Modelling and abstraction for MDPs

Contents

Learning outcomes

The learning outcomes of this chapter are:

1. Describe modelling and abstraction strategies to scale MDP algorithms to problems.
2. Apply modelling and abstraction strategies to non-trivial MDP problems..

Overview

As discussed through Part I of this book, often our reinforcement learning algorithms struggle with *scale*. That is, as the state and action spaces of problems get larger, the time taken to arrive at good policies becomes increases exponentially.

Two strategies that are useful if you can apply them are to : (1) apply more computational power to the problem so that you can run more episodes or assess more states; or (2)

Skip to main content

However, often we may already be using all the computational power we have access to, or maybe we are not working in a simulated environment at all. In these cases, one way is to simplify the problem and solve that.

In this section, I give some tips that I have found useful when applying reinforcement learning techniques. Implementing these tips requires us to understand the domain that we are working in. Such an implementation will only work on that domain or others that are very similar – they will not likely generalise to other domains. However, these tips/strategies themselves are general, and would work on many domains if we take the time to apply them.

Abstraction

One solution is to define an abstracted version of the MDP, solve that abstracted problem, and then translate the policy back to the original problem. Using abstraction, we take the original MDP model M that we have, and define a new MDP model M_A that is smaller than the original: it has fewer states and/or fewer actions than the original model.

The process is that the modeller should ask themselves the following questions:

1. What are the important decisions that I need to make in this problem to reach achieve some objective?
2. What information do I need to do this?
3. Can I translate the original MDP action and state spaces into smaller action and state spaces so that I can access this information and discard some information that is less important?

We then define an abstract model $M' = (S', s'_0, A', P', r', \gamma')$ that is an abstraction of M, such that there is some mapping $f_s : S \rightarrow S'$ such that $f_s(s)$ abstracts state s into another state s'; and similarly mappings for A, P, r', and even γ (the length of plans will be different if the action space is smaller).

We solve the new model M', obtaining abstract policy $\pi' : S' \rightarrow A'$. We then need to define a mapping $g : A' \rightarrow A$ to map each abstract actions to actions in the original model. Given this, we can then define the policy for M as:

$$\pi(s) = g(\pi'(f_s(s)))$$

This means that $\pi(s)$ translates s to $s' \in S'$ using f_s, then applies abstract policy π' to get

Skip to main content

By “model”, I don’t just mean for model-based reinforcement learning: even in a model-free environment we can abstract the actions and states that we see, and modify the rewards, by putting a “layer” in between the environment and the reinforcement learning algorithm.

For example, in these notes we use the GridWorld problem to illustrate various techniques because it is simple and intuitive. This is already an abstraction of a real navigation problem. In a real navigation problem, a robot would have to move in a continuous environment, with actions such as rotating and accelerating. However, our objective in the problem is to find the best route, not to move the wheels. For the purposes of finding the best route, we model the problem differently by dividing the state space into grids and mapping the actions into directions, as this is all the information that we need to find a good, but perhaps not optimal, route. If we wanted to actually move a robot, we would need a layer in-between that: (1) translates the real states into our abstracted grid coordinators and identifies when we are in a reward state, which is function f_s; and (2) translates our abstracted actions back into actions in the environment, which is function g. For example, the action `Right` may translate into rotating 90 degrees and then moving forward 2 metres.

Sometimes a problem may already be as abstract as it can be. For example, if we are learning a controller for a robotic arm in a manufacturing facility, the information set from the sensors and to the actuators may strongly influence the effect of higher-level actions, so cannot be abstracted away. However, in practice, there are usually abstractions we can use.

In reality, most of the techniques in this section are abstraction techniques of some sort, but some other tricks can help.

State abstraction

In some cases, the size of the state space can be abstracted by combining some states that are different, but are either equivalent from a semantic view, or are similar enough that they will lead to the same action most of the time.

AlphaGoZero, the famour Go-playing agent, used state abstraction by noting that the state is game is symmetric: it is a 19x19 grid and if the first move is to e.g. place a stone in one of the corners, then *which* corner is not relevant: placing a stone in all four corners corresponds to the same abstract state. This reduces the size of the state space to a quarter of its original size, and does not lose any information

Other abstraction techniques may lose information, but still be useful. For example, the state abstraction in GridWorld from precise coordinates into a simple grid.

Skip to main content

Action abstraction

Similar to state abstraction, we can abstract actions. There are two main ways:

1. Abstract sequences of smaller actions into single higher-level actions, such as in the GridWorld domain where we abstract physical navigation actions to just `Left`, `Right`, `Up`, and `Down`.
2. We can also *group* similar actions into a single action. For example, in the GridWorld domain, if we have lower-level actions like `Rotate`, which rotates by a given number of degrees, instead of reasoning about 360 different version of `Rotate` (one for each degree), we could group actions into buckets: one abstract action for turning 0-15, one for 16-30, etc. We *lose* information/precision, but in may cases, the solution may be good enough, and the problem is much easier to solve.

Rewarding sub-goals

In some problems, there are final goals that need to be achieved, and we receive rewards for transitioning to states where those goals are achieved. One problem we saw in these notes, such as in the Freeway example, is when those rewards are a long way from the start state. As we often start with a random policy or value/Q-function, we spend a lot of time doing random simulation before we receive a reward by achieving the goal.

However, often these problems have *sub-goals* that we need to achieve on the way. If we do not achieve the sub-goals, we cannot achieve the primary goal. For example, in the Freeway game, we need to reach row 1, then row 2, etc., until we reach the other side.

If we can identify these sub-goals, we can give “partial” rewards to the sub-goals on the way to the goal. In this case, we identify key states (or more accurately, key transitions) that are like sub-goals of our problem. In Freeway, assuming the reward for reaching the other side is 100 points, we could give 1 point for reaching row 1, then 2 points for reaching row 2, etc., and then 100 - (1 + 2 + ... + n) for reaching the other side, where n is the 2nd-last row.

For an MDP $(S, s_0, A, P, r, \gamma)$, the process is:

1. Identify sub-goals that need to be achieved to achieve the final goal.
2. Create a new reward function r' where the rewards are re-distributed to sub-goals, and where cumulative reward of using the original reward r of any episode e that reaches a

Skip to main content

3. Solve a new MDP $(S, s_0, A, P, r', \gamma)$ to obtain policy π (the MDP is the same but just with a different reward function).
4. Evaluate π on the original MDP.

For example, consider the single-agent card game *Solitaire*. The aim of the game is to win by having all cards placed in four piles, with various rules for when you can move a card to the pile. Each pile must be in order from ace to king, all of the same suit.

Imagine we are given a simulator for Solitaire and the only reward received for winning the game by arriving in one of the terminal states where the cards are in the correct piles. This reward would only be given in that terminal state, and requires perhaps a few hundred actions to arrive at.

Simulations of a game like this would have to be long because there are no rewards until the end of the game, and we would spend a LOT of random actions before we first won the game By re-distributed sub-goals that give a reward whenever a correct card is place on one of the piles, we feed rewards forward much earlier and give information for our reinforcement learning algorithm to exploit.

This idea can be implemented using reward shaping. In these notes, we focused on potential-based reward shaping, which gives small "fake" rewards. This effectively asks that you define a heuristic for states that guide the agent towards promising actions early during learning when there is not much information to exploit. By using sub-goals, we can give a shaped reward $\Phi(s)$ to any state that is a sub-goal, and assign $\Phi(s) = 0$ for any state that does not. Thinking about reward shaping as sub-goals can often be conceptually simpler to implement than considering it as an heuristic.

Alternatively, we can just "intercept" the real reward function with our new reward function and not use reward shaping, but reward shaping is a more principled way of doing this and offers convergence guarantees when using Q-tables and linear approximation.

Partially-observable state variables

Often we have situations where some variables are not observable for at least part of the game. For example, in Solitaire, the draw cards are not visible until we take each one. In this case, the model is a partially-observable Markov Decision Process (POMDP). We can translate this to an MDP where the states are not represented as states in the real-world, but are *beliefs* over states in the real world. However, the computational complexity of this is high.

One option is to simplify by *ignoring the partially observable variables.* In Solitaire, it is simple enough to just not track the cards in the deck and only consider actions for the cards that we can already see. In Solitaire, this will work well. If you have ever played the game, you will know that you spend very little time wondering about the cards in the deck.

Of course, this will not always be a good idea: sometimes it is important to keep track of the partially-observable variables; for example, by mapping the probability that certain cards could be played in the future.

Heuristics

Using heuristics is a good way to speed up learning. There are three places where we can often use heuristics.

State/action pruning. First, if we have a reasonable heuristic, we can choose to "prune" states that are not promising, and explore only those that appear to be promising. For example, consider using MCTS for Solitaire. If we are using a model or a simulator, we may be able to peek at the states resulting from actions. If one action results in us being able to move a block of cards to the goal pile, we may prioritise doing simulations using this action. Given we expand the first node fully before expanding any other nodes, we could look at the heuristic value of each node and prune it from the tree if it is too low. This would cut down our search space. Of course, a poor heuristic could do more damage than good! But it is not just MCTS that benefits. For example, using Q-learning or SARSA, we could terminate episodes when we reach unpromising states, either start new episodes or backtracking to an earlier promising node is using a simulator.

Exploiting. Second, heuristics can be good for helping to guide episodes towards good solutions. In any model-free or simulation-based technique, early episodes effectively choose each action with a uniform probability, as there is no good information to exploit. Like reward shaping, we can use heuristics to choose the most promising actions more often, meaning that simulations are still *randomised*, but informed. In MCTS, when we expand a node and then simulate, this is particularly valuable, as if we can find reasonable simulations, we get a better idea of the value of the expanded node. The trick here is to balance the heuristic against the information that we learn into our Q-function. Either we have a weighted measure, or we stop using the heurstic after we gain sufficient information in our Q-functions.

Shorter episodes and simulations. Finally, we can use heuristics to terminate episodes/simulations early. As we move towards terminal states, our heuristic values are

Skip to main content

more information. As such, we can use heuristics to terminate episodes early. For example, in MCTS, instead of simulating until a terminal state, we can simulate for X number of actions and then return the heuristic value of the state. This is an estimate, but early in the learning process, it can often be more valuable to shorter simulations that return heurstic values rather than much longer ones that take more time and often involve a lot of actions that were not helpful. In fact, this is precisely what temporal different methods like Q-learning and SARSA do! They use $\max_a Q(s', a')$ and $Q(s', a')$ as heuristic estimates of the value of state s' so that we can assign 'credit' to the previously executed action. In those techniques, we are learning the heuristic. Hand-crafting a heuristic will be more helpful in earlier episodes.

Summary

- Techniques for solving MDPs face scalability issues. Using modelling tricks, we can find problems that are easier to solve, and apply them back
- Sometimes the smaller problem is enough to solve our problem; other times, it is not.

Previous
Policy gradient methods

Next
Normal form games