Model Based Planning

Aaron Lou

June 2018

1 Overview

1.1 Introduction

In the methods covered so far (policy gradient, actor-critic, q-learning), we do not assume any knowledge of the model. Instead, we just use the model to sample rewards.

However, there are methods involving models. Model Based Planning is a subset of methods which assumes knowledge of the dynamics. It finds ways to figure out the policy given the model dynamics.

1.2 Methodologies

The deterministic case (open loop) is looking for the best set of actions given a state. This is given by

$$a_1, \dots, a_T = \arg \max_{a_1, \dots, a_T} \sum_{t=1}^{T} r(s_t, a_t)$$

where $a_{t+1} = f(s_t, a_t)$.

The stochastic case is similar and is given by

$$p_\theta(s_1, \dots, s_T | a_1, \dots, a_T) = p(s_1) \prod_{t=1}^{T} p(s_{t+1} | s_t, a_t)$$

$$a_1, \dots, a_T = \arg \max_{a_1, \dots, a_T} E\left[\sum_t r(s_t, a_t) | a_1, \dots, a_T \right]$$

But these methods are suboptimal as closed-loop (information is constantly sent) intuitively is better. The stochastic closed loop case is given below

$$p(s_1, a_1, \dots, s_T, a_T) = p(s_1) \prod_{t=1}^{T} \pi(a_t | s_t) p(s_{t+1} | s_t, a_t)$$

$$\pi = \arg\max_{\pi} E_{\tau \sim p(\tau)} \left[\sum_t r(s_t, a_t) \right]$$

and we will set π to be a neural net for globcal functions, and a linear function of the form $K_t s_t + k_t$ for local functions.

2 Optimization methods

2.1 Stochastic Case

Obviously, we have the simple method of guess and check. We say $A = a_1, \ldots, a_T$. If we pick uniform $A_1, \ldots A_N$ from some distribution, we pick

$$A_i \text{ based on } \arg\max_i J(A_i)$$

where

$$a_1, \ldots, a_T = \arg\max_{a_1, \ldots, a_T} J(a_1, \ldots, a_T)$$

The Cross-Entropy method is a better way of this case. It is given as follows

Result: The best value A for our
while *our number of loops* **do**
1. Sample $A_1, \ldots, A_N$ from $p(A)$ for our distribution p;
2. evaluate $J(A_1), \ldots, J(A_N)$;
3. pick the *elites* $A_{i_1}, \ldots, A_{i_M}$ with the highest value $M < N$;
4. refit $p(A)$ to the elites;

end

Algorithm 1: The Cross Entropy Method (CEM)

Another This is very simple and fast (if parallelized). However it only works for small dimensions and open-loop planning.

2.2 Monte Carlo tree search

Monte Carlo tree search is method of optimization for discrete dynamics. We can represent our states as a tree. The intuition is to have a tree policy to find the values for leaf nodes and choose the best leaf node. This is because if we're in a good state, we can just use a random policy.

Result: The best action s_1
while *our loop number* **do**
1. Find a leaf s_l using TreePolicy(s_1);
2. evaluate the leaf using DefaultPolicy(s_1);
3. update all values in tree between s_1 and s_l;

end
take the best action s_1;

Algorithm 2: Monte Carlo tree search

The DefaultPolicy is the random policy. UCT TreePolicy(s_t) is as follows.

If s_t not fully expanded, choose new a_t else choose the best child based on the following score function

$$\text{Score}(s_t) = \frac{Q(s_t)}{N(s_t)} + 2C\sqrt{\frac{2\ln N(s_{t-1})}{N(s_t)}}$$

3 Trajectory Optimization

We are using x_t for state, u_t for action, f for transition dynamics, and c for cost function because Russian people. We also use min for our cost. Our new formulas are

$$\min_{u_1,\dots,u_T} \sum_{t=1}^{T} c(x_t, u_t) \text{ s. t. } x_t = f(x_{t-1}, u_{t-1})$$

$$\min_{u_1,\dots,u_T} c(x_1, u_1) + c(f(x_1, u_1), u_2) + \cdots + c(f(f(\dots)\dots), u_T)$$

We can differentiate using

$$\frac{df}{dx_t}, \frac{df}{du_t}, \frac{dc}{dx_t}, \frac{dc}{du_t}$$

but we notice that this isn't actually good because early actions will have a ridiculously large gradient. Instead, we use second order methods.

3.1 Shooting methods vs collocation

Shooting methods optimize over actions only

$$\min_{u_1,\dots,u_T} c(x_1, u_1) + c(f(x_1, u_1), u_2) + \cdots + c(f(f(\dots)\dots), u_T)$$

while collocation methods optimizes over both states and actions.

$$\min_{u_1,\dots,u_T,x_1,\dots,x_T} \sum_{t=1}^{T} c(x_t, u_t) \text{ s.t. } x_t = f(x_{t-1}, u_{t-1})$$

3.2 Linear Case: LQR

If we assume that our transition dynamic f is linear and our cost function c is quadratic we notice that our function

$$\min_{u_1,\dots,u_T} c(x_1, u_1) + c(f(x_1, u_1), u_2) + \cdots + c(f(f(\dots)\dots), u_T)$$

can be simplified using

$$f(x_t, u_t) = F_t \begin{bmatrix} x_t \\ u_t \end{bmatrix} + f_t \quad c(x_t, u_t) = \frac{1}{2} \begin{bmatrix} x_t \\ u_t \end{bmatrix}^T C_t \begin{bmatrix} x_t \\ u_t \end{bmatrix} + \begin{bmatrix} x_t \\ u_t \end{bmatrix}^T c_t$$

If we take

$$C_T = \begin{bmatrix} C_{x_T,x_T} & C_{x_T,u_T} \\ C_{u_T,x_T} & C_{u_T,u_T} \end{bmatrix} \quad c_T = \begin{bmatrix} c_{x_T} \\ c_{u_T} \end{bmatrix}$$

and we maximize $Q(x_t, u_t) = \text{const} + \frac{1}{2} \begin{bmatrix} x_T \\ u_T \end{bmatrix}^T C_T \begin{bmatrix} x_T \\ u_T \end{bmatrix} + \begin{bmatrix} x_T \\ u_T \end{bmatrix}^T c_T$.

$$u_T = -C_{u_T,u_T}^{-1}(C_{u_T,x_T} x_T + c_{u_T})$$

$$u_T = K_T x_T + k_T \quad K_T = -C_{u_T,u_T}^{-1} \quad k_T = -C_{u_T,u_T}^{-1} c_{u_T}$$

and can substitute this in to get that

$$V(x_t) = Q(x_t, u_t) = \text{const} + \frac{1}{2} x_T^T V_T x_T + x_T^T v_T$$

$$V_T = C_{x_T,x_T} + C_{x_T,u_T} K_T + K_T^T C_{u_T,x_T} + K_T^T C_{u_T,u_T} K_T$$

$$v_T = c_{x_T} + C_{x_T,u_T} k_T + K_T^T C_{u_T} + K_T^T C_{u_T,u_T} k_T$$

Note that for the previous time step, we have the recurrence relation

$$f(x_{T-1}, u_{T-1}) = x_T = F_{T-1} \begin{bmatrix} x_{T-1} \\ u_{T-1} \end{bmatrix} + f_{T-1}$$

$$Q(x_{T-1}, u_{T-1}) = \text{const} + \frac{1}{2} \begin{bmatrix} x_{T-1} \\ u_{T-1} \end{bmatrix}^T C_{T-1} \begin{bmatrix} x_{T-1} \\ u_{T-1} \end{bmatrix} + \begin{bmatrix} x_{T-1} \\ u_{T-1} \end{bmatrix}^T c_{T-1} + V(x_T)$$

$$V(x_T) = \text{const} + \frac{1}{2} \begin{bmatrix} x_{T-1} \\ u_{T-1} \end{bmatrix}^T F_{T-1}^T V_T F_{T-1} \begin{bmatrix} x_{T-1} \\ u_{T-1} \end{bmatrix} + \begin{bmatrix} x_{T-1} \\ u_{T-1} \end{bmatrix}^T F_{T_1}^T V_T f_{T-1} + \begin{bmatrix} x_{T-1} \\ u_{T-1} \end{bmatrix}^T F_{T-1}^T v_T$$

and can solve this to get

$$u_{T-1} = K_{T-1} x_{T-1} + k_{T-1} \quad K_{T-1} = -Q_{u_{T-1},u_{T-1}}^{-1} Q_{u_{T-1},x_{T-1}} \quad k_{T-1} = -Q_{u_{T-1},u_{T-1}}^{-1} q_{u_{T-1}}$$

The full algorithm can be enumerated as follows

Result: We enumerate the recurrent relations for u_t
for *$t = T$ to 1* **do**
$Q_t = C_t + F_t^T V_{t+1} F_t$;
$q_t = c_t + F_t^T V_{t+1} f_t + F_t^T v_{t+1}$;
$Q(x_t, u_t) = \text{const} + \frac{1}{2} \begin{bmatrix} x_t \\ u_t \end{bmatrix}^T Q_t \begin{bmatrix} x_t \\ u_t \end{bmatrix} + \begin{bmatrix} x_t \\ u_t \end{bmatrix}^T q_t$;
$u_t \leftarrow \arg\min_{u_t} Q(x_t, u_t) = K_t x_t + k_t$;
$K_t = -Q_{u_t,u_t}^{-1} Q_{u_t,x_t}$;
$k_t = -Q_{u_t,u_t}^{-1} q_{u_t}$;
$V_t = Q_{x_t,x_t} + Q_{x_t,u_t} K_t + K_t^T Q_{u_t,x_t} + K_t^T Q_{u_t,u_t} K_t$;
$v_t = q_{x_t} + Q_{x_t,u_t} k_t + K_t^T Q_{u_t} + K_t^T Q_{u_t,u_t} k_t$;
$V(x_t) = \text{const} + \frac{1}{2} x_t^T V_t x_t + x_t^T v_t$;
end

Algorithm 3: LQR algorithm - backward recursion

and the forward pass is defined by

Result: We calculate the value of u_t
for *$t = 1$ to T* **do**
$u_t = K_t x_t + k_t$;
$x_{t+1} = f(x_t, u_t)$;
end

Algorithm 4: LQR algorithm - forward pass

and we find that $Q(x_t, u_t)$ is the total cost if we take u_t from x_t. $V(x_t)$ is the total cost from x_t to the end.

3.3 Stochastic Dynamics

If we have stochastic dynamics (ie not deterministic), we have

$$p(x_{t+1}|x_t, u_t) = \mathcal{N}\left(F_t \begin{bmatrix} x_t \\ u_t \end{bmatrix} + f_t \Sigma_t\right)$$

We can derive that we just use the same algorithm and pick $u_t = K_t x_t + k_t$ since the noise is symmetric.

3.4 Non-linear dynamics: DDP/Iterative LQR

Since the linear model is a relatively rare circumstance, we can't really use LQR that often. However often, we can approximate using taylor series and use LQR as follows

$$f(x_t, u_t) \approx f(\hat{x_t}, \hat{u_t}) + \nabla_{x_t,u_t} f(\hat{x_t}, \hat{u_t}) \begin{bmatrix} x_t - \hat{x_t} \\ u_t - \hat{u_t} \end{bmatrix}$$

$$c(x_t, u_t) \approx c(\hat{x_t}, \hat{u_t}) + \nabla_{x_t, u_t} c(\hat{x_t}, \hat{u_t}) \begin{bmatrix} x_t - \hat{x_t} \\ u_t - \hat{u_t} \end{bmatrix} + \frac{1}{2} \begin{bmatrix} x_t - \hat{x_t} \\ u_t - \hat{u_t} \end{bmatrix}^T \nabla^2_{x_t, u_t} c(\hat{x_t}, \hat{u_t}) \begin{bmatrix} x_t - \hat{x_t} \\ u_t - \hat{u_t} \end{bmatrix}$$

and our specific functions are

$$\overline{f}(\delta x_t, \delta u_t) = F_t \begin{bmatrix} \delta x_t \\ \delta u_t \end{bmatrix} \quad F_t = \nabla_{x_t, u_t} f(\hat{x_t}, \hat{u_t})$$

$$\overline{c}(\delta x_t, \delta u_t) = \frac{1}{2} \begin{bmatrix} \delta x_t \\ \delta u_t \end{bmatrix}^T C_t \begin{bmatrix} \delta x_t \\ \delta u_t \end{bmatrix} + \begin{bmatrix} \delta x_t \\ \delta u_t \end{bmatrix}^T c_t$$

$$C_t = \nabla^2_{x_t, u_t} c(\hat{x_t}, \hat{u_t}) \quad c_t = \nabla_{x_t, u_t} c(\hat{x_t}, \hat{u_t})$$

$$\delta x_t = x_t - \hat{x_t} \quad \delta u_t = u_t - \hat{u_t}$$

We can use LQR on dynamics $\overline{f}$, cost $\overline{c}$, state δx_t and action δu_t. The simplified iterative LQR algorithm is

Result: Updated x_t and u_t
while *until convergence* **do**
  $F_t = \nabla_{x_t, u_t} f(\hat{x_t}, \hat{u_t})$;
  $c_t = \nabla_{x_t, u_t} c(\hat{x_t}, \hat{u_t})$;
  $C_t = \nabla^2_{x_t, u_t} c(\hat{x_t}, \hat{u_t})$;
  Run LQR backwards pass on state $\delta x_t = x_t \hat{x_t}$ and action $\delta u_t = u_t - \hat{u_t}$;
  Run forward pass with real nonlinear dynamics and $u_t = K_t(x_t - \hat{x_t}) + k_t + \hat{u_t}$;
  Update $\hat{x_t}$ and $\hat{u_t}$ based on states and actions in forward pass;
end

Algorithm 5: iLQR algorithm

This works because it is an approximation to Newton's method for computing $\min_x g(x)$.

Result: Our local approximation of the function
while *until convergence* **do**
  $g = \nabla_x g(\hat{x})$;
  $H = \nabla^2_x g(\hat{x})$;
  $\hat{x} \leftarrow \arg\min_x \frac{1}{2}(x - \hat{x})^T H (x - \hat{x}) + g^T (x - \hat{x})$;
end

Algorithm 6: Newton's Algorithm

Newton's method is based on second order dynamics approximation (called DDP differential dynamic programming):

$$f(x_t, u_t) \approx f(\hat{x_t}, \hat{u_t}) + \nabla_{x_t, u_t} f(\hat{x_t}, \hat{u_t}) \begin{bmatrix} \delta x_t \\ \delta u_t \end{bmatrix} + \frac{1}{2} \left(\nabla^2_{x_t, u_t} f(\hat{x_t}, \hat{u_t}) \cdot \begin{bmatrix} \delta x_t \\ \delta u_t \end{bmatrix} \right) \begin{bmatrix} \delta x_t \\ \delta u_t \end{bmatrix}$$

Newton's method isn't that great because we often overshoot since we're approximating. We do a line search by introducing a term α, and similarly, we have an updated iLQR algorithm.

Result: Updated x_t and u_t
while *until convergence* **do**
  $F_t = \nabla_{x_t,u_t} f(\hat{x}_t, \hat{u}_t)$;
  $c_t = \nabla_{x_t,u_t} c(\hat{x}_t, \hat{u}_t)$;
  $C_t = \nabla^2_{x_t,u_t} c(\hat{x}_t, \hat{u}_t)$;
  Run LQR backwards pass on state $\delta x_t = x_t \hat{x}_t$ and action
   $\delta u_t = u_t - \hat{u_t}$;
  Run forward pass with real nonlinear dynamics and
   $u_t = K_t(x_t - \hat{x_t}) + \alpha k_t + \hat{u_t}$;
  Update $\hat{x_t}$ and $\hat{u_t}$ based on states and actions in forward pass;
end

Algorithm 7: updated line search iLQR algorithm