3. Value Iteration and Our First Lower Bound

PDF Version

Last time, we discussed the Fundamental Theorem of Dynamic Programming, which then led to the efficient "value iteration" algorithm for finding the optimal value function. And then we could find the optimal policy by greedifying w.r.t. the optimal value function. In this lecture we will do two things:

1. Elaborate more on the the properties of value iteration as a way of obtaining near-optimal policies;
2. Discuss the computational complexity of planning in finite MDPs.

Finding a Near-Optimal Policy using Value Iteration

In the previous lecture we found that the iterative computation that starts with some and then obtains using the "Bellman update"

leads to a sequence whose th term approaches , the optimal value function, at a geometric rate:

While this is reassuring, our primary goal is to obtain an optimal, or at least a near-optimal policy. Since any policy that is greedy with respect to (w.r.t) is optimal, a natural idea is to stop the value iteration after some finite number of iteration steps and return a policy that is greedy w.r.t. the approximation of that was just obtained. If we stop the process after the th step, this defines a policy such that is greedy w.r.t. : . The hope is that as approaches , the policies will also get better in the sense that decreases.

The next theorem guarantees that this will indeed be the case.

Theorem (Policy Error Bound): Let be arbitrary and be the greedy policy w.r.t. : . Then,

In words, the theorem states that the **policy error** ( ) of a policy that is greedy with respect to a function is controlled by the distance of to . This can also be seen as stating that the "greedy operator" , which maps functions to a policy that is greedy w.r.t. , is continuous at when the "distance" between policies is defined as the maximum norm distance between their value functions: . Indeed, with the help of this notation, an alternative form of the theorem statement is that for any ,

In words, this can be described as that is is " -smooth" at when the input space is equipped with the maximum norm distance and the output space is equipped with . One can also show that this result is sharp in that the constant cannot be improved.

The proof is an archetypical example of proofs of using contraction and monotonicity arguments to prove error bounds. We will see variations of this proof many times. Before the proof, let us introduce the notation for a vector to mean the componentwise absolute value of the vector: , .

As a way of using this notation, note that for any memoryless policy ,

and hence

In Eq. the first inequality follows because is monotone and . For the proof it will also be useful to recall that we also have

for any , and memoryless policy . These two identities follow just by the definitions of and , as the reader can easily verify them.

Proof: Let be as in the theorem statement and let . Let . The result follows by algebra once we prove that . Hence, we only need to prove this inequality.

By our assumptions on and , . Now,

Taking the (pointwise) absolute value of both sides and using the triangle inequality, and then Eq. we find that The proof is finished by taking the maximum over the components, noting that .

An alternative way of finishing the proof is to note that from , by reordering and using that is a monotone operator, . Taking the max-norm of both sides, we get .

Value Iteration as an Approximate Planning Algorithm

From Eq. we see for ——————, started with , value iteration yields such that and consequently, for a policy that is greedy w.r.t. , ——. Now, for a fixed setting so that —— holds, we see that after ____ iterations, we get a -optimal policy : . Computing using takes elementary arithmetic (and logic) operations. Putting things together we get the following result:

Theorem (Runtime of Approximate Planning with Value Iteration): Fix a finite discounted MDP and a target accuracy . Then, after

elementary arithmetic operations, value iteration produces a policy that is -optimal: , where the result holds when is fixed and hides a term.

Note that the number of operations needed depends very mildly on the target accuracy. However, accuracy here means an additive error. While the optimal value could be as high as , it can easily happen that the best value that can be achieved, , is significantly smaller than . It may be for example that , in which case a guarantee with is vacuous.

By a careful inspection of we can improve the previous result so that this problem is avoided:

Theorem (Runtime when Controlling for the Relative Error): Fix a finite discounted MDP and a target accuracy . Then, stopping value iteration after iterations, the policy produced satisfies the relative error bound

while the total number of elementary arithmetic operations is

where hides .

Notice that the runtime required to achieve a fixed relative accuracy appears to be the same as the runtime required to achieve the same level of absolute accuracy. In fact, the

runtime slightly decreases. This should make sense: The worst-case for the fixed absolute accuracy is when , and in this case the relative accuracy is significantly less demanding: With , value iteration can stop after guaranteeing values of , which, as a value, is much smaller than , the target with the absolute accuracy level of .

Note that the relative error bound is not without problems either: It is possible that for some states , is negative, a vacuous guarantee. A reasonable stopping criteria would be to stop when the policy that we read out satisfies

Since is not available, to arrive at a stopping condition that can be verified and which implies the above inequality, one can replace above with an upper bound on it, such as . In this imagined procedure, in each iteration, one also needs to compute the value function of policy to verify whether the stopping condition is met. If we do this much computation, we may as well replace with in the update equation hoping that this will further speed up convergence. This results in what is known as **policy iteration**, which is the subject of the next lecture.

The Computational Complexity of Planning in MDPs

Now that we have our first results for the computation of approximately optimal policies, it is time to ask whether the algorithm we discovered is doing unnecessary work. That is, what is the minimax computational cost of calculating an optimal, or approximately optimal policy?

To precisely formulate this problem, we need to specify the inputs and the outputs of the algorithms considered. The simplest setting is when the inputs to the algorithms are arrays, describing the transition probabilities and the rewards for each state action pair with some ordering of state-action pairs (and next states in the case of transition probabilities). The output, by the Fundamental Theorem, can be a memoryless policy, either deterministic or stochastic. To describe such a policy, the algorithm could write a table. Clearly, the runtime of the algorithm will be at least the size of the table that needs to be written, so the shorter the output, the better the runtime can be. To be nice with the algorithms, we should allow them to output deterministic policies. After all, the Fundamental Theorem also guarantees that we can always find a deterministic memoryless policy which is optimal. Further, greedy policies can also be chosen to be deterministic, so the value-iteration algorithm would also satisfy this requirement. The

shortest specification for a deterministic policy is an array of the size of the state space that has entries.

Thus, the runtime of any algorithm that needs to "produce" a fully specified policy is at least .

This is quite bad! As was noted before, , the number of states, in typical problems is expected to be gigantic. But by this easy argument we see that if we demand algorithms to produce fully specified policies then without any further help, they have to do as much work as the number of states. However, things are a bit even worse.

In Homework 0, we have seen that no algorithm can find a given value in an array without looking at all entries of the array (curiously, we saw that if we allow randomized computation, that on expectation it is enough to check half of the entries).

Based on this, it is not hard to show the following result:

Theorem (Computation Complexity of Planning in MDPs):

Let . Any algorithm that is guaranteed to produce -optimal policies in any finite MDP described with tables, with a fixed discount factor and rewards in the interval needs at least elementary arithmetic operations on some MDP with the above properties and whose state space is of size and action space is of size .

Proof sketch: We construct a family of MDPs such that no matter the algorithm, the algorithm will need to perform the said number of operations in at least one of the MDPs.

One-third of the states is reserved for "heaven", one-third is reserved for "hell" states. The remaining one-third set of states, call them , is where the algorithms will need to make some nontrivial amount of work. The MDPs are going to be deterministic. In the tables given to the algorithms as input, we (conveniently for the algorithms) order the states so that the "hell" states come first, followed by the "heaven" states, followed by the states in .

In the "heaven" class, all states self-loop under all actions and give a reward of one. The optimal value of any of these states is . In the "hell" class, states also self-loops

under all actions but give a reward of zero. The optimal value of these states is . For the remaining states, all actions except one lead to some hell state, while the chosen special action leads to some state in the heaven class.

The optimal value of all states in set have a value of and the value of a policy that in a state in does not choose the special optimal action gets the value of in that state. It follows that any algorithm that is guaranteed to be optimal needs to identify the unique optimal action at every state in .

In particular, for every state and action , the algorithm needs to read entries of the transition probability vector or it can't find out whether leads to a state in the heaven class or the hell class: The probability vector will have a single one at such an entry, either among the entries representing the hell, or the entries representing the heaven states. By the aforementioned homework problem, any algorithm that needs to find this "needle" requires to check entries. Since the number of states in is also , we get that the algorithm needs to do work.

We immediately see two differences between the lower bound and our previous upper bound(s): In the lower bound there is no dependence on (the effective horizon at a constant precision). Furthermore, there is no dependence on , the inverse accuracy.

As it turns out, the dependence on of value-iteration is superfluous and can be removed. The algorithm that achieves this is policy iteration, which was mentioned earlier. However, this result is saved for the next lecture. After this, the only remaining gap will be the order of the polynomials and the dependence on , which is closely related to the said polynomial order.

And of course, we save for later the most pressing issue that we need to somehow be able to avoid the situation when the runtime depends on the size of the state space (forgetting about the action space for a moment). By the lower bound just presented we already know that this will require changing the problem setting. Just how to do this will be the core question that we will keep returning to in the class.

Notes

Value iteration

The idea of value iteration is probably due to Richard Bellman.

Error bound for greedification

This theorem is due to Singh & Yee, 1994.

The example that shows that the result stated in the theorem is **tight**. Consider an MDP with two states, call them and , two actions, and deterministic dynamics. Call the two actions and . Regardless the state where it is used, action makes the next state transit to state , while giving a reward of . Analogously, action makes the next state transit to state , while giving a reward of . The optimal values in both states are . Let be so that , while . Thus, underestimates the value of , while it overestimates the value of state . It is not hard to see that the policy that uses action regardless the state is greedy with respect to (actually, the action-values of the two actions tie at both states). The value function of this policy assigns the value of to both states, showing that the result stated in the theorem is indeed tight.

Computational complexity lower bound

The last theorem is due to Chen and Wang (2017), but the construction is also (unsurprisingly) similar to one that appeared in an earlier paper that studied query complexity in the setting when the access to the MDP is provided by a simulation model. In fact, we will present this lower bound later in a lecture where we study batch RL. According to this result, the **query-complexity** (also known as sample-complexity) of finding a -optimal policy with constant probability in discounted MDPs accessible through a random access simulator, apart from logarithmic factors, is , where .

Representations matter

We already saw that in order to just clearly define the computational problems (which is necessary for being able to talk about lower bounds), we need to be clear about the inputs (and the outputs). The table representation of MDPs is far from being the only possibility. We just mentioned the "simulation model". Here the algorithm "learns" about the MDP by issuing next state and reward queries to the simulator at some state-action pair of its choice to which the simulator responds with a random next state (drawn fresh) and the . Interestingly, this can provably reduce the number of queries compared to the table representation.

Another alternative, which still keeps tables, is to give the algorithm a cumulative probability representation. In this representation, the states are identified with as before but instead of giving the algorithm the tables for fixed , the algorithm is given

(the last entry could be saved, because it is always equal to one, but in the grand scheme of things, of course, this does not matter). Now, it is not hard to see that if the original probability vector had a single one and zeroes everywhere else, the "needle in the haystack problem" used in the lower bound, with the integral representation above, a clever algorithm can find the entry with the one with at most queries. As it turns out, with this representation, the **query complexity** (number of queries required) of producing a good policy can indeed be reduced from the quadratic dependence on the size of the state-space to a log-linear dependence. Hence, we see that the input representation crucially matters. Chen and Wang (2017) also make this point and they discuss yet another, "tree" representation, which leads to a similar speedup.

MDPs with short descriptions

The simulator model assumption addresses the problem that just reading the input may be the bottleneck. This is not the only possibility. One can imagine various classes of MDPs that have a short description, which may raise the hope that one can find out a good policy in them without touching each state-action pair. There are many examples of classes of MDPs that belong to this category. These include

- factored MDPs: The transition dynamics have a short, structured (factored) representation, and the same applies to the reward
- parametric MDPs: The transition dynamics and the rewards have a short, parametric representation. Examples include linear-quadratic regulation (linear dynamics, quadratic reward, Euclidean state and action spaces, Gaussian noise in the transition dynamics), robotic systems, various operations research problems.

For factored MDPs one is out of luck: In these, planning is provably "very hard" (computationally). For linear-quadratic regulation, on the other hand, planning is "easy"; once the data is read, all one has to do is to solve some algebraic equations, for which efficient solution methods have been worked out.

Query vs. computational complexity

The key idea of the lower bound crucially hinges upon that good algorithms need to "learn" about their inputs: The number of arithmetic and logic operations of any algorithm is at least as large as the number of "read" operations it issues. The minimum number of required read operations to produce an input of some desired property is often called the problems **query complexity** and by the above reasoning we see that the computational complexity is lower bounded by the query complexity. As it happens, query

complexity is much easier to bound than computational complexity in the sense that it is rare to see computational complexity lower bounds strictly larger than the query complexity (the exceptions to this come when a "compact" representation of the MDP is available, such as in the case of factored MDPs). At the heart of query complexity lower bounds is often the needle in the haystack problem. This seems to be generally true when the inputs are "deterministic". When querying results in stochastic (random) outcomes, multiple queries may be necessary to "reject", "reduce", or "filter out" the noise and then new considerations appear.

In any case, query complexity is a question about quickly determining the information crucial to arrive at a good decision early and is in a way about "learning": Before a table is read, the algorithm does not *know* which MDP it faces. Hence, query complexity is essentially an "information" question and is also sometimes called **information complexity** and we can think of query complexity as the most basic information theory question. This is a bit different though than mainstream information theory, which is somehow tied up in dealing with reducing the effect of random responses (random "corruptions" of the clean information).

Query complexity everywhere

Query complexity is widely studied in a number of communities which, sadly, are almost entirely disjoint. Information-theory, mentioned above is one of them, though as was noted, here the problems are often tied to studying the speed of gaining information in the presence of noise. Besides information theory, there is the whole field of information-based complexity, which has its own journal, multiple books and more. Also notable is the theory community that studies the complexity of evolutionary algorithms. Besides these, of course, query complexity made appearances in the optimization literature (with or without noise), operations research, and of course in the machine learning and statistics community. In particular, in the machine learning and statistics community, when the algorithm is just handed over noisy data, "the sample", one can ask how large this sample needs to be to achieve some good outcome (e.g., good predictions on unseen data). This leads to the notion of **sample complexity**, which is the same as our query complexity except that the queries are of the "dull", "passive" nature of "give me the next datapoint". As opposed to this, "active learning" refers to the case when the algorithms themselves control some aspects of how the data is collected.

Free lunches, needles and a bit of philosophy

Everyone after going to a few machine learning conferences or reading their first book, or blog posts would have heard about David Wolpert's "no-free lunch theorems". Yet, I find

that to most people the exact nature (or significance) of these theorems remain elusive. Everyone heard that these theorem essentially state that "in the lack of bias, all algorithms are equal" (and therefore there is no free lunch), from which we should conclude that the only way to choose between algorithms is by introducing bias.

But what does bias means? If one reads these results carefully (and the theory community of evolutionary computation made a good job of making them accessible) one finds that the results are nothing more that describing some corollaries that to find a needle in a haystack (the special entry in a long array), one needs to search the whole haystack (query almost all entries of the array).

Believers of the power of data like to dismiss the significance of the no-free lunch result by claiming that it is ridiculous in that it assumes no structure at all. I find these arguments weak. The main problem is that they are evasive. The evasiveness comes from the reluctance to be clear about what we expect the algorithms to achieve. The claim is that once we are clear about this, that is, clear about the goals, or just the problem specification, we can always hunt for the "needle in the haystack" subproblems within the problem class. This is about figuring out the symmetries (as symmetry equals no structure) that sneakily appear in pretty much any reasonable problem we think of worth studying. The only problems that do not have "needle in the haystack" situations embedded into them are the ones that are not specified at all.

What is the upshot of all this? In a way, the real problem is to be clear about what the problem we want to solve is. This is the problem that most theoreticians in my field struggle with every day. Just because this is hard, we cannot give up on this before even starting, or this will just lead to chaos.

As we shall see in this class, how to specify the problem is also at the very heart of reinforcement learning theory research. We constantly experiment with various problem definitions, tweaking them in various ways, trying to separate hopelessly hard problems from the easy, but reasonably general ones. Theoreticians like to build a library of various problem settings that they can classify in various ways, including relating the problem settings to each other. While algorithm design is the constructive side of RL (and computer science, more generally), understanding the relationship between the various problem settings is just as equally important.

References

- Chen, Y., & Wang, M. (2017). Lower bound on the computational complexity of discounted markov decision problems. arXiv preprint arXiv:1705.07312. [link]
- Singh, S. P., & Yee, R. C. (1994). An upper bound on the loss from approximate optimal-value functions. Machine Learning, 16(3), 227-233. [link]