17-740 Spring 2025

Algorithmic Foundations of Interactive Learning

Lecture 17: Model Predictive Control & Test-Time Scaling

Lecturer : Gokul Swamy Scribe: Michael, Megan Li, Yuemin Mao, Jehan Yang

1.1 What is Model Predictive Control (MPC)?

Model Predictive Control (MPC) is a fundamental building block of most modern robotics systems. Intuitively, rather than solving a planning problem at all states one could see in an *offline* fashion, we simply solve a truncated horizon planning problem at the states we do see during *online* rollouts in MPC. In this sense, MPC is a *lazy* algorithm, in the sense it only does computation at the last moment before it is required.

A bit more formally, for each $h \in [H]$:

1. Plan a sequence of k actions starting from s_h in $T^\star$ (perfect model) or $\hat{T}$ (imperfect model): $\tilde{a}_{h+h+k}$.
2. Execute the first of these actions $a_h = \tilde{a_h}$ in $T^\star$.

In other words, we solve

$$\tilde{a}_{h:h+k} = \arg\max_{a_{h:h+k}} \mathbb{E}_T \left[\sum_{\tau=h}^{h+k} r(s_\tau) + V(s_{h+k}) | a_{h:h+k} \right], \tag{1.1}$$

before executing the first of these actions in the real world and then re-planning. Re-planning is only valuable when our estimated s'_{h+1} (what we expect would happen) differs from the true s_{h+1}, i.e. when the dynamics are stochastic. This is true for many robotics problems but not for auto-regressive language generation. Thus, the variants of MPC that are used for language models (e.g. best-of-N) usually don't involve a re-planning procedure.

1.2 How much does lookahead improve performance?

A natural question at this point might be what the value of MPC / lookahead search is for performance. As we will discuss in greater detail below, a k-step lookahead search can be considered performing k iterations of the policy improvement procedure.

Consider some policy π with value function V^π. Single-step policy improvement looks like

$$\pi_1^+(s) = \arg\max_{a \in \mathcal{A}} r(s) + \mathbb{E}_{s' \sim T(s,a)}[V^\pi(s')]]. \tag{1.2}$$

Observe that this is just MPC with $k = 1$. Now, using the Performance Difference Lemma, we can prove that this *local* improvement guarantees a better policy, *globally* speaking:

$$\begin{aligned} J(\pi_1^+) - J(\pi) &= \mathbb{E}_{\zeta \sim \pi_1^+}\left[\sum_h^H Q^\pi(s_h, a_h) - \mathbb{E}_{a' \sim \pi(s_h)}\left[Q^\pi(s_h, a')\right]\right] && (1.3)\\ &= \mathbb{E}_{\zeta \sim \pi_1^+}\left[\sum_h^H \max_a Q^\pi(s_h, a) - \mathbb{E}_{a' \sim \pi(s_h)}\left[Q^\pi(s_h, a')\right]\right] && (1.4)\\ &\geq 0, && (1.5) \end{aligned}$$

where the last follows from the fact that each term inside the expectation is non-negative.

Let us now consider MPC with k-step lookahead. For simplicity, we will assume access to a perfect model but a more general version of the following results can be proved via an appeal to the simulation lemma we discussed previously. Define the k-step lookahead policy as

$$\pi_k^+(s_h) = \arg\max_{a_{h:h+k}} \mathbb{E}_{T^\star}\left[\sum_{\tau=h}^{h+k} \gamma^{\tau-h} r(s_\tau) + \gamma^k V^\pi(s_{h+k}) \mid s_h, a_{h:h+k}\right]. \tag{1.6}$$

Define ϵ as the largest value difference between our current policy and the optimal policy over the state space: $\epsilon = \max_{s \in S} \frac{V^*(s) - V^\pi(s)}{H}$. We define ϵ this way to make things horizon-independent. We will now show that

$$J(\pi^*) - J(\pi_k^+) \leq O\left(\frac{\epsilon \gamma^k H(1-\gamma^H)}{1-\gamma^k}\right), \tag{1.7}$$

i.e. that lookahead search allows us to decrease the performance difference as we scale up k.

Proof: Similar to the simulation lemma, the proof proceeds via adding / subtracting a "cross-term" and then canceling terms to end up with time-shifted terms. For notational convenience, we will drop the $s_1 = s$ conditioning from all of the preceding equations.

$$\begin{aligned} V^\star(s) - V_+^k(s) &= \underbrace{\mathbb{E}\left[\sum_h^k \gamma^h r(s_h) + \gamma^k V^\star(s_k) \mid \pi^\star\right]}_{a} - \underbrace{\mathbb{E}\left[\sum_h^k \gamma^h r(s_h) + \gamma^k V_+^k(s_k) \mid \pi_k^+\right]}_{d} \\ &= a + \underbrace{\mathbb{E}\left[\sum_h^k \gamma^h r(s_h) + \gamma^k V^\star(s_k) \mid \pi_k^+\right]}_{b} - \underbrace{\mathbb{E}\left[\sum_h^k \gamma^h r(s_h) + \gamma^k V^\star(s_k) \mid \pi_k^+\right]}_{c} + d. \end{aligned}$$

Observe that the the discounted sum of rewards in b and d cancel out as the expectations are taken over actions chosen by the same policy. After canceling and collecting like terms, we are left with

$$= a - \mathbb{E}\left[\sum_{h}^{k} \gamma^h r(s_h) + \gamma^k V^\star(s_k) \mid \pi_k^+\right] + \gamma^k \mathbb{E}\left[V^*(s_h) - V_k^+(s_h) \mid \pi_k^+\right]. \tag{1.8}$$

Now, given we don't a-priori know that $V^\star$ is, we're going to replace it with the value function of the policy V^π, which by assumption differs by at most ϵH at any state. We'll do this in both terms a and b, which gives us

$$\begin{aligned}
&\leq \mathbb{E}\left[\sum_{h}^{k} \gamma^h r(s_h) + \gamma^k V^\pi(s_k) \mid \pi^\star\right] - \mathbb{E}\left[\sum_{h}^{k} \gamma^k r(s_h) + \gamma^k V^\pi(s_h) \mid \pi_k^+\right] \\
&+ \gamma^k \mathbb{E}\left[V^\star(s_k) - V_k^+(s_h) \mid \pi_k^+\right] + 2\epsilon H \gamma^k. \qquad (1.9)
\end{aligned}$$

Next, we note that π_+^k is the policy that by definition maximizes the first term so we can upper bound the expectation under $\pi^\star$ and cancel like terms:

$$\begin{aligned}
&\leq \mathbb{E}\left[\sum_{h}^{k} \gamma^h r(s_h) + \gamma^k V^\pi(s_k) \mid \pi_+^k\right] - \mathbb{E}\left[\sum_{h}^{k} \gamma^h r(s_h) + \gamma^k V^\pi(s_k) \mid \pi_k^+\right] \\
&+ \gamma^k \mathbb{E}\left[V^*(s_k) - V_k^+(s_h) \mid \pi_k^+\right] + 2\epsilon H \gamma^k \qquad (1.10) \\
&= \gamma^k \mathbb{E}\left[V^*(s_k) - V_k^+(s_h) \mid \pi_k^+\right] + 2\epsilon H \gamma^k. \qquad (1.11)
\end{aligned}$$

Observe this is the same term we started off with, just shifted k steps into the future. We can therefore compute the sum of the geometric series analytically to arrive at

$$\leq 2\epsilon H \gamma^k \sum_{i=1}^{H/k} (\gamma^k)^i = \frac{2\epsilon \gamma^k H(1-\gamma^H)}{1-\gamma^k}. \tag{1.12}$$

For the infinite horizon version, one would instead solve a recurrence relation. ■

1.3 What are the different variants of planning?

Given we now understand the performance benefits of MPC, let us consider several standard algorithms which fall under the template we sketched above. Once again, our optimization problem will be to solve

$$\tilde{a}_{h:h+k} = \arg\max_{a_{h:h+k}} \mathbb{E}_T\left[\sum_{\tau=h}^{h+k} r(s_\tau) + V(s_{h+k}) \mid a_{h:h+k}, s_h\right]. \tag{1.13}$$

Let us consider each part of the above in sequence. Consider for a moment a deterministic problem. A *"shooting" method* expands the above objective via recursion:

$$r(s_0) + r(T(s_0, a_0)) + r(T(T(s_0, a_0), a_1) + \dots \tag{1.14}$$

Observe that the repeated applications of T can lead to an ill-conditioned optimization problem, similar to the vanishing / exploding gradient issue when training a recurrent neural network. This can make it hard to use first-order methods (which rely on gradients) unless there's a more nicely behaved $\hat{T}$ that approximates the ground-truth dynamics well (e.g. in iLQR algorithms). Thus, we'll mostly focus on zeroth-order algorithms for this lecture.

Solving the argmax over k actions can often be expensive, so it is common to have some "base policy" one samples m k-step plans from before selecting from this set of m options.

The expectation over the next k steps is often done inside a (learned) world model.

The value function / terminal heuristic V is left underspecified in the above general formulation. Ideally it would be $V^\star$. In practice, we might hope to have access to some expert value function V^E, in which case the above is a lookahead version of DAgger / AggraVaTe.

Let us now discuss four popular algorithms that fit under the above template.

1.3.1 Best-of-N (BoN)

Under the assumption of having tree-structured, deterministic, perfectly known dynamics (i.e. auto-regressive language generation), we can simplify the above formulation to $a_{1:H} = \arg\max_{a^1_{1:H} \dots a^N_{1:H}} r(s_H)$. No re-planning is required because we always append the token we expect to. A terminal cost is sufficient because s_H allows us to uniquely decode all actions / tokens – it is merely their concatenation.

1.3.2 Cross Entropy Method (CEM)

Sample $a^1_{1:k} \dots a^N_{1:k} \sim \pi_t$. Then, pick the top (both in terms of the k rewards and the terminal cost) $\approx 10\%$ of these and compute their mean μ_{t+1}. Finally, set $\pi_{t+1}(s_h) = \mathcal{N}(\mu_{t+1}, \sigma^2)$. Commonly used robotics planning algorithms like Model Predictive Path Integral Control (MPPI) are essentially fancier versions of the above idea.

1.3.3 Sparse Sampling (Kearns, Mansour, Ng)

The algorithm proceeds as follows: start with some state s. Try each action a_i some number (c) of times and track where each action led. Next, from each of these resulting states, repeat the process. Do this up to a maximum depth of H. Critically, we don't do this for all possible next states – only for the next states seen during the sampling process. Given these empirical frequencies, we can just divide the number of times taking action a in state s lead to state s' by c to build an approximate dynamics model / MDP $\hat{M}$. One can then run an arbitrary RL algorithm inside $\hat{M}$ to compute $\hat{\pi}^*$. It is possible to set c and H such that we have strong policy performance guarantees independent of the size of the state space $|\mathcal{S}|$. This contrasts with the standard policy / value iteration algorithms, where the amount of computation you have to do scales linearly with the size of the state space. We won't delve into it in this lecture but this is, more formally, what the "laziness" of MPC means.

1.3.4 Monte Carlo Tree Search

Rather than sampling each action c times, we can use a more efficient exploration strategy. Intuitively, if we consistently see that some state/action pair leads to a bad outcome, we stop exploring from it. More formally, to trade off exploration and exploitation, we use

$$\pi_{UCT}(s) = \arg\max_a \hat{Q}(s,a) + c\sqrt{\frac{\ln(n(s))}{n(s,a)}}. \tag{1.15}$$

The second term comes from an upper confidence / Hoeffding bound. This can be seen as the natural sequential generalization of a bandit algorithm.

1.4 How can we iterate this process?

Once we've ran a k-step look-ahead search, it often makes sense to not just throw away that computation and instead use it to update the base policy so that we can have a better starting point for the next round of policy improvement. This is often called *expert / dual policy iteration* and is fundamental to how systems like AlphaGo work. Intuitively, one uses MPC as a *local* expert and updates the policy via DAgger / distillation. A bit more formally:

For each round of the algorithm $t \in [T]$:

1. Do a k-step local search / MPC around π_t to produce an improved η_t.
2. Use DAgger with η_t as the expert to produce $\pi_{t+1} = \arg\max_{\pi \in \Pi} \mathbb{E}_{s_h \sim \pi_t, a_h \sim \eta_t}[\log \pi(a_h|s_h)]$.