CRICOS
00213J

ENN582: Reinforcement Learning and Optimal Control

Approximate Dynamic Programming

Daniel E. Quevedo[‡]

VERSION 1.

Abstract

The purpose of these notes is to present methods to approximate Dynamic Programming algorithms. We will give special emphasis to approximations in value space. A more comprehensive exposition can be found in books, such as [5].

1 Key Learning Objectives

In conjunction with the practical, you will learn to

- formulate and implement rollout methods
- understand the concept of certainty equivalent control
- develop algorithms for parametric approximations in value space

[‡]Please report errors within this document to daniel.quevedo@qut.edu.au

2 Approximations in Value Space

We recall finite horizon optimal control of deterministic systems $x_{k+1} = f_k(x_k, u_k)$ and the cost-to-go functions

$$J_k(x_k) = \min_{u_k,\dots,u_{N-1}} \left\{ g_N(x_N) + \sum_{m=k}^{N-1} g_m(x_m, u_m) \right\}.$$

The Dynamic Programming algorithm is given by:

$$J_k(x_k) = \min_{u_k \in \mathbb{U}} \Big\{ g_k(x_k, u_k) + J_{k+1}(f_k(x_k, u_k)) \Big\}, \quad k = 0, 1, \dots, N-1, \tag{1}$$

starting from $J_N(x_N) = g_N(x_N)$.

The computation of the cost-to-go function sequence $J_N, J_{N-1}, \dots J_0$ provides the optimal policy π^* through the offline minimisations in (1). This policy can then be used online once the state values $x_0, x_1, \dots x_N$ become available. Alternatively, one can also store the cost-to-go functions for all possible state values $x_k \in \mathbb{X}$ and all $k \in \{0, 1 \dots, N\}$ and perform the optimisations online. Unfortunately, for large state spaces or long horizons, these methods are often impractical.

Approximations in value space and rollout algorithms

To simplify computations, albeit incurring loss of optimality, instead of storing the cost-to-go-functions, one may alternatively store suitable approximations, say $\tilde{J}_k$, and use $\tilde{J}_k$ instead of J_k to find the control values:

$$\tilde{u}_k \in \arg\min_{u_k \in \mathbb{U}} \Big\{ g_k(x_k, u_k) + \tilde{J}_{k+1}(f_k(x_k, u_k)) \Big\}, \quad k = 0, 1, \dots, N-1, \quad \tilde{J}_N = g_N. \tag{2}$$

Note that, when performing such an approximation in value space, the sub-optimal control sequence $\{\tilde{u}_0, \tilde{u}_1, \dots, \tilde{u}_{N-1}\}$ is obtained by going forward in time. No backward calculations are needed, since $\tilde{J}_k$ are assumed available.

A key issue is how to find suitable approximations to the cost to go functions. Of particular interest to us is the *rollout* algorithm, wherein $\tilde{J}_k(x_k)$ are obtained by running a heuristic ("base") policy from the state x_k to the end of the horizon.[1] This heuristic policy generates a sequence of tentative control actions, say $\{\check{u}_k, \check{u}_{k+1}, \dots, \check{u}_{N-1}\}$, and corresponding states $\{\check{x}_{k+1}, \check{x}_{k+2}, \dots, \check{x}_N\}$. The approximate cost-to-go function is then computed by

$$\tilde{J}_k(x_k) = g_k(x_k, \check{u}_k) + g_{k+1}(\check{x}_{k+1}, \check{u}_{k+1}) + \dots + g_{N-1}(\check{x}_{N-1}, \check{u}_{N-1}) + g_N(\check{x}_N).$$

Then, using (2), the rollout algorithm generates a suboptimal policy, which is called rollout policy. Typically, the rollout policy has a policy improvement property: The total cost obtained by the above approximation in value space method is less or equal to the total cost obtained with the base policy.[2]

[1]The term "rollout" was coined by Gerald Tesauro in 1996 in the context of rolling dice in a backgammon playing computer program. A given backgammon position was evaluated by "rolling out" many games starting from that position, and taking averages.

[2]For a detailed analysis, see [5, Section 2.4].

Example: Travelling Salesman Problem

Consider N cities and find the minimum cost tour that visits each city exactly once and returns to the city you started from. Assuming that each pair of cities is directly connected by a road[3], a simple nearest neighbour heuristic can be formulated as follows:

- Start from an arbitrary city
- The next city visited is the one with minimum distance from the current city (and that has not been previously visited)

To implement the rollout algorithm using this nearest neighbour heuristic as a base policy, we proceed as follows:

Step 1 For each node not yet visited, assume the nearest neighbour heuristic is run afterwards, and compute the cost of the corresponding tour.

Step 2 Choose the next city as the one that gives best tour (the lowest cost) from the ones computed in Step 1 above.
Provided there are still new nodes to be visited, go to Step 1.

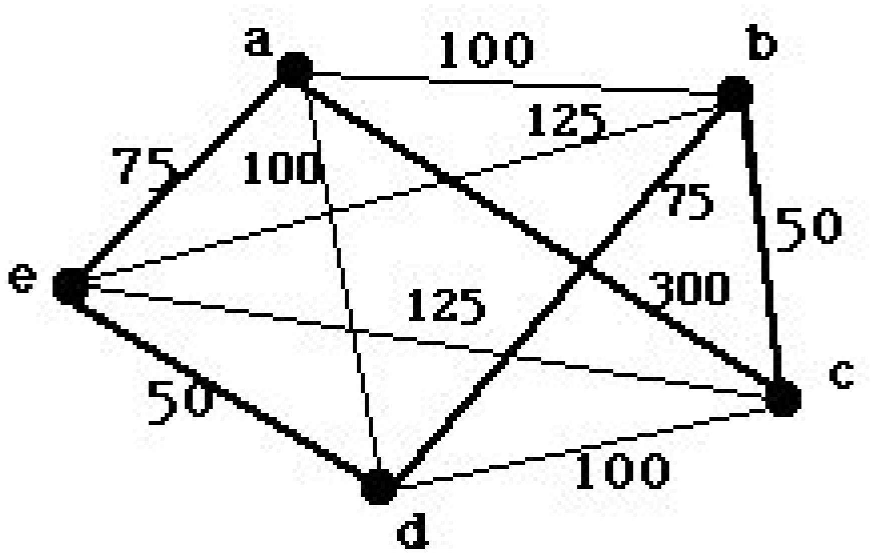

Figure 1: Travelling Salesman Problem

Consider the travelling salesman problem for the graph shown above and let a be the node with which we start and end the tour. An optimal tour can be shown to be *abcdea*, with length 375. In contrast, the nearest neighbour (*N.N.*) heuristic yields the tour *aedbca* with length 550.

[3]The underlying graph is complete.

The rollout algorithm with the nearest neighbour heuristic as a base policy provides:

- 1st stage:
  $ab\underbrace{cdea}_{N.N.}$ length = 375
  $ac\underbrace{bdea}_{N.N.}$ length = 550
  $ad\underbrace{ebca}_{N.N.}$ length = 625
  $ae\underbrace{dbca}_{N.N.}$ length = 550
  So the next node should be b.

- 2nd stage:
  $abc\underbrace{dea}_{N.N.}$ length = 375
  $abd\underbrace{eca}_{N.N.}$ length = 650
  $abe\underbrace{dca}_{N.N.}$ length = 675
  Thus, the next node is c

- 3rd stage:
  $abcd\underbrace{ea}_{N.N.}$ length = 375
  $abce\underbrace{da}_{N.N.}$ length = 425
  The next node is d

- 4th stage:
  $abcdea$ = tour computed by rollout, with length 375

We note that in this example the rollout algorithm gives the optimal solution. This is not always the case.

Stochastic Systems

For stochastic systems of the form

$$x_{k+1} = f_k(x_k, u_k, w_k), \quad k \in \mathbb{N}_0,$$

where the DP recursions are given by

$$J_k(x_k) = \min_{u_k \in \mathbb{U}} \underset{w_k}{E} \Big\{ g_k(x_k, u_k, w_k) + J_{k+1}(f_k(x_k, u_k, w_k)) \Big\}, \quad k = 0, 1, \ldots, N-1,$$

the counterpart to the approximation in value space expression (2) becomes

$$J_k(x_k) = \min_{u_k \in \mathbb{U}} \mathop{E}_{w_k} \Big\{ g_k(x_k, u_k, w_k) + \tilde{J}_{k+1}(f_k(x_k, u_k, w_k)) \Big\}, \quad k = 0, 1, \ldots, N-1. \tag{3}$$

In addition to the problem of approximating the cost-to-go function, which for example can be one using rollout of a base policy, we are often also faced with approximating the expectation operator. The simplest approach here is to represent each w_k by a single value, say $\bar{w}_k$, rather than using its distribution.[4] This reduces the problem to the deterministic one in (2):

$$J_k(x_k) = \min_{u_k \in \mathbb{U}} \{ g_k(x_k, u_k, \bar{w}_k) + \tilde{J}_{k+1}(f_k(x_k, u_k, \bar{w}_k)) \}, \quad k = 0, 1, \ldots, N-1. \tag{4}$$

Such an approximation is called Certainty Equivalent Control and underlies, e.g., Linear Quadratic Gaussian Control (where it is optimal) [1].

Multistep Lookahead

Expressions (2) and (3) look one step ahead, from step k to $k+1$. A natural extension is *multistep lookahead*:

$$J_k(x_k) = \min_{u_k, \mu_{k+1}, \ldots, \mu_{k+\ell-1}} \mathop{E}_{w_k, \ldots, w_{k+\ell-1}} \left\{ g_k(x_k, u_k, w_k) + \sum_{j=k+1}^{k+\ell-1} g_j(x_j, \mu_j(x_j), w_j) + \tilde{J}_{k+\ell}(x_{k+\ell}) \right\}. \tag{5}$$

Such an optimisation yields an optimal sequence $\{\tilde{u}_k, \tilde{\mu}_{k+1}, \ldots, \tilde{\mu}_{k+\ell-1}\}$. Similar to the receding horizon principle used in Model Predictive control, one only uses the first element, $\tilde{u}_k$, and then repeats the optimisation once $x_{k+1} = f_k(x_k, \tilde{u}_k, w_k)$ becomes available.

The reason for using multistep lookahead is that large values of ℓ, can compensate for poor cost-to-go approximations. Unfortunately, large values of ℓ lead to increased computational complexity. Hence, for a given cost-to-go approximation, there exists a trade-off between computations and resulting control performance, similar to the one encountered in MPC.

Starting from a base policy, say $\check{\mu}$, and some terminal cost approximation, one can also formulate a truncated rollout with terminal cost approximation controller. This can be achieved by setting

$$J_k(x_k) = \min_{u_k \in \mathbb{U}} \mathop{E}_{w_k, \ldots, w_{k+\ell-1}} \left\{ g_k(x_k, u_k, w_k) + \sum_{j=k+1}^{k+\ell-1} g_j(x_j, \check{\mu}_j(x_j), w_j) + \tilde{J}_{k+\ell}(x_{k+\ell}) \right\}.$$

In many problem instances, and provided ℓ is chosen large enough, setting $\tilde{J}_{k+\ell}(x) = 0$, for all $x \in \mathbb{X}$, may give good results.

[4]Often one chooses $\bar{w}_k = E\{w_k\}$.

Infinite Horizons

For infinite horizon problems, the approximation procedure is similar. In particular, in the stationary and stochastic case with cost function

$$J_\pi(x) = \lim_{N\to\infty} \underset{w_k}{E}\left\{ \sum_{k=0}^{N-1} \alpha^k g(x_k, \mu(x_k), w_k) \right\}, \quad x_0 = x, \quad \pi = \{\mu, \mu, \dots\},$$

we recall Bellman's Equation and use

$$\tilde{\mu}(x) \in \arg\min_{u\in\mathbb{U}} \underset{w}{E}\big\{ g(x,u,w) + \alpha \tilde{J}\big(f(x,u,w)\big)\big\}, \quad \forall x \in \mathbb{X}, \tag{6}$$

where $\tilde{J}$ approximates the infinite-horizon cost J_{π^*}. This approximation can, for example, be computed using multi-step rollout of a base policy.

3 Parametric Approximation in Value Space

As an alternative to rollout methods for approximating the cost-to go functions J_k (or the infinite horizon cost), one can directly adopt a parameterised class of functions, i.e., set

$$\tilde{J}_k(x_k, r_k), \tag{7}$$

where

$$r_k = \begin{bmatrix} r_{1,k} & r_{2,k} & \dots r_{m,k} \end{bmatrix}^T$$

are the parameters to be tuned. Within this *approximation architecture*, parameters are optimised (or "learned") using some algorithm. Many approximation architectures can be used, including polynomials, wavelets, Gaussian basis functions and neural networks.

Feature-based architectures

In general, cost functions are complicated nonlinear mappings and finding good parameters r_k in the architecture (7) is difficult. This problem can often be circumvented if one extracts *features* of the state values to form the *feature vectors*

$$\phi_k(x_k) = \begin{bmatrix} \phi_{1,k}(x_k) & \phi_{2,k}(x_k) & \dots \phi_{m,k}(x_k) \end{bmatrix}^T$$

Thus, instead of directly using the states x_k as inputs in (7), the cost approximation depends on the state x_k through the feature vector $\phi_k(x_k)$:

$$\tilde{J}_k(x_k, r_k) = \hat{J}_k(\phi_k(x_k), r_k). \tag{8}$$

Feature vectors are often handcrafted using heuristics and can involve state aggregation and state space partitioning. There also exist method that generate features automatically from data, see [5, Chapter 4] and [6, Chapter 9]. If the feature vectors encode much of the nonlinear aspects, then simpler approximation architectures can be used and finding parameters becomes less involved.

In particular, with well-designed feature vectors $\phi_k(x_k)$, a good approximation of the cost is often provided by the linear feature-based architecture:

$$\hat{J}_k(\phi_k(x_k), r_k) = \sum_{\ell=1}^{m} r_{\ell,k}\phi_{\ell,k}(x_k) = r_k^T \phi_k(x_k). \tag{9}$$

In this case, the architecture spans the subspace generated by the feature vectors and the parameters in r_k are commonly called *weights*.

Training of architectures

Training of an architecture of the form (7) (or similarly (8)) amounts to choosing parameter vectors r_k. This can be done sequentially, starting from the end of the horizon and proceeding backwards. For that purpose, one can collect a (large) *training set* of state-cost pairs (x_k^s, β_k^s), $s = 1, \ldots, q$, for each stage k. Given that the cost satisfies

$$\beta_k^s = \min_{u_k \in \mathbb{U}} \mathop{E}_{w_k} \Big\{ g_k(x_k^s, u_k, w_k) + \tilde{J}_{k+1}(f_k(x_k^s, u_k, w_k), r_{k+1}) \Big\}, \quad k = 0, 1, \ldots, N-1, \tag{10}$$

having found r_{k+1}, one can find r_k by solving the least squares problem

$$\min_r \Phi(r), \quad \text{where} \quad \Phi(r) = \frac{1}{2}\sum_{s=1}^{q} (\tilde{J}(x_k^s, r) - \beta_k^s)^2. \tag{11}$$

More generally, $\tilde{J}(x, r)$ seeks to approximate some target cost function $J(x)$, and β^s are the sampled costs $J(x^s)$ plus some noise or error.

As an alternative, one can also use iterative methods, such as gradient descent to find each parameter vector r_k.. In its basic form, gradient descent generates a sequence of iterates r_k^ℓ, starting from some initial guess, say r_k^0. The recursions are given by:

$$r_k^{\ell+1} = r_k^\ell - \gamma^\ell \nabla\Phi(r_k^\ell) = r_k^\ell - \frac{\gamma^\ell}{2}\sum_{s=1}^{q} \nabla(\tilde{J}(x_k^s, r_k^\ell) - \beta_k^s)^2 \tag{12}$$

and follow the negative of the gradient direction with a time-varying step-size (or *learning rate*) γ^ℓ. Whilst there don't exist general guidelines on how to tune the step-sizes for all problem instances, step-sizes are typically chosen as a decreasing sequence; see, e.g., [2].

We note that in (12) the entire batch of q state-cost pairs is used to evaluate the gradient at each iteration step ℓ. Interestingly, often the use of only one (or a few) randomly chosen state-cost pairs per iteration will lead to faster convergence and reduced computations. Such methods are called stochastic gradient descent and are widely used in practice; see [6, Section 9.3].

Training of the linear feature-based architecture

In general, (11) constitutes a non-convex optimisation problem and is therefore difficult to solve. However, for the linear architecture in (9), the loss function Φ becomes convex quadratic

$$\Phi(r) = \frac{1}{2}\sum_{s=1}^{q}(r^T\phi_k(x_k^s) - \beta_k^s)^2$$

and we can obtain a closed form solution by setting its gradient to zero:

$$\nabla\Phi(r) = \sum_{s=1}^{q}\phi_k(x_k^s)(r^T\phi_k(x_k^s) - \beta_k^s) = 0 \Longleftrightarrow \sum_{s=1}^{q}\phi_k(x_k^s)\phi_k(x_k^s)^T r = \sum_{s=1}^{q}\phi_k(x_k^s)\beta_k^s$$

which yields the optimal parameter vector

$$\hat{r}_k = \left(\sum_{s=1}^{q}\phi_k(x_k^s)\phi_k(x_k^s)^T\right)^{-1}\sum_{s=1}^{q}\beta_k^s\phi_k(x_k^s). \tag{13}$$

To avoid issues related to matrix inversions needed in (13), one can also adopt gradient descent algorithms. In its basic form (12), gradient descent for the linear architecture (9) reduces to:

$$r_k^{\ell+1} = r_k^\ell - \gamma^\ell\nabla\Phi(r_k^\ell) = r_k^\ell - \gamma^\ell\sum_{s=1}^{q}\phi(x_k^s)((r_k^\ell)^T\phi_k(x_k^s) - \beta_k^s).$$

References

[1] B. D. O. Anderson and J. Moore, *Linear Optimal Control.* Englewood Cliffs, NJ: Prentice Hall, 1971.

[2] D. P. Bertsekas, *Nonlinear Programming.* Athena Scientific, 1999.

[3] D. P. Bertsekas, *Dynamic Programming and Optimal Control*, vol. 2. Belmont, MA: Athena Scientific, 2007.

[4] D. P. Bertsekas, *Reinforcement Learning and Optimal Control.* Athena Scientific, 2019.

[5] D. P. Bertsekas, *Rollout, Policy Iteration, and Distributed Reinforcement Learning.* Athena Scientific, 2020.

[6] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction.* MIT Press, 2nd ed., 2018.