Using Local Trajectory Optimizers To Speed Up Global Optimization In Dynamic Programming

Christopher G. Atkeson
College of Computing
Georgia Institute of Technology
801 Atlantic Drive
Atlanta, GA, 30332-0280
cga@cc.gatech.edu

July 28, 1995

Abstract

Dynamic programming provides a methodology to develop planners and controllers for nonlinear systems. However, general dynamic programming is computationally intractable. We have developed procedures that allow more complex planning and control problems to be solved. We use second order local trajectory optimization to generate locally optimal plans and local models of the value function and its derivatives. We maintain global consistency of the local models of the value function, guaranteeing that our locally optimal plans are actually globally optimal, up to the resolution of our search procedures.

1 Introduction

Learning to do the right thing at each instant in situations that evolve over time is difficult, as the future cost of actions chosen now may not be obvious immediately, and may only become clear with time. Value functions are a representational tool that makes the consequences of actions explicit. Value functions are difficult to learn directly, but they can be built up from learned models of the dynamics of the world and the cost function. This paper focuses on how fast optimizers that only produce locally optimal answers can play a useful role in speeding up the process of computing or learning a globally optimal value function.

Consider a system with dynamics $\mathbf{x}_{k+1} = \mathbf{f}(\mathbf{x}_k, \mathbf{u}_k)$ and a cost function $L(\mathbf{x}_k, \mathbf{u}_k)$, where $\mathbf{x}$ is the state of the system and $\mathbf{u}$ is a vector of actions or controls. The subscript k serves as a time index, but will be dropped in the equations that follow. A goal of reinforcement learning and optimal control is to find a policy that minimizes the total cost, which is the sum of the costs for each time step. One approach to doing this is to construct an optimal value function, $V(\mathbf{x})$. The value of this value function at a state $\mathbf{x}$ is the sum of all future costs, given that the system started in state $\mathbf{x}$ and followed the optimal policy $\mathbf{P}(\mathbf{x})$ (chose optimal actions at each time step as a function of the state). A local planner or controller can choose globally optimal actions if it knew the future cost of each action. This cost is simply the sum of the cost of taking the action right now and the future cost of the state that the action leads to, which is given by the value function.

$$\mathbf{u}^* = \arg\min_{\mathbf{u}} \left(L(\mathbf{x}, \mathbf{u}) + V(\mathbf{f}(\mathbf{x}, \mathbf{u})) \right) \tag{1}$$

Value functions are difficult to learn. The environment does not provide training examples that pair states with their optimal cost $(\mathbf{x}, V(\mathbf{x}))$. In fact, it seems that the optimal policy depends on the optimal value function, which in turn depends on the optimal policy. Algorithms to compute value functions typically iteratively refine a candidate value function and/or a corresponding policy (dynamic programming). These algorithms are usually expensive. We use local optimization to generate locally optimal plans and local models of the value function and its derivatives. We maintain global consistency of the local models of the value function, guaranteeing that our locally optimal plans are actually globally optimal, up to the resolution of our search procedures.

2 Local Trajectory Optimization

We base our local optimization process on dynamic programming within a tube surrounding our current best estimate of a locally optimal trajectory (Dyer and McReynolds 1970, Jacobson and Mayne 1970). We have a local quadratic model of the cost to get to the goal (V) at each time step along the optimal trajectory (assume a time step index k in everything below unless otherwise indicated):

$$V(\mathbf{x}) \approx V_0 + V_x \mathbf{x} + \frac{1}{2}\mathbf{x}^{\mathrm{T}} V_{xx} \mathbf{x} \tag{2}$$

A locally optimal policy can be computed using local models of the plant (in this case local linear models) at each time step along the trajectory:

$$\mathbf{x}_{k+1} = \mathbf{f}(\mathbf{x}, \mathbf{u}) \approx \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u} + \mathbf{c} \tag{3}$$

and local quadratic models of the one step cost at each time step along the trajectory:

$$L(\mathbf{x}, \mathbf{u}) \approx \frac{1}{2}\mathbf{x}^{\mathrm{T}}\mathbf{Q}\mathbf{x} + \frac{1}{2}\mathbf{u}^{\mathrm{T}}\mathbf{R}\mathbf{u} + \mathbf{x}^{\mathrm{T}}\mathbf{S}\mathbf{u} + \mathbf{t}^{\mathrm{T}}\mathbf{u} \tag{4}$$

At each point along the trajectory the optimal policy is given by:

$$\begin{aligned} \mathbf{u}^{opt} = &-(\mathbf{R} + \mathbf{B}^{\mathrm{T}} V_{xx} \mathbf{B})^{-1} \times \\ &(\mathbf{B}^{\mathrm{T}} V_{xx} \mathbf{A}\mathbf{x} + \mathbf{S}^{\mathrm{T}}\mathbf{x} + \mathbf{B}^{\mathrm{T}} V_{xx} \mathbf{c} + V_x \mathbf{B} + \mathbf{t}) \end{aligned}$$

One can integrate the plant dynamics forward in time based on the above policy, and then integrate the value functions and its first and second spatial derivatives backwards in time to compute an improved value function, policy, and trajectory.

For a one step cost of the form:

$$L(\mathbf{x}, \mathbf{u}) \approx \frac{1}{2}(\mathbf{x} - \mathbf{x}_d)^{\mathrm{T}}\mathbf{Q}(\mathbf{x} - \mathbf{x}_d) + \frac{1}{2}(\mathbf{u} - \mathbf{u}_d)^{\mathrm{T}}\mathbf{R}(\mathbf{u} - \mathbf{u}_d) + (\mathbf{x} - \mathbf{x}_d)^{\mathrm{T}}\mathbf{S}(\mathbf{u} - \mathbf{u}_d)$$

the backward sweep takes the following form (in discrete time):

$$Z_x = V_x\mathbf{A} + \mathbf{Q}(\mathbf{x} - \mathbf{x}_d) \tag{5}$$

$$Z_u = V_x\mathbf{B} + \mathbf{R}(\mathbf{u} - \mathbf{u}_d) \tag{6}$$

$$Z_{xx} = \mathbf{A}^{\mathrm{T}}V_{xx}\mathbf{A} + \mathbf{Q} \tag{7}$$

$$Z_{ux} = \mathbf{B}^{\mathrm{T}}V_{xx}\mathbf{A} + \mathbf{S} \tag{8}$$

$$Z_{uu} = \mathbf{B}^{\mathrm{T}}V_{xx}\mathbf{B} + \mathbf{R} \tag{9}$$

$$\mathbf{K} = Z_{uu}^{-1}Z_{ux} \tag{10}$$

$$V_{x_{k-1}} = Z_x - Z_u\mathbf{K} \tag{11}$$

$$V_{xx_{k-1}} = Z_{xx} - Z_{xu}\mathbf{K} \tag{12}$$

3 Using Local Trajectory Optimization With Dynamic Programming

A typical implementation of dynamic programming in continuous state spaces discretizes the state space into cells, and assigns a fixed control action to each cell. Larson's state increment dynamic programming (Larson 1968) is a good example of this type of approach.

We want to minimize the number of cells used in dynamic programming by making the cells as large as possible. Combining local trajectory optimization with dynamic programming allows us to greatly reduce the resolution of the grid on which we do dynamic programming and still correctly estimate the cost to get to the goal from different parts of the space. By performing the local trajectory optimizations on a grid and forcing adjacent trajectories to be consistent, this local optimization process becomes a global optimization process. Forcing adjacent trajectories to be consistent means requiring that all trajectories can be generated from a single underlying policy. A trajectory can be made consistent with a neighbor by using the neighboring trajectory as an initial trajectory in the local optimization process, or by using the value function from the neighboring trajectory to generate the initial trajectory in the local optimization process. Each grid element stores the trajectory that starts at that point and achieves the lowest cost.

Using locally optimal trajectories which go all the way to the goal as building blocks for our dynamic programming algorithm allows us to avoid the problem of correctly interpolating the cost to get to the goal function on a sparse grid. Instead, the cost to get to the goal is measured directly on the optimal trajectory from each node to the goal. We can use a much sparser grid than state increment dynamic programming and still converge.

4 Adaptive Grids Based On Constant Cost Contours

We can limit the search by "growing" the volumes searched around the initial and goal states by gradually increasing a cost threshold C_g. We will only consider states around the goal that have a cost less than C_g to get to the goal and states around the initial state that have a cost less than C_g to get from the initial state to that state. These two regions will increase in size as C_g is increased. We stop increasing C_g as soon as the two regions come into contact. The optimal trajectory has to be entirely within the union of these two regions, and has a cost of $2C_g$.

Instead of having the initial conditions of the trajectories laid out on a grid over the whole space, the initial conditions are laid out on a grid over the surface separating the inside and the outside surfaces of the volumes described above. The resolution of this grid is adaptively determined by checking whether the value function of one trajectory correctly predicts the cost of a neighboring trajectory. If it does not, additional grid points are added between the inconsistent trajectories.

During this global optimization we separate the state space into a volume around the goal which has been completely solved and the rest of the state space, in which no exploration or computation has been done. Each iteration of the algorithm enlarges the completely solved volume by performing dynamic programming from a surface of slightly increased cost to the current constant cost surface. When the solved volume includes a known starting point or contacts a similar solved volume with constant cost to get to the boundary from the starting point, a globally optimal trajectory from the start to the goal has been found.

5 DP Based On Approximating Constant Cost Contours

Unfortunately, adaptive grids based on constant cost contours still suffer from the curse of dimensionality, having only reduced the dimensionality of the problem by 1. We are currently exploring methods to approximate constant cost contours. For example, constant cost contours can be approximated by growing "key" trajectories. Trajectories cab be grown along the "bottoms" of the value function "valleys". The location of a constant cost contour can be estimated by using local quadratic models of the value function produced by the process which optimizes the trajectory. These approximate representations do not suffer from the curse of dimensionality. They require on the order of TD^2, where T is the length of time the trajectory requires to get to the goal, and D is the dimensionality of the state space.

6 Summary

Dynamic programming provides a methodology to plan trajectories and design controllers and estimators for nonlinear systems. However, general dynamic programming is computationally intractable. We have developed procedures that allow more complex planning problems to be solved. We have modified the State Increment Dynamic Programming approach of Larson (1968) in several ways:

1. In State Increment DP, a constant action is integrated to form a trajectory segment from the center of a cell to its boundary. We use second order local trajectory optimization (Differential Dynamic Programming) to generate an optimal trajectory and form an optimal policy in a tube surrounding the optimal trajectory within a cell. The trajectory segment and local policy are globally optimal, up to the resolution of the representation of the value function on the boundary of the cell.

2. We use the optimal policy within each cell to guide the local trajectory optimization to form a globally optimal trajectory from the center of each cell all the way to the goal. This helps us avoid the accumulation of interpolation errors as one moves from cell to cell in the state space, and avoid limitations caused by limited resolution of the representation of the value function over the state space.

3. The second order trajectory optimization provides us with estimates of the value function and its first and second spatial derivatives along each trajectory. This provides a natural guide for adaptive grid approaches.

4. During the global optimization we separate the state space into a volume around the goal which has been completely solved and the rest of the state space, in which no exploration or computation has been done. The surface separating these volumes is a surface of constant cost, with respect to achieving the goal.

5. Each iteration of the algorithm enlarges the completely solved volume by performing dynamic programming from a surface of slightly increased cost to the current constant cost surface.

6. When the solved volume includes a known starting point or contacts a similar solved volume with constant cost to get to the boundary from the starting point, a globally optimal trajectory from the start to the goal has been found. No optimal trajectory will ever leave the solved volumes. This would require the trajectory to increase rather than decrease its cost to get to the goal as it progressed.

7. The surfaces of constant cost can be approximated by a representation that avoids the curse of dimensionality.

8. The true test of this approach lies ahead: Can it produce reasonable solutions to complex problems?

References

Bellman, R., (1957) *Dynamic Programming,* Princeton University Press, Princeton, NJ.

Bertsekas, D.P., (1987) *Dynamic Programming: Deterministic and Stochastic Models,* Prentice-Hall, Englewood Cliffs, NJ.

Dyer, P. and S.R. McReynolds, (1970) *The Computation and Theory of Optimal Control,* Academic Press, New York, NY.

Jacobson, D.H. and D.Q. Mayne, (1970) *Differential Dynamic Programming,* Elsevier, New York, NY.

Larson, R.E., (1968) *State Increment Dynamic Programming,* Elsevier, New York, NY.