

# learning dynamical models

- the question is: what if the dynamics of a system aren't known? eg. robotic manipulation in a cluttered environment, a website that shows a recommendation feed to a human user etc.
- we will learn about global dynamics models and local dynamics models (when we can't learn the global model directly)
- we will also see approaches to learning dynamics + policy vs just learning dynamics and using something else for the control part of it.
- a basic version that learns dynamics from data often works pretty well:

model-based reinforcement learning version 0.5:

1. run base policy  $\pi_0(\mathbf{u}_t|\mathbf{x}_t)$  (e.g., random policy) to collect  $\mathcal{D} = \{(\mathbf{x}, \mathbf{u}, \mathbf{x}')_i\}$
2. learn dynamics model  $f(\mathbf{x}, \mathbf{u})$  to minimize  $\sum_i \|f(\mathbf{x}_i, \mathbf{u}_i) - \mathbf{x}'_i\|^2$
3. backpropagate through  $f(\mathbf{x}, \mathbf{u})$  to choose actions (e.g. using iLQR)


- the above is what is meant when people use the term "system identification" in classical robotics.
- the above tends to work well when the distribution it is trained on is representative of the distribution it will see when the policy is run. but just like dagger, we will have a distribution mismatch problem sometimes. this problem becomes especially exacerbated with more expressive model classes like neural nets. but if we have a physics model with just a few parameters to fit and you know that with the right parameters that is a good model, then this actually works pretty well.
- to get around the above problem we will do a thing similar to what we did with dagger which is to collect data by running the current policy iteratively so that there is a better match between distributions of the input and output.

model-based reinforcement learning version 1.0:

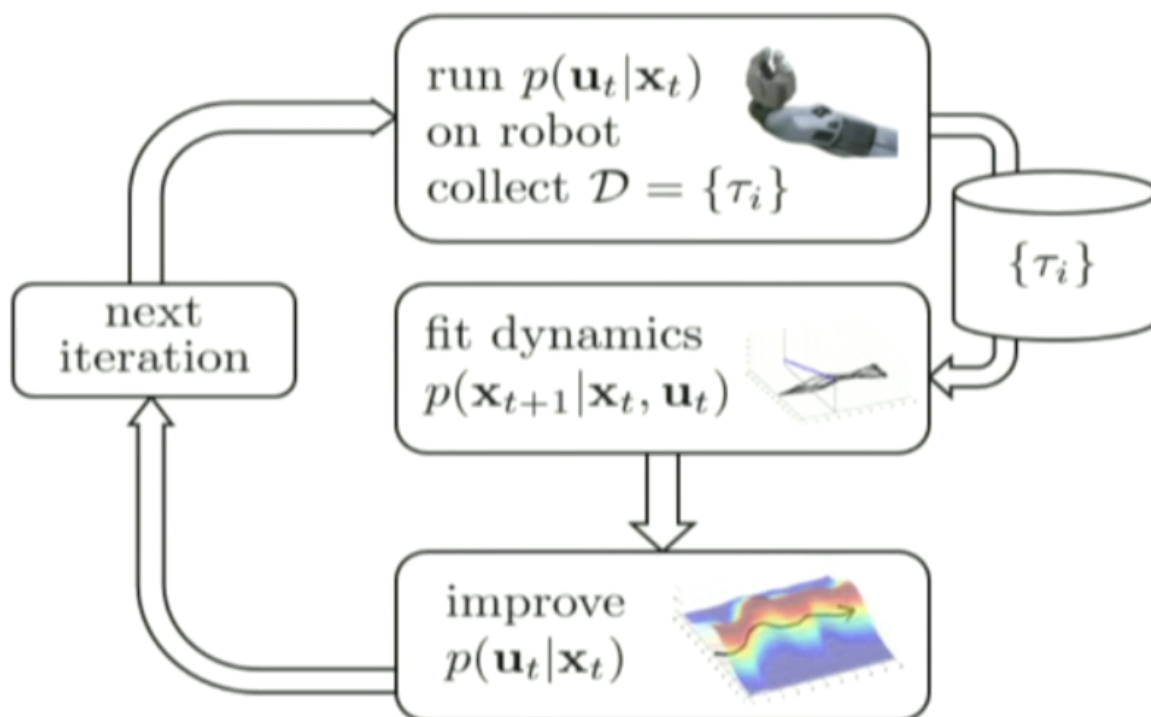
1. run base policy  $\pi_0(\mathbf{u}_t|\mathbf{x}_t)$  (e.g., random policy) to collect  $\mathcal{D} = \{(\mathbf{x}, \mathbf{u}, \mathbf{x}')_i\}$
2. learn dynamics model  $f(\mathbf{x}, \mathbf{u})$  to minimize  $\sum_i \|f(\mathbf{x}_i, \mathbf{u}_i) - \mathbf{x}'_i\|^2$
3. backpropagate through  $f(\mathbf{x}, \mathbf{u})$  to choose actions (e.g. using iLQR)
4. execute those actions and add the resulting data  $\{(\mathbf{x}, \mathbf{u}, \mathbf{x}')_j\}$  to  $\mathcal{D}$

- if our model is slightly off and this is going to cause us to accumulate errors over time and do very bad things, we can do model predictive control here, just like we did with lqr and ilqr → this is model based RL version 1.5
- we can also do model based policy search, not just model based control. its similar, but we just learn a policy:

## model-based reinforcement learning version 2.0:

- 
1. run base policy  $\pi_0(\mathbf{u}_t|\mathbf{x}_t)$  (e.g., random policy) to collect  $\mathcal{D} = \{(\mathbf{x}, \mathbf{u}, \mathbf{x}')_i\}$
  2. learn dynamics model  $f(\mathbf{x}, \mathbf{u})$  to minimize  $\sum_i \|f(\mathbf{x}_i, \mathbf{u}_i) - \mathbf{x}'_i\|^2$
  3. backpropagate through  $f(\mathbf{x}, \mathbf{u})$  into the policy to optimize  $\pi_\theta(\mathbf{u}_t|\mathbf{x}_t)$
  4. run  $\pi_\theta(\mathbf{u}_t|\mathbf{x}_t)$ , appending the visited tuples  $(\mathbf{x}, \mathbf{u}, \mathbf{x}')$  to  $\mathcal{D}$

- policies can be very cheap to run once computed (vs replanning every step), but hopefully they should also compensate for errors like model predictive control does.
- closed loops systems means we plan at first and get a set of actions, only execute the first one, observe and plan again. open loop means plan at first and just execute, execute, execute.
- in general, the dynamic model is stochastic  $\rightarrow$  it outputs  $p(x_{t+1}|\mathbf{x}_t, \mathbf{u}_t)$ .
- what are the concrete dynamics models? they could be gaussian processes, neural nets (especially LSTMs  $\rightarrow$  sequence modeling!) etc. which one to use depends on how much data you have, how much knowledge you have (a domain specific model like a physics model with just a few parameters can be fit with much lesser data than a neural net)
- global models may not be right everywhere. this can lead the planner to seek out regions where the model is erroneously optimistic. to get around this problem, we need to find a good model in most of the state space, which can be hard in many tasks.
- local models try to find good dynamics in the region we are currently located instead of finding a good model everywhere.
- The trick: if we want to backpropagate thru the cost and dynamics, we need the derivatives  $\frac{df}{dx_t}, \frac{df}{du_t}, \frac{dc}{dx_t}, \frac{dc}{du_t}$ . The derivatives of the dynamics are the ones we don't know. instead of computing a big global model, we will just fit the derivatives around the current trajectory/policy! we will use those derivatives to improve our policy/trajectory and then repeat:



- note the difference from before: we throw out the dynamics model every iteration and fit a new one. here too, we will use model predictive control here too so as to correct for drift.
- one approach to fitting the derivative is to execute a few actions with current trajectory/policy and get some data about  $x_t$ ,  $u_t$  and  $x_{t+1}$ . we can use this to fit a linear model of the dynamics using derivatives from our linear model. the problem with a non-stochastic policy/trajectory is that we will end up getting the same set of  $x_t$ ,  $u_t$  and  $x_{t+1}$  over many iterations and we won't be able to fit a linear model/get derivatives.
- So instead, we will use a slightly stochastic controller

$$p(\mathbf{u}_t|\mathbf{x}_t) = \mathcal{N}(\mathbf{K}_t(\mathbf{x}_t - \hat{\mathbf{x}}_t) + \mathbf{k}_t + \hat{\mathbf{u}}_t, \Sigma_t)$$

with the above, we now get some variability in the data samples used to fit the local dynamics model. how do we choose the covariance matrix to use here? once choice is a constant, but a better, but yet simple choice is to use the covariance inversely proportional to how much the cost changes as a result of a small change in action. If it changes a lot, we will use a smaller covariance in that direction to generate points closer together to compute the derivative estimate. and vice versa.

- Now, one last thing to get right: we've used linearization here. so this is only accurate if the new trajectory is close to the old trajectory. to ensure this is true, we will keep limit the distance between the probability distributions of our current and new trajectory, by limiting the KL divergence between the distributions. note that a stochastic trajectory/control policy induces a distribution over trajectories. The KL divergence between the induced distributions is what we will limit. This is called the trust region.
- the math:

# KL-divergences between trajectories

$$D_{\text{KL}}(p(\tau) \parallel \bar{p}(\tau)) = E_{p(\tau)} [\log p(\tau) - \log \bar{p}(\tau)]$$

$$\begin{aligned} \log p(\tau) - \log \bar{p}(\tau) = & \cancel{\log p(\mathbf{x}_1)} + \sum_{t=1}^T \log p(\mathbf{u}_t | \mathbf{x}_t) + \cancel{\log p(\mathbf{x}_{t+1} | \mathbf{x}_t, \mathbf{u}_t)} \\ & - \cancel{\log \bar{p}(\mathbf{x}_1)} + \sum_{t=1}^T -\log \bar{p}(\mathbf{u}_t | \mathbf{x}_t) - \cancel{\log \bar{p}(\mathbf{x}_{t+1} | \mathbf{x}_t, \mathbf{u}_t)} \end{aligned}$$

$$D_{\text{KL}}(p(\tau) \parallel \bar{p}(\tau)) = E_{p(\tau)} \left[ \sum_{t=1}^T \log p(\mathbf{u}_t | \mathbf{x}_t) - \log \bar{p}(\mathbf{u}_t | \mathbf{x}_t) \right]$$

$$D_{\text{KL}}(p(\tau) \parallel \bar{p}(\tau)) = \sum_{t=1}^T E_{p(\mathbf{x}_t, \mathbf{u}_t)} [\log p(\mathbf{u}_t | \mathbf{x}_t) - \log \bar{p}(\mathbf{u}_t | \mathbf{x}_t)]$$

Now we can use a constrained optimization method to minimize cost subject to constraint that KL divergence (for which the expression is above) is below a certain value.