CE 273
Markov Decision Processes

Lecture 17

Introduction to Approximate Dynamic Programming

Previously on Markov Decision Processes

The DP algorithm is more efficient than a brute force computation for finite state spaces. To see why, assume that there are n states and m actions in each state at each time step.

Then a total of nmN expectation calculations must be performed. Each expectation calculation roughly requires $O(n)$ calculations.

On the other hand, if we were to evaluate all policies we would require $(m^n)^N$ calculations.

Previously on Markov Decision Processes

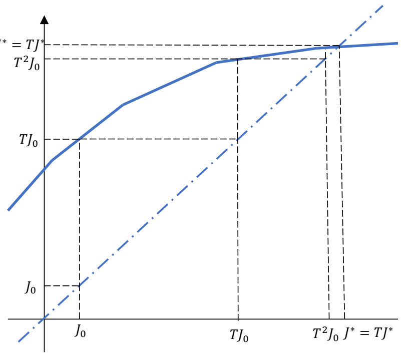


Figure: Value Iteration

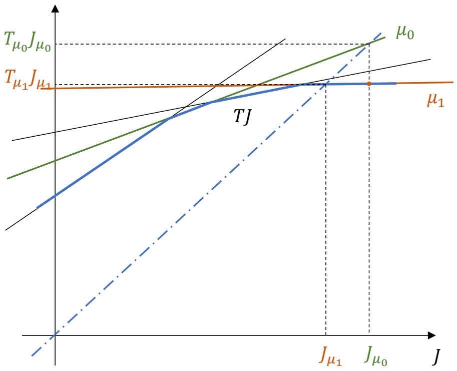
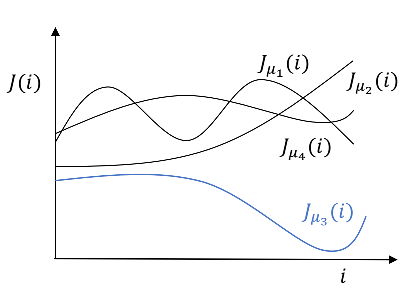


Figure: Policy Iteration

Previously on Markov Decision Processes



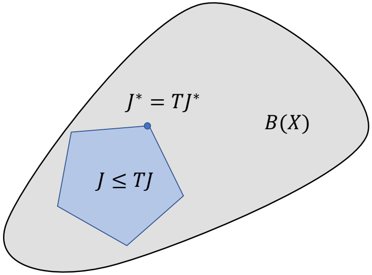


Figure: Linear Programming

Lecture Outline

Lecture Outline

Overview

Overview
Motivation

As seen earlier, MDPs suffer from the curse of dimensionality because of potentially large

- State spaces (storing large J and μ may be computationally prohibitive)
- Action spaces (minimization in VI or searching over policies is time consuming)
- Disturbance spaces (expectation calculations become difficult)

Can you give an example where each of the above problems might arise?

Overview

Motivation

The other major issue with MDPs is that it is data intensive. One needs to know the transition probabilities for all actions and the one-stage costs. In practice,

- We might not have a mathematical model or it might be too clumsy to write in closed form. Instead we might have a simulator of the system (e.g., a complex network of queues).
- We might be making decisions in a completely unknown environment. These are also called unknown MDPs (e.g., a robot learning to navigate).

Solution techniques which assume explicit transition probabilities and costs (or simulate them) are also called *Model-based methods* while those that solve unknown MDPs are called *Model-free methods*.

Overview

Motivation

For these reasons, practical applications of MDPs were fairly limited till the 90s.

Increasing data, compute power, and more importantly advancements in the area of AI and reinforcement learning (RL) have made it more appealing for solving several problems.

In the next few lectures, we will study a few methods that solve the curse of dimensionality and unknown MDPs.

Overview
Terminology

The notation that we've been using so far is more common in OR and control community. Computer scientists who have largely contributed to the development of RL like to use slightly different terms. For instance,

- *Planning* refers to a dynamic program with a mathematical model and *Learning* refers to unknown MDPs.
- In PI algorithm, *prediction* implies policy evaluation and term *control* is used for policy improvement.
- Modified PI is also called *Generalized* PI.
- System is also referred to as *environment*.
- The DTMC induced by a policy is called *Markov reward process*.

Be aware of these differences when reading papers on RL.

Overview

Terminology

Solution methods in ADP are often also classified as:

- **Off-line methods**
  The idea here is to precompute approximations of the value function or the policy and use them as lookup tables just as before. The only difference is that storing the entire function may be difficult and hence we resort to approximations.

  They are ideal for situations in which the time is available for making decisions at each state is very limited.

- **On-line methods**
  Most of the computation in these problems is done after the state is observed. In addition, we usually also have time constraints on choosing an action (e.g., Tetris, Go). It may still use approximations built from experience.

RL algorithms are also classified as on-policy and off-policy methods. We will talk about these later.

Overview

Techniques

We will switch back to the infinite horizon discounted cost MDP for the discussion on ADP.

Substantial literature on approximate methods for total and average cost MDPs exists but we will not cover them in this course.

There are many approaches for approximately solving MDPs but not all of them have sound theoretical properties.

A certain technique may work well for one problem, but may not provide good solutions for other settings. In that sense, ADP is a bit of an art.

Overview
Techniques

Before we proceed, it is worthwhile to group the methods used in ADP. However, it is very difficulty to construct a taxonomy since the differences between various methods are often not pronounced.

Further, a majority of algorithms are heuristics and mix and match ideas from different solution approaches.

The different *techniques that we will discuss in this class* can be loosely be classified as

- Lookahead methods (Lecture 17)
- Approximations in value space (Lectures 18-20)
- Approximations in policy space (Lecture 21)

Overview
Lookahead Methods

The first set of methods are called lookahead policies, which are simple heuristics that involve limited computation.

As the name suggests, they involve optimization only up to a small number of stages and use some approximate policy or value functions thereafter.

This class of heuristics are also closely related to approximations in value space and are widely used in game playing. They are ideal when limited time is available for choosing actions at each state.

Overview

Approximations in Value Space

This is the most popular approach in ADP. The goal in these methods to find a good approximation to the value function associated with a policy or the optimal value function.

Most algorithms in this class operate in a policy iteration format, although there are value iteration-like algorithms (fitted VI) which we will skip.

The idea is to start with some policy μ and approximate J_μ as $\tilde{J}_\mu$. This is equivalent to the policy evaluation step in PI.

A policy 'improvement' step follows in which a new policy μ' is constructed using

$$\mu'(i) \in \arg\min_{u \in U(i)} \left\{ g(i,u) + \alpha \sum_{j=1}^{n} p_{ij}(u) \tilde{J}_\mu(j) \right\}$$

And this process is repeated. The policy 'improvement' step is also referred to as choosing a greedy policy.

Overview

Approximations in Value Space

Several methods to approximate value functions of a given policy exist. Two most commonly used approaches that we will see in this course are

- **Simulation-based approximation**
  Imagine we have a simulator that mimics our system (e.g., an Arena model of queues). To find $\tilde{J}_\mu(i)$, one could start the system with state i and calculate the discounted cost of a trajectory and repeat this to estimate the the true $J_\mu(i)$ using sample averages.
- **Parametric methods**
  Another option is to represent the value function in a parameterized form and shift our attention to calculating the parameters that gives us the best fit (like regression). These parameters would however be different for different policies.

Non-parametric models for approximating value function have also been shown to work well in the recent past.

Overview

Approximations in Value Space: Simulation-based Approximation

A variety of techniques (MC methods, TD learning) exist for simulation-based approximation. For example, one can use multiple one-step simulated transitions to accurately estimate $\tilde{J}_\mu(i)$.

Alternately, it can directly be used online for a certain 'training period' and the costs incurred can be used to inform the approximate value functions.

However, in online settings the frequency of decisions must be high in order to get reasonable estimates. (This is not a problem with computer simulations since we can speed it up.)

Advantages:

- Ideal for parallelization
- Simple versions offer theoretical guarantees

Disadvantages:

- Some states may not be visited by the policy
- Problems with large state spaces require more memory (Why?)

There are ways to address these drawbacks by combining it with other methods.

Overview

Approximations in Value Space: Parametric Methods

Parametric methods simplify the representation of the value functions and involve selecting an **approximation architecture** and then **training** it.

The approximation architecture selection either involves

- **Selecting the shape of the value function**:
  Is it linear, quadratic, sinusoidal, etc.? Each shape can be characterized by a small parameter vector r. (Think coefficients of a polynomial.)
- **Transforming the state to a lower-dimensional object**:
  Given a state vector, we can extract 'features' which are derived quantities that might influence decisions. The approximate value function can be written as a function (usually linear) of the features and a parameter vector r.

The first technique can also be cast as a lower-dimensional feature representation problem as we will see shortly.

Once the architecture is chosen, the weights r must be tuned to fit $\tilde{J}_\mu(i;r)$ to $J_\mu(i)$. One can write r as r_μ to be a more precise but we will avoid it for brevity.

Overview

Approximations in Value Space: Example 1

For example, imagine a traffic queue. Suppose we want to minimize the overall delay at intersections and are in search of a signal control strategy.

It is likely that as the number of vehicles increases, the cost function increases.

Thus, one guess is to assume that the cost function is quadratic and write $\tilde{J}_\mu(x; r) = r_0 + r_1 x + r_2 x^2$ and train the values of $r_0, r_1,$ and r_2 using simulation.

The original state x can take a large number of values but we have now reduced the problem to finding just three real numbers.

Overview

Approximations in Value Space: Example 1

Similar approximation framework can be extended to a network of queues. Suppose $x = (x_1, \dots, x_N)$ denotes the vector of queue lengths on different sections of a road network.

The combined effects of different queue lengths on the total delay can be possibly written as

$$\tilde{J}_\mu(x; r) = r_0 + \sum_{k=1}^{N} r_k x_k + \sum_{k=1}^{N} \sum_{l=1}^{N} r_{kl} x_k x_l$$

As before, the training part of the approximation involves finding r_k, where $k = 0, \dots, N$ and r_{kl}, where $k = 1, \dots, N$ and $l = 1, \dots, N$.

Thus, instead of dealing with an high-dimensional state space, we reduce the search space to $1 + N + N^2$ dimensions.

In this example, given a state vector x, we can think of the features as $1, \{x_k\}_{k=1}^N, \{x_k x_l\}_{k,l=1}^N$.

Overview

Approximations in Value Space: Example 2

Consider the state space of Tetris. For a 10×20 board, the number of states is of the order 2^{200}.

For each state, we may extract features that we think are important. For example,

- Height of each column
- Difference in heights of successive columns
- Maximum column height
- Number of holes in the wall
- Constant

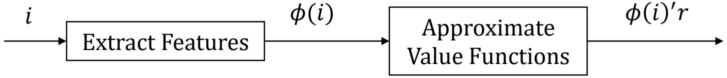
The number of features is about 22 which is extremely low-dimensional compared the the dimension of the original state.

Overview

Approximations in Value Space: Example 2

Thus, for each state i, we can find a column vector $\phi(i)$ which gives the above features. Each feature has an associated tuneable parameter r_k, and a linear approximation architecture looks like

$$\tilde{J}_\mu(i; r) = \phi(i)' r, \; \forall\, i = 1, \dots, n$$



Overview

Approximations in Value Space: Parametric Methods

Advantages:

- Involves tuning few parameters and is memory friendly
- Can be combined with other methods

Disadvantages:

- Highly dependent on choice of the architecture
- Requires domain knowledge and insight

Overview

Approximations in Value Space: Learning Optimal Value Functions

So far, we have discussed how to approximate the value associated with a policy μ. Alternately, we can try to approximate the optimal value functions directly.

We will study the following methods in this course:

- Approximate Linear Programming
- Q-learning

Overview

Approximations in Policy Space

This type of approximation is based on direct search in the space of policies. Approximations in policy space can also be parametric or non-parametric.

We will look at one method called **Policy Gradient** in which the policy $\mu(i, \theta)$ is parameterized by a vector θ and the search space of the optimal policy is reduced to the space of θs.

This method is appropriate when one can guess the optimal policy (threshold or stopping type) or when it would be convenient to implement a parameterized policy. For example,

- (s, S) policies of inventory can be used for larger systems (such as redistribution of shared cars/bicycles)
- Dynamic pricing strategies for airlines or parking.
- Suppose the level of water in a reservoir is x, then the output quantity that has to be released could be $\mu(x; \theta) = \theta_0 + \theta_1 x$.

Overview

Other Options

Another major class of ADP methods involve aggregation of states. If one believes that subset of states are similar in nature, it might make sense to group them and take the same action at all such states. This is also called hard aggregation.

Soft aggregation on the other hand refers to a case where a single state can belong to multiple aggregated states with different probabilities.

Unlike VI, PI, and LPs which work for any MDP, there is no single ADP method that is suited for all problems.

Domain knowledge and information regarding the availability of a mathematical model/simulator, amount of time and compute to make online decisions, etc. are extremely critical.

Lecture Outline

Q-factors

Q-factors

Introduction

A very useful value function-like concept in the context of RL is called Q-factors. Recall that the VI step takes the form

$$J_{k+1}(i) = \min_{u \in U(i)} \left\{ \sum_{j=1}^{n} p_{ij}(u)\Big(g(i,u,j) + \alpha J_k(j)\Big) \right\} \forall\, i = 1, \ldots, n$$

Notice that we have now explicitly assumed that the one-step costs are a function of the future state j and the reason for this will become clear later.

We define new iterates $Q_{k+1}(i,u) \,\forall\, i = 1, \ldots, n, u \in U(i)$ such that

$$Q_{k+1}(i,u) = \sum_{j=1}^{n} p_{ij}(u)\Big(g(i,u,j) + \alpha J_k(j)\Big)$$

Thus, we can rewrite VI algorithm as

$$J_{k+1}(i) = \min_{u \in U(i)} Q_{k+1}(i,u) \,\forall\, i = 1, \ldots, n$$

Can the VI algorithm be re-written in terms of the Q values only?

Q-factors
Value Iteration

Replacing $J_k(j)$ in the second equation on the previous slide with

$$\min_{v \in U(j)} Q_k(j, v)$$

we get

$$Q_{k+1}(i, u) = \sum_{j=1}^{n} p_{ij}(u)\Big(g(i, u, j) + \alpha \min_{v \in U(j)} Q_k(j, v)\Big)$$

This method helps in formulating the Q-learning algorithm which we will revisit later. Does this algorithm take more iterations compared to VI? No.

It however requires more memory as we now have to store a value for each state-action pair.

Q-factors
Bellman Equations

The Bellman equations using Q-factors can be written as

$$J^*(i) = \min_{u \in U(i)} Q^*(i,u) \, \forall \, i = 1, \dots, n$$

where Q^* satisfies

$$Q^*(i,u) = \sum_{j=1}^{n} p_{ij}(u) \Big(g(i,u,j) + \alpha \min_{v \in U(j)} Q^*(j,v) \Big)$$

Notice that $Q^*(i,u) = \sum_{j=1}^{n} p_{ij}(u) \Big(g(i,u,j) + \alpha J^*(j) \Big)$. Thus, one can interpret $Q^*(i,u)$ as the value of taking an action u in state i and behaving optimally thereafter.

Q-factors

Bellman Equations

The optimal policy at state i is the control which minimizes the RHS in the following equation.

$$J^*(i) = \min_{u \in U(i)} Q^*(i, u) \, \forall \, i = 1, \ldots, n$$

In general, given the optimal value functions J^*, to recover the optimal policy, one has to find μ^* such that $T_{\mu^*} J^* = T J^*$.

In that sense, Q-factors are ideal for model-free situations since if an oracle gave you the optimal Q factors, the optimal policies can be derived without the knowledge of one-stage costs and transition probabilities!

Q-factors
Policy Iteration

Given a policy μ, J_μ was the long-run discounted cost of using the policy.

Similarly, we can define $Q_\mu(i,u)$ as the cost of using control u in state i and thereafter following policy μ.

The equations for finding $Q_\mu(i,u)$ parallel that of J_μ but without the minimization operator.

$$Q_\mu(i,u) = \sum_{j=1}^{n} p_{ij}(u)\Big(g(i,u,j) + \alpha J_\mu(j)\Big) \; \forall\, i = 1,\dots,n, u \in U(i)$$

The earlier expression $J_{k+1}(i) = \min_{u\in U(i)} Q_{k+1}(i,u)$ now becomes

$$J_\mu(j) = Q_\mu(j,\mu(j))$$

Can you interpret both sides of the the above equation in words?

Q-factors
Policy Iteration

Thus, one can construct a policy iteration algorithm in which a policy μ_k is evaluated by solving

$$Q_{\mu_k}(i,u) = \sum_{j=1}^{n} p_{ij}(u)\Big(g(i,u,j) + \alpha Q_{\mu_k}(j,\mu_k(j))\Big)\,\forall\, i = 1,\ldots,n, u \in U(i)$$

and the policy improvement is carried out by finding a μ_{k+1} such that

$$Q_{\mu_k}(i,\mu_{k+1}(i)) = \min_{u\in U(i)} Q_{\mu_k}(i,u)\,\forall\, i = 1,\ldots,n$$

Lecture Outline

Lookahead Methods

Lookahead Methods
Introduction

As discussed earlier, lookahead methods are approximation techniques in which some optimization is performed for a small and limited number of iterations/time steps.

For example, suppose we have an approximation $\tilde{J}$ of the optimal value function J^*. Then a *one-step lookahead policy* μ' is defined as

$$\mu'(i) \in \arg \min_{u \in U(i)} \left\{ g(i,u) + \alpha \sum_{j=1}^{n} p_{ij}(u) \tilde{J}(j) \right\}$$

We expect that the lookahead policy μ' to be a good approximation to the optimal policy μ^*.

How is the above equation different from regular PI? It is like the policy improvement step but $\tilde{J}$ can be any approximate value function not necessary J of some policy μ.

Lookahead Methods

Two-step Lookahead Policies

Likewise we can define a two-step lookahead policy (and in general *m*-step lookahead policy in the following way.

Suppose $\tilde{\tilde{J}}$ is an approximation of J^*. Then we first define a new function $\tilde{J}$ using

$$\tilde{J}(i) = \min_{u \in U(i)} \left\{ g(i,u) + \alpha \sum_{j=1}^{n} p_{ij}(u) \tilde{\tilde{J}}(i) \right\}$$

and construct a policy μ'' in the same way as before.

$$\mu''(i) \in \arg \min_{u \in U(i)} \left\{ g(i,u) + \alpha \sum_{j=1}^{n} p_{ij}(u) \tilde{J}(j) \right\}$$

One way to interpret this technique is that we are solving a finite horizon DP with 2 stages and terminal cost vector $\tilde{\tilde{J}}$.

Lookahead Methods

Rollout Methods

In the previous example, how do we know an approximation of the optimal policy? This might be difficult, but if we have one, we can use a one-step lookahead policy to improve it.

Rollout is essentially a one-step lookahead method where $\tilde{J}$ is replaced with J_μ where μ is some policy that is easy to construct (say a myopic policy). It is also called the *base policy*.

Thus, rollout is simply one step of policy improvement from some base policy. Another way to interpret it is that we take the optimal action for one step assuming that we will revert to the base policy thereafter.

Lookahead Methods

Rollout Methods

In practice, J_μ may also be difficult to estimate and could be approximated using the other simulation or parameteric methods that we will discuss. Rollout heuristics are effective when there is limited time for computation.

Lookahead methods often improve policies since convergence of PI is usually fast and a single iteration can offer significant improvement. Multistep rollout can be be similarly designed along the lines of multistep lookahead methods.

Here's a demonstration of the algorithm on Solitaire:

- Yan, X., Diaconis, P., Rusmevichientong, P., & Roy, B. V. (2005). Solitaire: Man versus machine. In Advances in Neural Information Processing Systems (pp. 1553-1560).

Lookahead Methods

Rolling Horizon Approach

Consider the m-stage rollout or lookahead policy. Once the policy is obtained, we've assumed that it will be continued to be used thereafter.

However, one can perform more policy iterations by estimating an approximation of $J_{\mu'}$ and reapplying the rollout method.

However, time constraints on calculations may prevent us from doing so. One option is to use a rolling horizon approach in which the m-stage rollout policy is used for H periods after which it is updated using another rollout.

Lookahead Methods

Monte Carlo Tree Search

A more sophisticated version of lookahead methods is Monte Carlo Tree Search (MCTS) and it has been key to the success of programs such as AlphaGo.

In traditional rollout, we are evaluating the cost of using an action u in state i followed by the base policy μ.

This is equivalent to finding Q-factors $Q(i, u)$ and selecting the control which gives the best Q-value.

As mentioned earlier, evaluating J_μ or $Q_\mu(i, u)$ is usually done approximately by simulating trajectories from each state i, taking various actions u and estimating the total discounted costs.

Lookahead Methods

Monte Carlo Tree Search

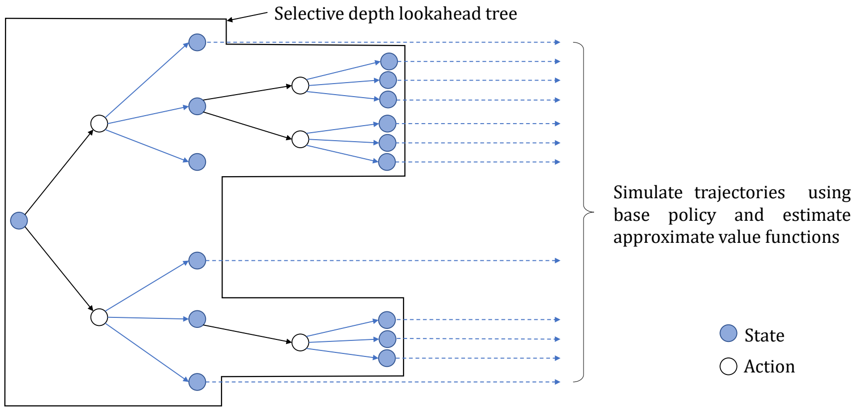
As we construct sample averages of the Q values we may realize that,

- Some controls are inferior compared to others and may not be worth exploring. (Since their contribution to the sample average may be minimal.)
- Controls that are promising may be work exploring using a multistep lookahead framework for a more accurate estimate.

These observations, result in creating a selective depth lookahead tree over which backward induction-style algorithms are run while evaluating the cost of the leaf nodes using the base policy and simulation.

Lookahead Methods

Monte Carlo Tree Search


Selective depth lookahead tree
Simulate trajectories using base policy and estimate approximate value functions
State
Action

Your Moment of Zen

THIS IS YOUR MACHINE LEARNING SYSTEM?
YUP! YOU POUR THE DATA INTO THIS BIG PILE OF LINEAR ALGEBRA, THEN COLLECT THE ANSWERS ON THE OTHER SIDE.
WHAT IF THE ANSWERS ARE WRONG?
JUST STIR THE PILE UNTIL THEY START LOOKING RIGHT.
DATA
ANSWERS