



ENN582: Reinforcement Learning and Optimal Control
Approximate Dynamic Programming

Jason J. Ford[‡]

Original notes by Daniel E. Quevedo, adjustments by Jason J. Ford.

VERSION 1.1.

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING:

This material has been reproduced and communicated to you by or on behalf of The Queensland University of Technology pursuant to Part VB of The Copyright Act 1968 (The Act).

The material in this communication may be subject to copyright under The Act. Any further copying or communication of this material by you may be the subject of copyright protection under The Act.

Do not remove this notice.

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 j2.ford@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:

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

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, heuristic cost-to-go, 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 $\tilde{J}_{k+1}$, and use $\tilde{J}_{k+1}$ instead of J_{k+1} to find the control values:

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

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 $\tilde{J}_{k+1}$ are assumed available.

Heuristic cost-to-go functions: A key issue is how to find suitable approximations to the cost to go functions, $\tilde{J}_{k+1}$. These might be any heuristic cost-to-go created from your intuition. If $\tilde{J}_{k+1} = J_{k+1}$, then the $\tilde{u}_k$ will correspond to the optimal control. Hence, we would like approximate cost-to-go-functions $\tilde{J}_{k+1}$ to be close to J_{k+1} .

Heuristic (“base”) policy approach: Of particular interest to us is the *rollout* algorithm, wherein $\tilde{J}_{k+1}(x_k)$ are obtained by running a heuristic (“base”) policy from the state x_{k+1} to the end of the horizon.¹ This heuristic policy generates a sequence of tentative control actions, say $\{\tilde{u}_{k+1}, \tilde{u}_{k+2}, \dots, \tilde{u}_{N-1}\}$,

¹The 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.

and corresponding states $\{\tilde{x}_{k+2}, \tilde{x}_{k+3}, \dots, \tilde{x}_N\}$. The approximate cost-to-go function is then computed by

$$\tilde{J}_{k+1}(x_{k+1}) = g_{k+1}(x_{k+1}, \tilde{u}_{k+1}) + g_{k+2}(\tilde{x}_{k+2}, \tilde{u}_{k+2}) + \dots + g_{N-1}(\tilde{x}_{N-1}, \tilde{u}_{N-1}) + g_N(\tilde{x}_N).$$

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

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 road³, 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.

Consider the travelling salesman problem for the graph shown in Figure 1 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.

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

³The underlying graph is complete.

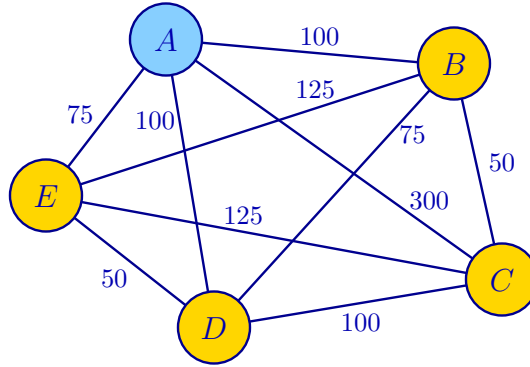


Figure 1: Travelling Salesman Problem

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

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

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

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

In addition to the problem of approximating the cost-to-go function, which for example can be done 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):

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

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*:

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

This yields a sub-optimal sequence (for the original problem), say $\{\tilde{u}_k, \tilde{\mu}_{k+1}, \dots, \tilde{\mu}_{k+\ell-1}\}$. 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.

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

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

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

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.

Infinite Horizons

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

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

we recall Bellman's Equation and use

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

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.

Recall that for shortest path problems, where the goal and stop condition can be reached in finite time, we can consider $\alpha = 1$.

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

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

where

$$r_k = [r_{1,k} \quad r_{2,k} \quad \dots \quad r_{m,k}]^T$$

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*

$$\phi_k(x_k) = [\phi_{1,k}(x_k) \quad \phi_{2,k}(x_k) \quad \dots \phi_{m,k}(x_k)]^T$$

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 $\phi_k(x_k)$:

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

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:

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

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

For infinite horizon optimal control problems we can consider the time-invariant features $\phi(x_k)$, weights r , and the approximation

$$\hat{J}(\phi(x_k), r) = \sum_{\ell=1}^m r_{\ell} \phi_{\ell}(x_k) = r^T \phi(x_k). \quad (10)$$

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) , for each iteration stage $s = 1, \dots, q$. That is, sampling at various $k \in \{0, N-1\}$, we have β_k^s which are, possibly noisy, cost-to-go estimates from x_k^s (typically sample number $q \gg N$ if we want good approximation). Given that the cost satisfies

$$\beta_k^s = \min_{u_k \in \mathbb{U}} E_{w_k} \left\{ g_k(x_k^s, u_k, w_k) + \tilde{J}_{k+1}(f_k(x_k^s, u_k, w_k), r_{k+1}^s) \right\}, \quad k = 0, 1, \dots, N-1. \quad (11)$$

having knowledge of estimate r_{k+1}^s , one can find next estimate for r_k^s by solving the least squares problem

$$\min_r \Phi(r), \quad \text{where} \quad \Phi(r) = \frac{1}{2} \sum_{s=1}^q (\tilde{J}_k(x_k^s, r) - \beta_k^s)^2. \quad (12)$$

Given the time dependence of features, weights and optimal cost, this might require large training set, at different k . That is, we need to solve a version of (12) for each r_k , for each fixed $k \in \{0, N-1\}$.

Things are slightly easier in the infinite horizon setting, where features, weights and optimal cost are time-invariant, we can write

$$\beta_k^s = \min_{u_k \in \mathbb{U}} E_{w_k} \left\{ g(x_k^s, u_k, w_k) + \tilde{J}(f(x_k^s, u_k, w_k), r^s) \right\}, \quad k = 0, 1, \dots, N-1. \quad (13)$$

having knowledge of estimate r^s , one can find next estimate for r^{s+1} by solving the least squares problem

$$\min_r \Phi(r), \quad \text{where} \quad \Phi(r) = \frac{1}{2} \sum_{s=1}^q (\tilde{J}(x_k^s, r) - \beta_k^s)^2. \quad (14)$$

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:

$$r_k^{\ell+1} = r_k^\ell - \gamma^\ell \nabla \Phi(r_k^\ell) = r_k^\ell - \frac{\gamma^\ell}{2} \sum_{s=1}^q \nabla (\tilde{J}_k(x_k^s, r_k^\ell) - \beta_k^s)^2 \quad (15)$$

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]. Again, we need to solve a version of (12) for each r_k , for $k \in \{0, N-1\}$ (note, ℓ is iteration number, not time $k \in \{0, 1, \dots, N-1\}$ which remain fixed in (15)).

In the infinite horizon setting, where features, weights and optimal cost are time-invariant, we can instead write

$$r^{\ell+1} = r^\ell - \gamma^\ell \nabla \Phi(r^\ell) = r^\ell - \frac{\gamma^\ell}{2} \sum_{s=1}^q \nabla (\tilde{J}(x_k^s, r^\ell) - \beta_k^s)^2 \quad (16)$$

We note that in (15) 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, (12) 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:

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

which yields the optimal parameter vector

$$\hat{r}_k = \left(\sum_{s=1}^q \phi_k(x_k^s) \phi_k(x_k^s)^T \right)^{-1} \sum_{s=1}^q \beta_k^s \phi_k(x_k^s). \quad (17)$$

To avoid issues related to matrix inversions needed in (17), one can also adopt gradient descent algorithms. In its basic form (15), gradient descent for the linear architecture (9) reduces to:

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

In the infinite horizon setting, where features, weights and optimal cost are time-invariant, we can instead write

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

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.