optimal control & planning

- how can we choose actions under perfect knowledge of system dynamics? this is called either of these names: optimal control, trajectory optimization or planning.
- in future classes, we will learn about what to do when we don't know these system dynamics → how we can learn it. for this topic, we will not do any deep learning.
- optimal control is really the problem of $min_{u_1...u_T, x_1...x_T} \sum_{t=1}^{T} c(x_t, u_t) s.t.\ x_t = f(x_{t-1}, u_{t-1})$
- shooting methods for optimal control are ones which substitute the constraint above into the cost function and solve it as an unconstrained optimization problem (like LQR and iLQR). collocation methods are one which address it as a constrained optimization problem.
- basic formulation:

Trajectory optimization

$$\min_{\mathbf{u}_1,\dots,\mathbf{u}_T} \sum_{t=1}^{T} c(\mathbf{x}_t, \mathbf{u}_t) \text{ s.t. } \mathbf{x}_t = f(\mathbf{x}_{t-1}, \mathbf{u}_{t-1})$$

$$\min_{\mathbf{u}_1,\dots,\mathbf{u}_T} c(\mathbf{x}_1, \mathbf{u}_1) + c(f(\mathbf{x}_1, \mathbf{u}_1), \mathbf{u}_2) + \cdots + c(f(f(\dots)\dots), \mathbf{u}_T)$$

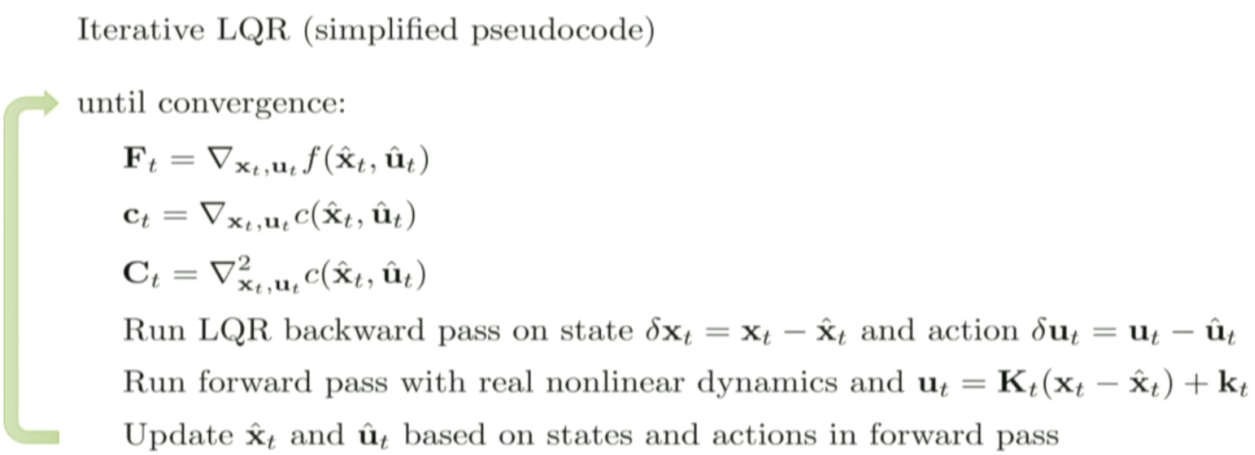
usual story: differentiate via backpropagation and optimize!

need $\frac{df}{d\mathbf{x}_t}, \frac{df}{d\mathbf{u}_t}, \frac{dc}{d\mathbf{x}_t}, \frac{dc}{d\mathbf{u}_t}$

in practice, it really helps to use a 2^{nd} order method!

- if someone refers to a "linearization of system dynamics", what they are talking about is taking these derivatives and doing a SGD.
- in practice, it helps to use a 2nd order method for numerical reasons which levine doesn't go into. one such class of second order techniques is the LQR: linear quadratic regulator.
- first we will look at a simple linear formulation (meaning the system dynamics are linear) LQR algorithm. later we will look at iterative LQR which also works for non-linear system dynamics.
- linear dynamical system means the transition function is some matrix times the concatenated vector of states and actions plus some vector, the matrix and vector might be time dependent, in general.
- we will assume that the cost is some quadratic function of state, action. hence the name LQR: linear dynamics and quadratic cost.
- now, the problem is to find $u_1...u_T$ that minimizes the cost function $c((x_1, u_1)) + c(x_2, u_2)..c(x_T, u_T)$ but replacing with dynamics,$c((x_1, u_1)) + c(f(x_1, u_1), u_2)..c(f(f(..)..), u_T)$.
- Now we differentiate the last term of this cost function with respect to the last action u_T and set it to 0. solving for u_T, we get something in terms of x_T, the last state. Now substituting that back in the cost term for the last state, we get a last cost term purely in terms of x_T.

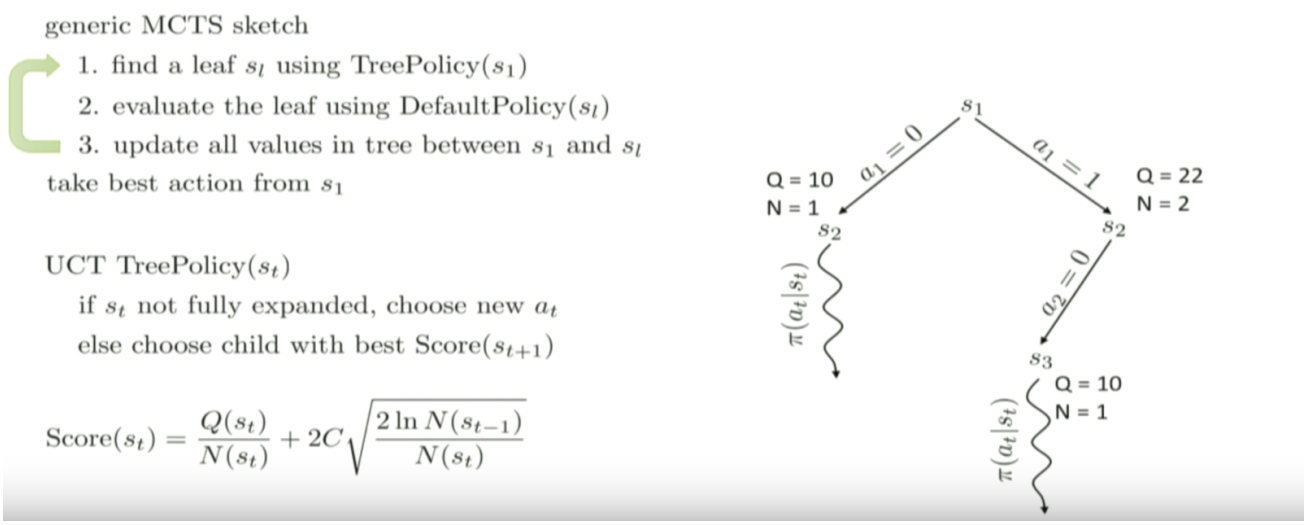
- we now have the cost in terms of x_T. Now we'd like to continue on to find u_{T-1}. Now we write the last two terms of the cost function purely as a function of u_{T-1} and x_{T-1}. Again, we differentiate with respect to u_{T-1}, set it to 0 and obtain u_{T-1} in terms of x_{T-1}.
- Continuing this pattern, we see a backward recursion. We start at time step T and iterate backward. Eventually, as we compute each action as a function of previous state, we will get to a stage where we compute the first action as a function of first step, which is known!
- Now you can see how this optimal control problem is an instance of the general reinforcement learning problem we saw in cs 229, which was to minimize total cost over time (we have assumed deterministic dynamics here, the same works for certain dynamics which have symmetric distributions around the mean → levine doesn't prove this)
- if we have non-linear dynamics, differential dynamic programming (also called iterative LQR is used). here, the basic problem is to linearize (approximate) the non-linear dynamics around some states/actions as a linear quadratic system using taylor expansions.
- we will do something inspired by newton's method. (newton's method is one where we approximate a function using second derivative in taylor expansion and choose a $\vec{x}$ that minimizes it based on this taylor expansion). we will do an approximation of newton's method, using only first derivative term, since second derivative gets messy numerically and first derivative seems to work.
- often, to make newton's method actually work, we will need to use something like backtracking line search because of the overshooting problem because our taylor expansion is only accurate locally. a **backtracking line search**, is a line search method to determine the maximum amount to move along a given search direction. It involves starting with a relatively large estimate of the step size for movement along the search direction, and iteratively shrinking the step size (i.e., "backtracking") until a decrease of the objective function is observed that adequately corresponds to the decrease that is expected, based on the local gradient of the objective function.
- the algorithm, which is a dynamic programming algorithm (because it runs LQR in the inner loop) that works iteratively.



- Note that this algorithm basically runs regular LQR in perturbation space with locally linear cost and dynamics functions.
- since we use DP to solve the system (just like we did in the ml class - cs 229), but we also use the differentiability property to approximate (linearize) it, we call it differential dynamic programming.
- note that a LQR loop gives us a solution in one step (we just have to find u_T from $t = T$ to $t = 1$. because our linearized model is only locally approximate, we HAVE to do a backtracking line search to

update our actions. we can't just take the LQR output as our output. We have to move from where we are in the direction of the LQR output, taking only a small step such that our cost is reduced by an adequate amount (as predicted by the local gradients).

- model predictive control (MPC) is a technique where once we solve the LQR, we take the first action (or a first k actions for small k) and then we re-plan again by running LQR. this way if the state we observe is not what we expected (because our dynamics model is off), we can correct for that and do our best now onwards.
- now, everything we have looked at so far assumes continuity and differentiability of action space and state space. these assumptions of course don't apply in many many situations, like atari games for example. in these cases we can't just use LQR so we need to do something else → introducing monte carlo tree search (MCTS)
- Monte Carlo definition: An analytical technique in which a large number of simulations are run using random quantities for uncertain variables and looking at the distribution of results to infer which values are most likely. The name comes from the city of Monte Carlo, which is known for its casinos.
- lets start with the idea of a exhaustive tree search like in intro AI game theory. of course, this takes exponential time, the states explode quickly and this is no good.
- the main idea behind monte carlo is that we will start with a random policy and just explore around and do some sampling to see which states are good to get into and what their sampled values are. its very possible that these samples are quite a bit off the true values because maybe there is a way to get out of a situation that looks bad but we will for now just sample around and estimate. we will carry out a random policy to the end and determine what happens (final score/reward). and then come back and take branches that maybe exploit paths that seemed good.
- while deciding what to explore after the first few samplings all the way to the end, we will generally choose nodes with the best rewards but we will also prefer rarely visited nodes, just to get better sampling and see if we missed something good.
- a generic version of MCTS:



- the score captures the notion of both average score based on past explorations but also frequency of visits → if a parent has been visited very often, but a child has been visited rarely, then it gives that child a higher score so we bias towards exploring it.
- "the real reason MCTS is used is because it works well empirically → there is a bit of theory but mainly, it tends to work well empirically for a lot of discrete tasks like atari"

- deepmind's alphago used monte carlo tree search as a component, using a smarter policy (which was the learned policy) rather than a random policy → little bit of planning + little bit of learning.
- Levine shows a cool example that combines Dagger (imitation learning) trained with MCTS as the expert, instead of a human labeler.
- now in this topic, for both LQR and MCTS we have assumed that we have a reasonably good model of the dynamics. this might be true in many cases (self driving), but this is not true in other tasks like a house robot assistant doing laundry folding. in these cases, as we will see in the next topics, we will learn a dynamics model and then use that in optimal control/planning. we will also see how to learn the right dynamics model for a given control/planning approach.
- note that monte carlo tree search runs live while playing. its not always a pre-computed policy mapping states to actions. this makes it infeasible to use sometimes.