8922$$
$$\text{down } 0.8(0.660) + 0.1(+1) + 0.1(0.918) = 0.7198$$

After all of these calculations, the largest number we have is for going right. Therefore, the optimal policy for s_{11} is still going right, same as our current policy. We'll do a very similar calculation for s_{21}. Again, trying all four possibilities, we'll end up with all of these numbers, and turns out the best policy for s_{21} is going up.

Can we stop the algorithm now? Our previous policy was right and up and our new policies still right and up — The policy did not change. Because of this, we can terminate the algorithm and claim that we have found the optimal policy.