Model-Based Policy Learning

today we will learn algos that use model-based RL to learn policies

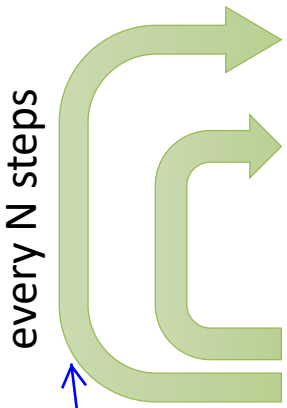
CS 285

Instructor: Sergey Levine
UC Berkeley

Last time: model-based RL with MPC

model-based reinforcement learning version 1.5:

1. run base policy $\pi_0(\mathbf{a}_t|\mathbf{s}_t)$ (e.g., random policy) to collect $\mathcal{D} = \{(\mathbf{s}, \mathbf{a}, \mathbf{s}')_i\}$
2. learn dynamics model $f(\mathbf{s}, \mathbf{a})$ to minimize $\sum_i \|f(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{s}'_i\|^2$
3. plan through $f(\mathbf{s}, \mathbf{a})$ to choose actions
4. execute the first planned action, observe resulting state $\mathbf{s}'$ (MPC)
5. append $(\mathbf{s}, \mathbf{a}, \mathbf{s}')$ to dataset $\mathcal{D}$



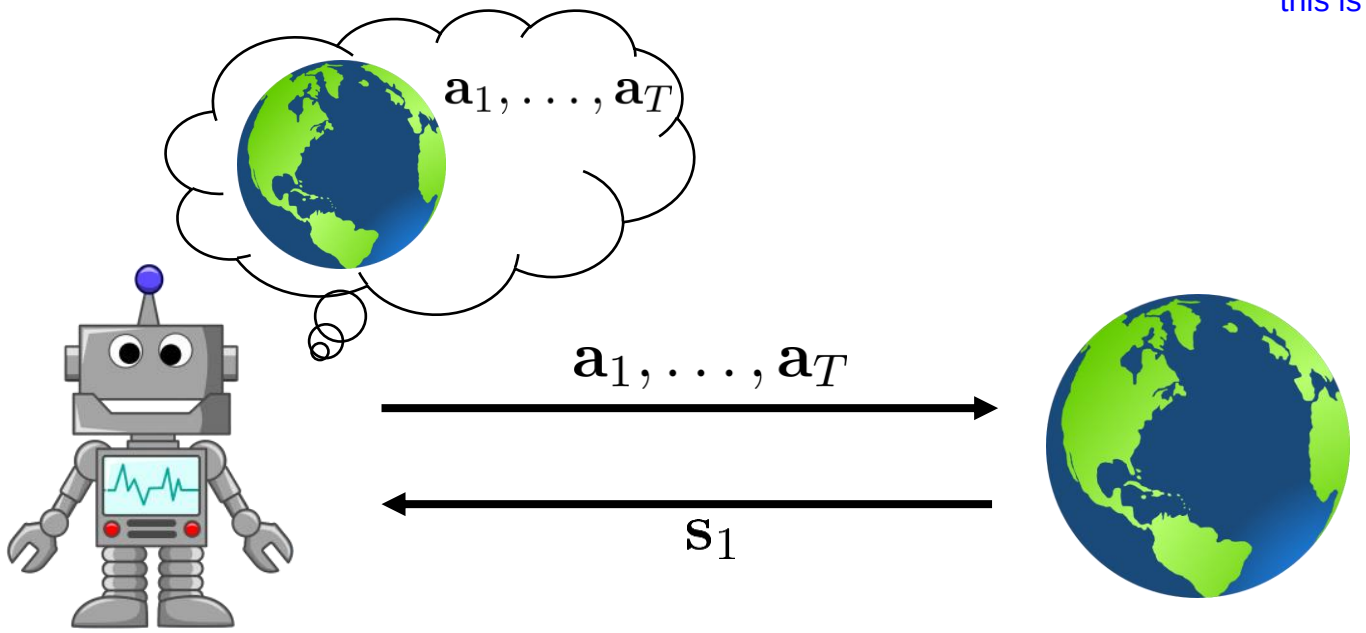
every N many steps or rollouts, go back and retrain the model with now larger dataset

collect data
use that data to train a model
use that model to plan to make decisions every timestep. we replan every timestep. the planning can use any planning algorithm

The stochastic open-loop case

this algorithm is an open loop algo. the facto that your replanning every timestep gives you some closed-loop capability, but the planning you do at each timestep is still open loop.

this is less flexible than policy learning



$$p_\theta(\mathbf{s}_1, \ldots, \mathbf{s}_T | \mathbf{a}_1, \ldots, \mathbf{a}_T) = p(\mathbf{s}_1) \prod_{t=1}^{T} p(\mathbf{s}_{t+1} | \mathbf{s}_t, \mathbf{a}_t)$$

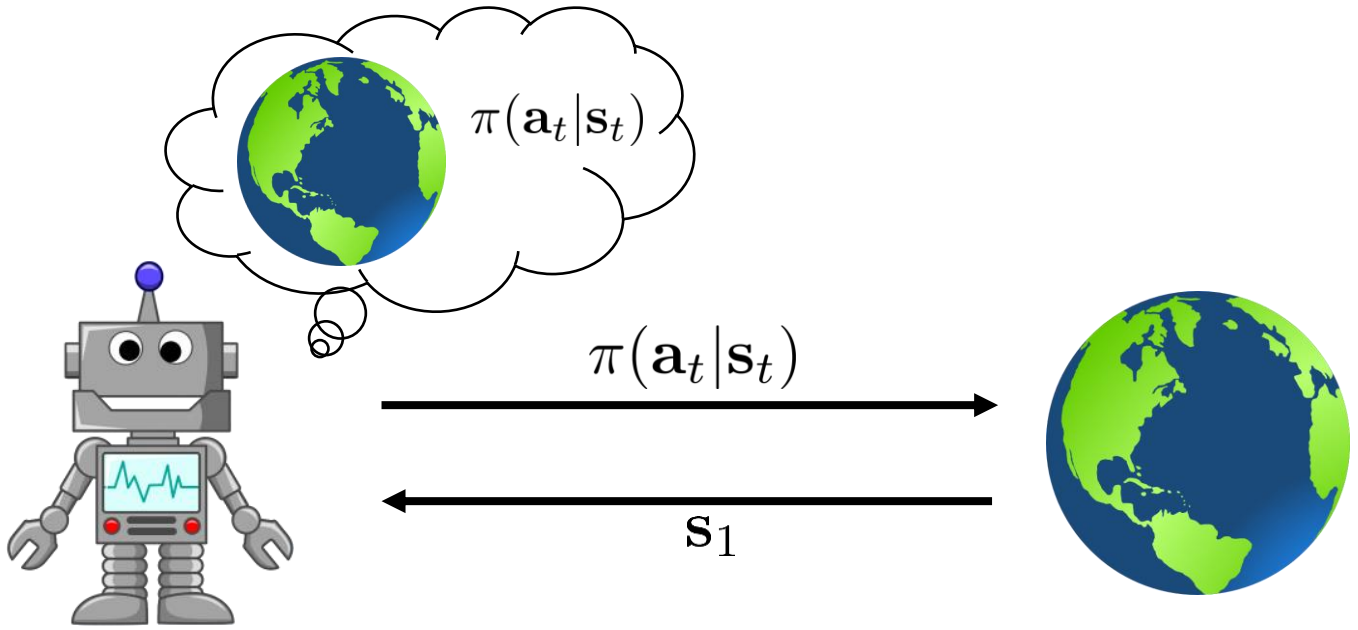
$$\mathbf{a}_1, \ldots, \mathbf{a}_T = \arg \max_{\mathbf{a}_1, \ldots, \mathbf{a}_T} E\left[\sum_t r(\mathbf{s}_t, \mathbf{a}_t) | \mathbf{a}_1, \ldots, \mathbf{a}_T \right]$$

note that this maximization is conditioned on an entire sequence of actions!

why is this suboptimal?

The stochastic **closed**-loop case

instead of the planner planning for multiple timesteps, what we the planner to send back a policy, then you execute the policy



form of π?

neural net global

time-varying linear

$\mathbf{K}_t\mathbf{s}_t + \mathbf{k}_t$ local

from LQR. local so that it only gives good actions around a certain neighborhood

$$p(\mathbf{s}_1, \mathbf{a}_1, \ldots, \mathbf{s}_T, \mathbf{a}_T) = p(\mathbf{s}_1) \prod_{t=1}^{T} \pi(\mathbf{a}_t|\mathbf{s}_t) p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$$

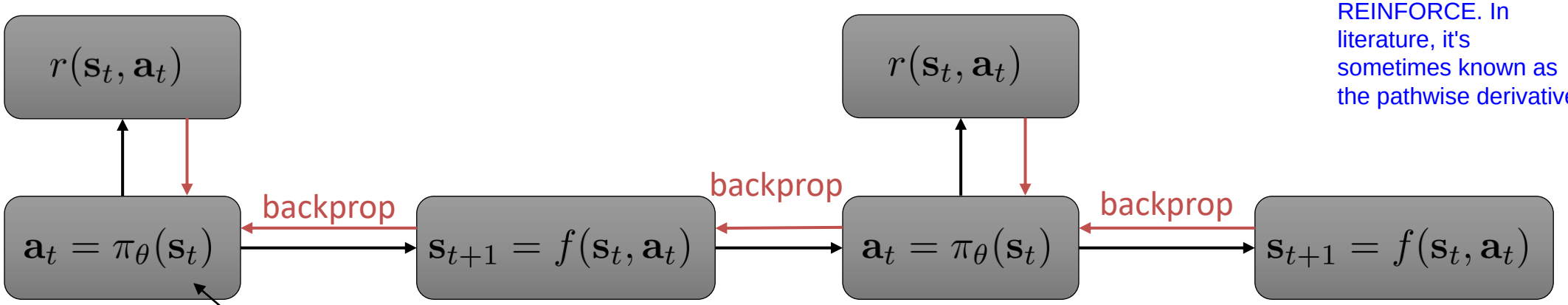
$$\pi = \arg\max_{\pi} E_{\tau \sim p(\tau)} \left[\sum_t r(\mathbf{s}_t, \mathbf{a}_t) \right]$$

only conditioned on the current state s_t, nothing else

Backpropagate directly into the policy?

example: trajectory of length 2

this is not the same as the PG in REINFORCE. In literature, it's sometimes known as the pathwise derivative



easy for deterministic policies, but also possible for stochastic policy

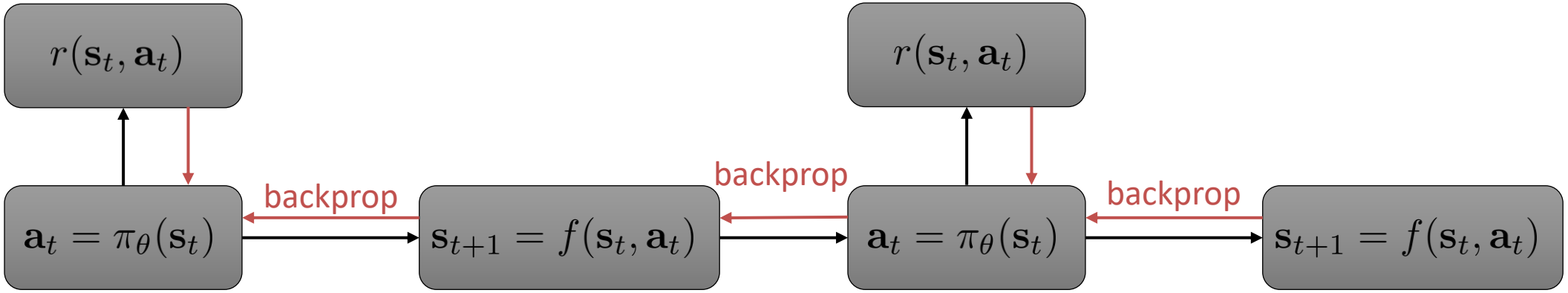
for stochastic policies we want to use the reparameterization trick

model-based reinforcement learning version 2.0:

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

construct this graph and call .gradients()

What's the problem with backprop into policy?



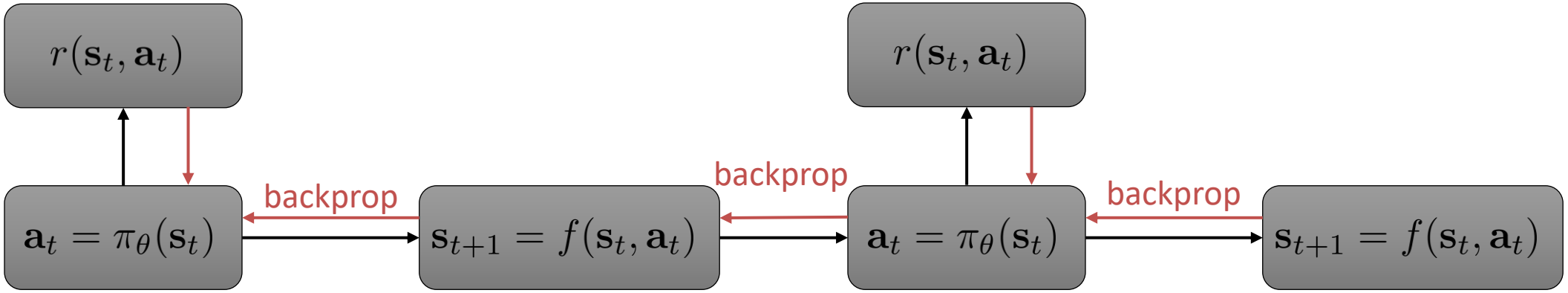
big gradients here

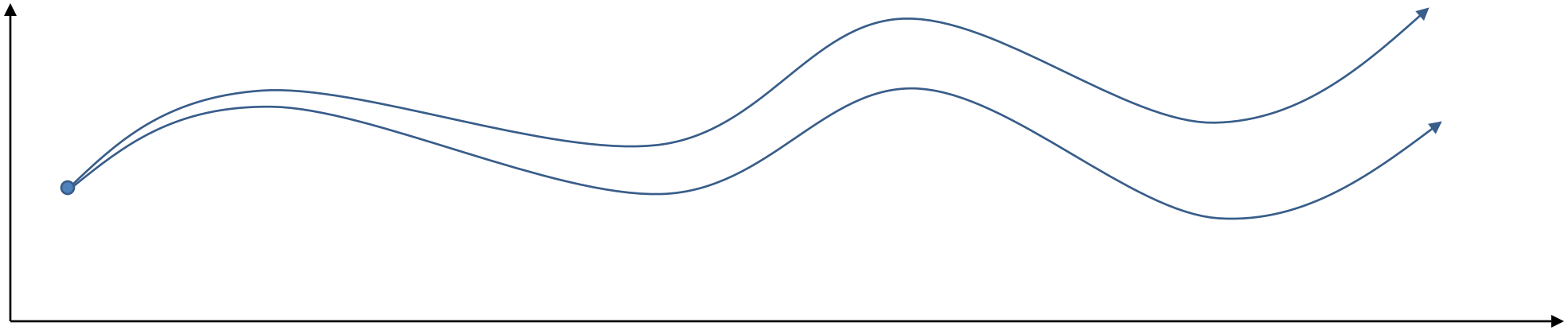
the action at the first timestep has a large effect on the states at the end of the trajectory

small gradients here

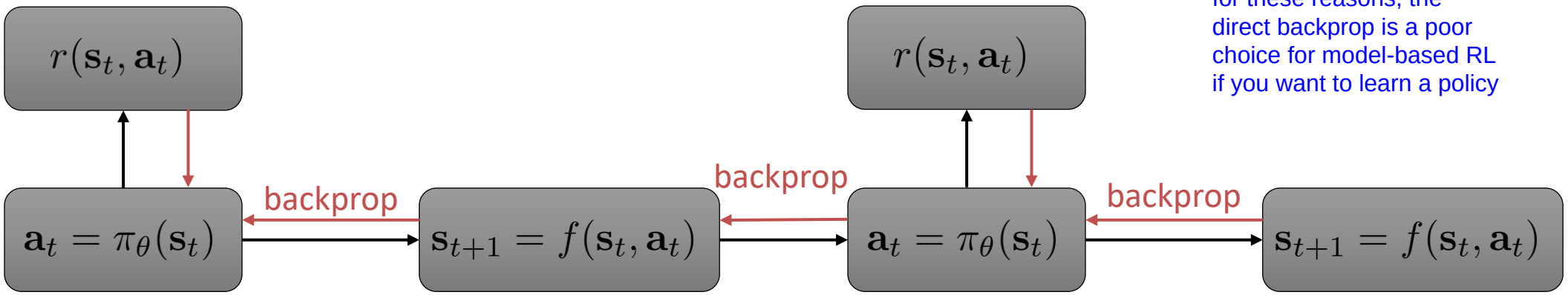
the action at the near the end of the trajectory doesn't have as big of an effect on the trajectory

What's the problem with backprop into policy?





What's the problem with backprop into policy?



for these reasons, the direct backprop is a poor choice for model-based RL if you want to learn a policy

- Similar parameter sensitivity problems as shooting methods
  - But no longer have convenient second order LQR-like method, because policy parameters couple all the time steps, so no dynamic programming
- Similar problems to training long RNNs with BPTT

"backpropagation through time"

  - Vanishing and exploding gradients
  - Unlike LSTM, we can't just "choose" a simple dynamics, dynamics are chosen by nature

What's the solution?

- Use derivative-free ("model-free") RL algorithms, with the model used to generate synthetic samples
  - Seems weirdly backwards
  - Actually works very well
  - Essentially "model-based acceleration" for model-free RL
- Use simpler policies than neural nets
  - LQR with learned models (LQR-FLM – **F**itted **L**ocal **M**odels)
  - Train **local** policies to solve simple tasks
  - Combine them into **global** policies via supervised learning

Model-Free Learning With a Model

What's the solution?

this allows you to build very sample-efficient model-free algorithms

you can use the model to generate unlimited synthetic data. a big issue with model-free RL is their high variance, and high variance can be mitigated through a large number of samples. Generating lots of samples in the real world is expensive, but doing so in simulation is much easier.

Use low variance supervised approach to train your model, and then high variance PG approach with a large number of samples generated by your model

- Use derivative-free ("model-free") RL algorithms, with the model used to generate synthetic samples
  - Seems weirdly backwards
  - Actually works very well
  - Essentially "model-based acceleration" for model-free RL
- Use simpler policies than neural nets
  - LQR with learned models (LQR-FLM – **F**itted **L**ocal **M**odels)
  - Train **local** policies to solve simple tasks
  - Combine them into **global** policies via supervised learning

Model-free optimization with a model

Policy gradient: $$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t}) \hat{Q}^\pi_{i,t}$$

the PG is also a gradient of the total reward w.r.t. the policy parameters. So in a sense, the PG is also computing a derivative through the dynamics, it just doesn't require knowing the functional form of the logprob to do this

Backprop (pathwise) gradient: $$\nabla_\theta J(\theta) = \sum_{t=1}^{T} \frac{dr_t}{d\mathbf{s}_t} \prod_{t'=2}^{t} \frac{d\mathbf{s}_{t'}}{d\mathbf{a}_{t'-1}} \frac{d\mathbf{a}_{t'-1}}{d\mathbf{s}_{t'-1}}$$

the pathwise gradient gives you the same thing.

- Policy gradient might be more *stable* (if enough samples are used) because it does not require multiplying many Jacobians
- See a recent analysis here:
  - Parmas et al. ‘18: PIPP: Flexible Model-Based Policy Search Robust to the Curse of Chaos

a practical approach is to train your model and essentially do model-based RL 2.0, but instead of the backpropagation, compute the PG using REINFORCE via samples generated by the model!

this is another class of methods that use a model together with model-free algorithms.

this uses value-based methods but can be adapted for AC methods too

Model-free optimization with a model

Dyna

the original Dyna algo is not typically used today

online Q-learning algorithm that performs model-free RL with a model

1. given state s, pick action a using exploration policy
2. observe s' and r, to get transition (s, a, s', r)
3. update model $\hat{p}(s'|s, a)$ and $\hat{r}(s, a)$ using (s, a, s')
4. Q-update: $Q(s, a) \leftarrow Q(s, a) + \alpha E_{s',r}[r + \max_{a'} Q(s', a') - Q(s, a)]$
5. repeat K times:
   6. sample $(s, a) \sim \mathcal{B}$ from buffer of past states and actions
   7. Q-update: $Q(s, a) \leftarrow Q(s, a) + \alpha E_{s',r}[r + \max_{a'} Q(s', a') - Q(s, a)]$

just like in regular online Q-learning

Step 4: train up the Q-function based on the transition you just did

this is the fun part. let's use our model to train up our Q-function even more

Step 5-7: train up the Q-function on states/ transitions that you didn't just see

in the original Dyna algo, it suggests to use the action that was in the buffer. in more modern versions, you can choose the action that your new epsilon greedy policy would have taken

then, using your model, estimate the next state s' and the reward, then perform an update

Richard S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming.

there are a number of different algorithms that use some variant of this general recipe!

eneral "Dyna-style" model-based RL recipe

we prefer to do this because distributional shift accumulates over time. shorter rollouts have less distributional shift! so this reduces model error!

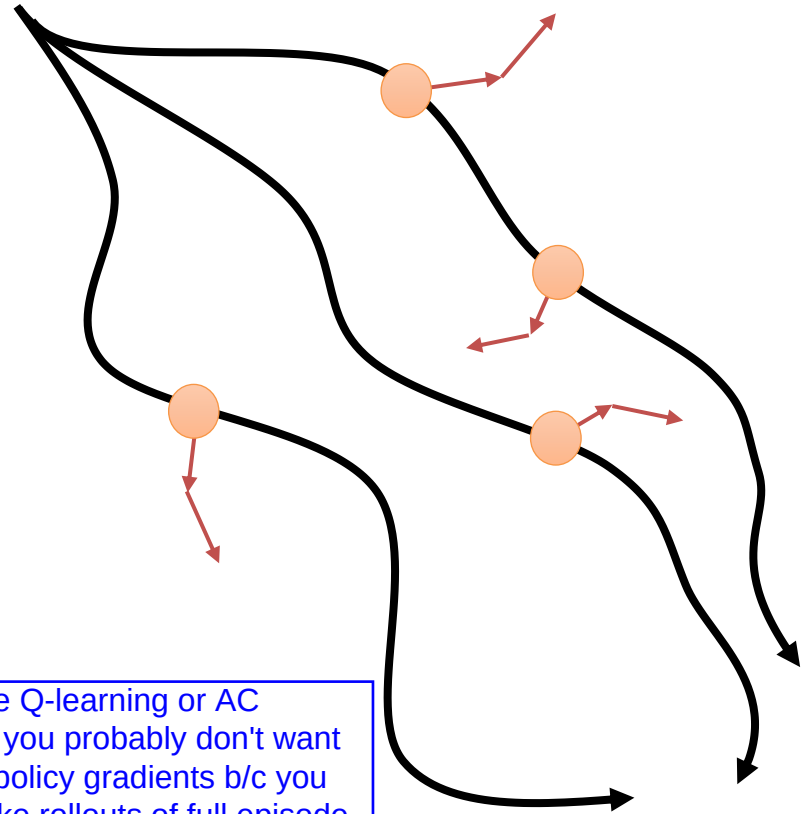
this doesn't have to be online

1. collect some data, consisting of transitions (s, a, s', r)
2. learn model $\hat{p}(s'|s, a)$ (and optionally, $\hat{r}(s, a)$)
3. repeat K times:
   4. sample $s \sim \mathcal{B}$ from buffer
   5. choose action a (from $\mathcal{B}$, from π, or random)
   6. simulate $s' \sim \hat{p}(s'|s, a)$ (and $r = \hat{r}(s, a)$)
   7. train on (s, a, s', r) with model-free RL
   8. (optional) take N more model-based steps

use all of that data with your favorite supervised learning algo to learn your model. and if you don't know the reward function, also learn that

use your model to simulate the next state (and reward)

you can use Q-learning or AC algorithms. you probably don't want to use MC policy gradients b/c you have to make rollouts of full episode length



+ only requires short (as few as one step) rollouts from model

+ still sees diverse states

you don't take short rollouts all from the beginning. we take them from different states we've actually seen from our data

instead of only taking one state, you could take multiple steps from each state in the buffer

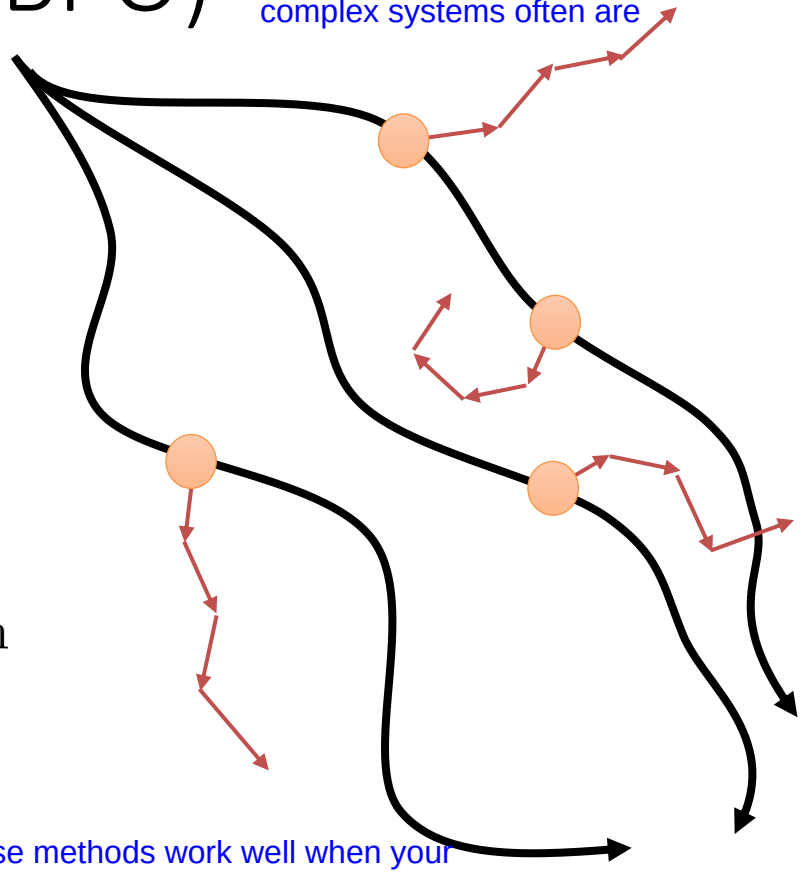
Model-Based Acceleration (MBA)
Model-Based Value Expansion (MVE)
Model-Based Policy Optimization (MBPO)

three algos, sorted from oldest to newest, that use some version of our dyna recipe on the last slide

In general this approach is a good approach if you want to hybridize models and model-free learning, especially if you think your model is only good for shorter-horizon rollouts, which learned models in complex systems often are

1. take some action $\mathbf{a}_i$ and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$, add it to $\mathcal{B}$
2. sample mini-batch $\{\mathbf{s}_j, \mathbf{a}_j, \mathbf{s}'_j, r_j\}$ from $\mathcal{B}$ uniformly
3. use $\{\mathbf{s}_j, \mathbf{a}_j, \mathbf{s}'_j\}$ to update model $\hat{p}(\mathbf{s}'|\mathbf{s}, \mathbf{a})$
4. sample $\{\mathbf{s}_j\}$ from $\mathcal{B}$
5. for each $\mathbf{s}_j$, perform model-based rollout with $\mathbf{a} = \pi(\mathbf{s})$
6. use all transitions $(\mathbf{s}, \mathbf{a}, \mathbf{s}', r)$ along rollout to update Q-function

or actor in AC



these methods work well when your model is decent for short rollouts. they still rely heavily on real-world rollouts to visit interesting states.

+ why is this a *good* idea?

- why is this a *bad* idea?

Gu et al. Continuous deep Q-learning with model-based acceleration. ‘16
Feinberg et al. Model-based value expansion. ’18
Janner et al. When to trust your model: model-based policy optimization. ‘19

Local Models

another way to mitigate instability without using model-free algos

first we'll talk about doing efficient model-based RL with simple policy classes, time-variant linear controllers, like LQR. Then we can use these simple policy classes to obtain more complex global policies

What's the solution?

- Use derivative-free ("model-free") RL algorithms, with the model used to generate synthetic samples
  - Seems weirdly backwards
  - Actually works very well
  - Essentially "model-based acceleration" for model-free RL
- Use simpler policies than neural nets
  - LQR with learned models (LQR-FLM – **F**itted **L**ocal **M**odels)
  - Train **local** policies to solve simple tasks
  - Combine them into **global** policies via supervised learning

What's the solution?

- Use derivative-free ("model-free") RL algorithms, with the model used to generate synthetic samples
  - Seems weirdly backwards
  - Actually works very well
  - Essentially "model-based acceleration" for model-free RL
- Use simpler policies than neural nets
  - LQR with learned models (LQR-FLM – **F**itted **L**ocal **M**odels)
  - Train **local** policies to solve simple tasks
  - Combine them into **global** policies via supervised learning

a local model is one that is valid within the neighborhood of a trajectory.

Local models

before, when talking about LQR, if you know the dynamics f, you could use gradient based methods to obtain a locally optimal policy represented by a time-varying linear controller.

if you don't know the model, you don't know the derivatives.

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

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

usual story: differentiate via backpropagation and optimize!

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

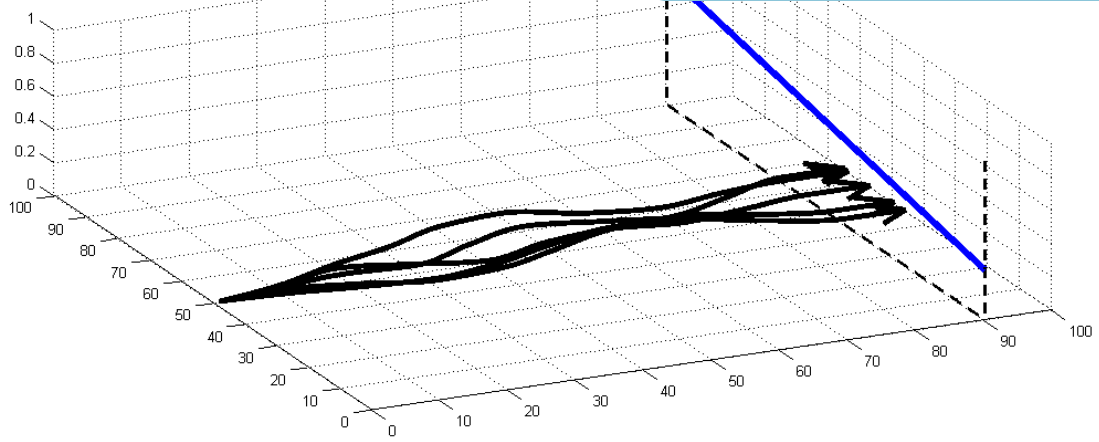
Local models

so what if we fit df/dx and df/du around a current trajectory or policy

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

you could take an initial time-varying linear policy and rollout multiple trajectories. Then you could empirically fit df/dx and df/du to the data for each timestep.

idea: just fit $\frac{df}{d\mathbf{x}_t}, \frac{df}{d\mathbf{u}_t}$ around current trajectory or policy!



LQR gives us a linear feedback controller

can **execute** in the real world!

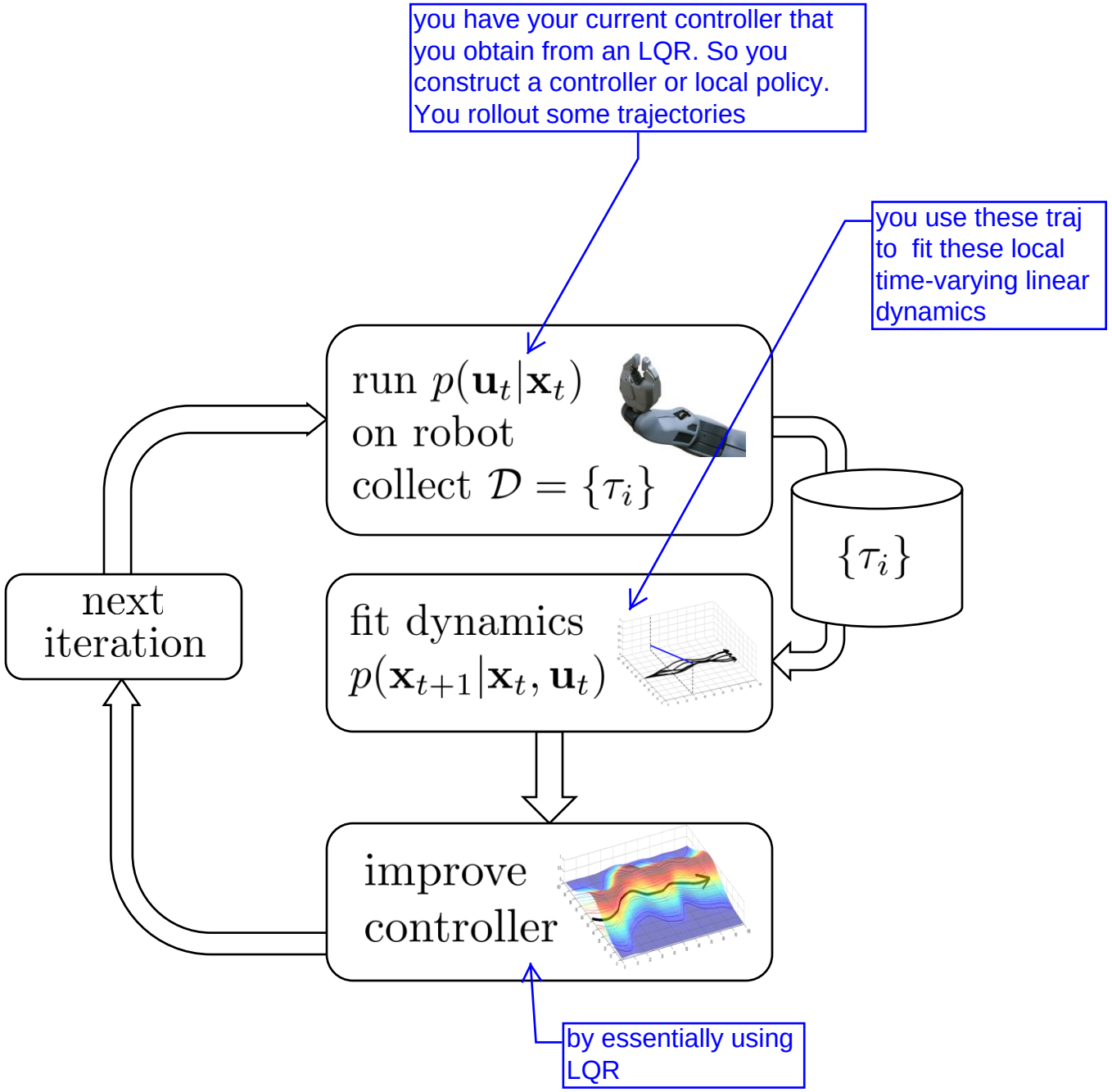
Local models

LQR-FM

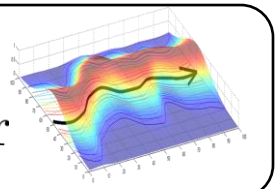
$$p(\mathbf{x}_{t+1}|\mathbf{x}_t, \mathbf{u}_t) = \mathcal{N}(f(\mathbf{x}_t, \mathbf{u}_t), \Sigma)$$

$$f(\mathbf{x}_t, \mathbf{u}_t) \approx \mathbf{A}_t \mathbf{x}_t + \mathbf{B}_t \mathbf{u}_t$$

$$\mathbf{A}_t = \frac{df}{d\mathbf{x}_t} \qquad \mathbf{B}_t = \frac{df}{d\mathbf{u}_t}$$

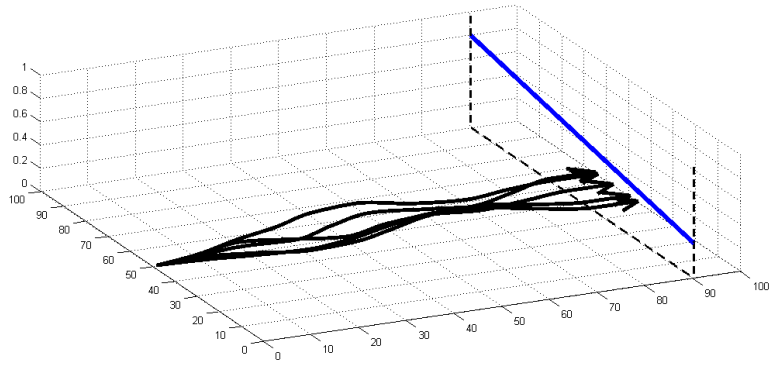


What controller to execute?

improve controller 

iLQR produces: $\hat{\mathbf{x}}_t,\ \hat{\mathbf{u}}_t,\ \mathbf{K}_t,\ \mathbf{k}_t$

$$\mathbf{u}_t = \mathbf{K}_t(\mathbf{x}_t - \hat{\mathbf{x}}_t) + \mathbf{k}_t + \hat{\mathbf{u}}_t$$



Version 0.5: $p(\mathbf{u}_t|\mathbf{x}_t) = \delta(\mathbf{u}_t = \hat{\mathbf{u}}_t)$

Doesn't correct deviations or drift

Version 1.0: $p(\mathbf{u}_t|\mathbf{x}_t) = \delta(\mathbf{u}_t = \mathbf{K}_t(\mathbf{x}_t - \hat{\mathbf{x}}_t) + \mathbf{k}_t + \hat{\mathbf{u}}_t)$

Better, but maybe a little too good?

Version 2.0: $p(\mathbf{u}_t|\mathbf{x}_t) = \mathcal{N}(\mathbf{K}_t(\mathbf{x}_t - \hat{\mathbf{x}}_t) + \mathbf{k}_t + \hat{\mathbf{u}}_t, \Sigma_t)$

Add noise so that all samples don't look the same!

Set $\Sigma_t = \mathbf{Q}^{-1}_{\mathbf{u}_t,\mathbf{u}_t}$

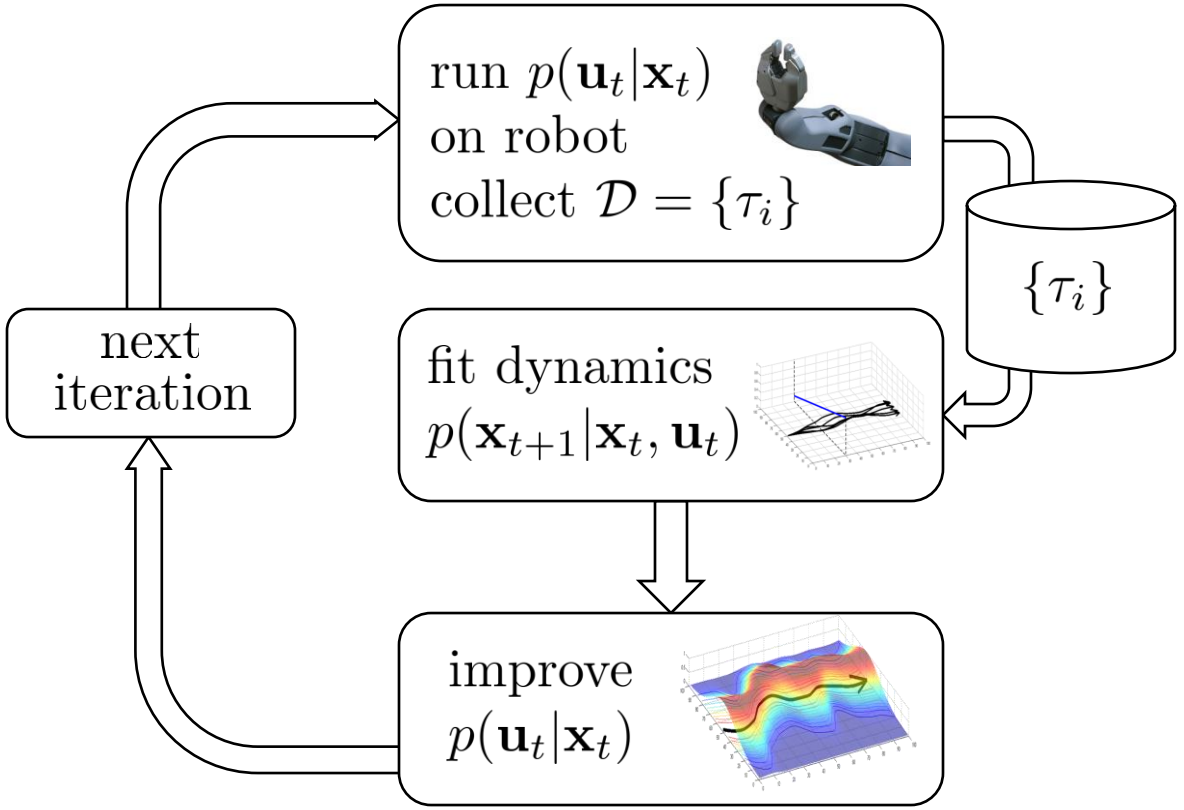
Local models

Run your controller to get a sample of trajectories.
Use those trajectories to fit your dynamics.
Use those samples to improve your controller
Repeat

$$p(\mathbf{x}_{t+1}|\mathbf{x}_t, \mathbf{u}_t) = \mathcal{N}(f(\mathbf{x}_t, \mathbf{u}_t), \Sigma)$$

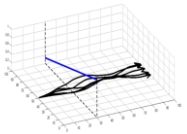
$$f(\mathbf{x}_t, \mathbf{u}_t) \approx \mathbf{A}_t\mathbf{x}_t + \mathbf{B}_t\mathbf{u}_t$$

$$\mathbf{A}_t = \frac{df}{d\mathbf{x}_t} \qquad \mathbf{B}_t = \frac{df}{d\mathbf{u}_t}$$



How to fit the dynamics?

fit dynamics
$p(\mathbf{x}_{t+1}|\mathbf{x}_t, \mathbf{u}_t)$



$\{(\mathbf{x}_t, \mathbf{u}_t, \mathbf{x}_{t+1})_i\}$

fit $p(\mathbf{x}_{t+1}|\mathbf{x}_t, \mathbf{u}_t)$ at each time step using linear regression

$$p(\mathbf{x}_{t+1}|\mathbf{x}_t, \mathbf{u}_t) = \mathcal{N}(\mathbf{A}_t\mathbf{x}_t + \mathbf{B}_t\mathbf{u}_t + \mathbf{c}, \mathbf{N}_t) \qquad \mathbf{A}_t \approx \frac{df}{d\mathbf{x}_t} \qquad \mathbf{B}_t \approx \frac{df}{d\mathbf{u}_t}$$

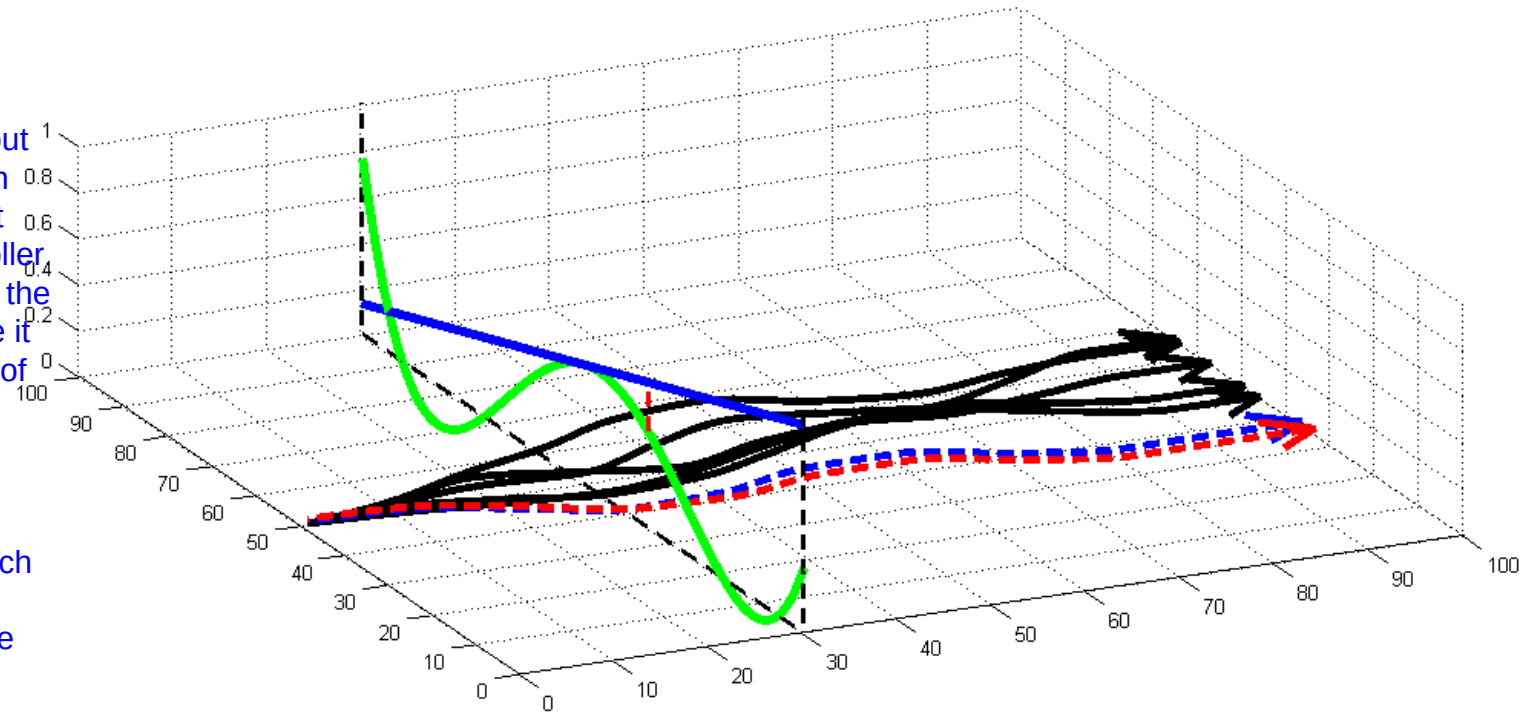
noise

What if we go too far?

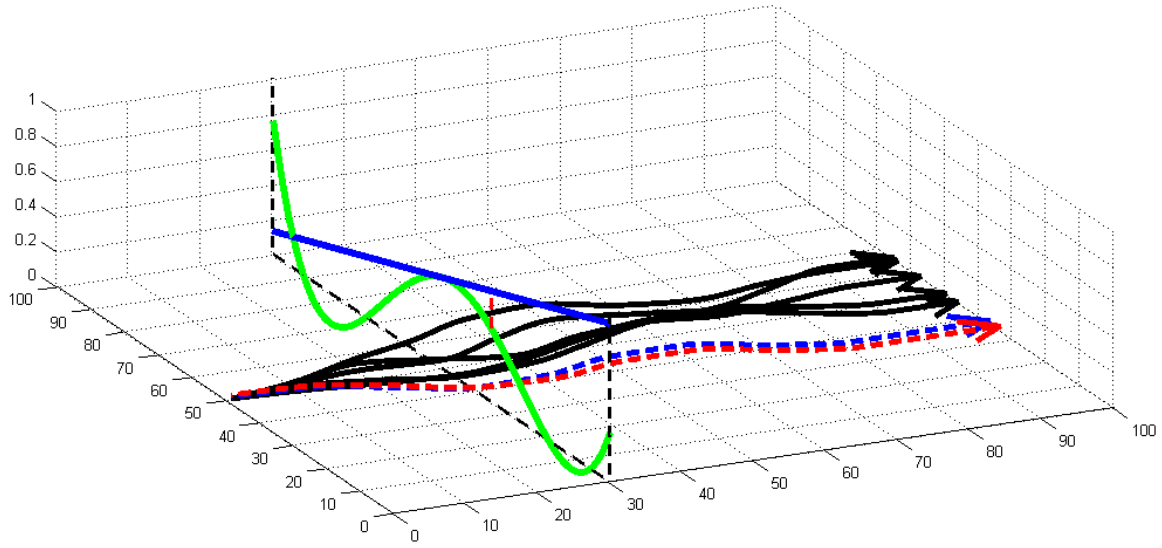
the true dynamics are non-linear, so far away from our data there might be a big discrepency between our linear fit and the true nonlinear dynamics. So if our optimize trajectories wanders in our high error regions, when we roll this out in the real world, we might get an outcome very different than what we expected because our controller won't be able to correctly control the system in these regions because it was optimized for a different set of dynamics.

this could cause divergence

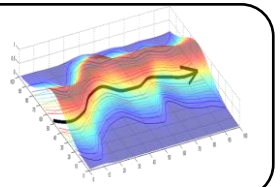
so we need to constrain how much we change the controller. the solution is very similar to what we saw in Natural PG and TRPO



How to stay close to old controller?



improve $p(\mathbf{u}_t|\mathbf{x}_t)$



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

$$p(\tau) = p(\mathbf{x}_1)\prod_{t=1}^{T} p(\mathbf{u}_t|\mathbf{x}_t)p(\mathbf{x}_{t+1}|\mathbf{x}_t, \mathbf{u}_t)$$

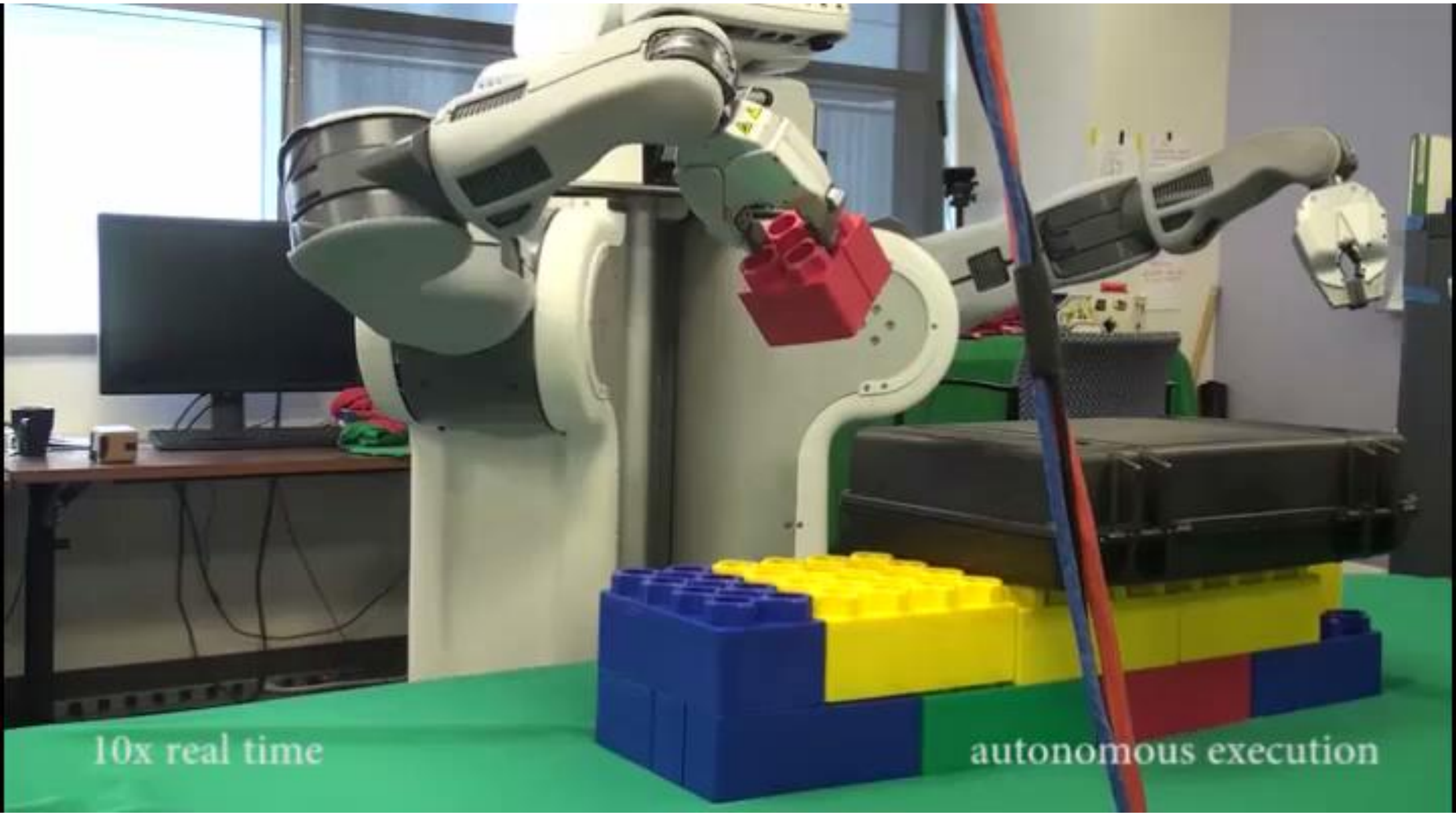
What if the new $p(\tau)$ is "close" to the old one $\bar{p}(\tau)$?

If trajectory distribution is close, then dynamics will be close too!

What does "close" mean? $D_{\mathrm{KL}}(p(\tau)\|\bar{p}(\tau)) \leq \epsilon$

This is easy to do if $\bar{p}(\tau)$ also came from linear controller!

For details, see: "**Learning Neural Network Policies with Guided Policy Search under Unknown Dynamics**"


10x real time
autonomous execution

Global Policies from Local Models

What's the solution?

- Use derivative-free ("model-free") RL algorithms, with the model used to generate synthetic samples
  - Seems weirdly backwards
  - Actually works very well
  - Essentially "model-based acceleration" for model-free RL
- Use simpler policies than neural nets
  - LQR with learned models (LQR-FLM – **F**itted **L**ocal **M**odels)
  - Train **local** policies to solve simple tasks
  - Combine them into **global** policies via supervised learning

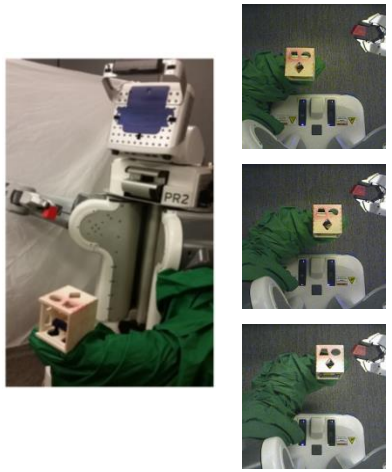
What's the solution?

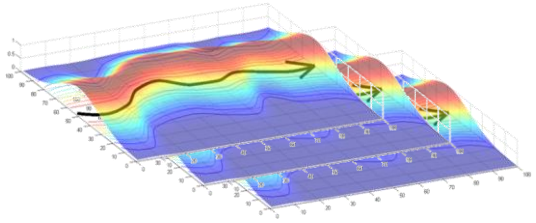
- Use derivative-free ("model-free") RL algorithms, with the model used to generate synthetic samples
  - Seems weirdly backwards
  - Actually works very well
  - Essentially "model-based acceleration" for model-free RL
- Use simpler policies than neural nets
  - LQR with learned models (LQR-FLM – **F**itted **L**ocal **M**odels)
  - Train **local** policies to solve simple tasks
  - Combine them into **global** policies via supervised learning

Guided policy search: high-level idea

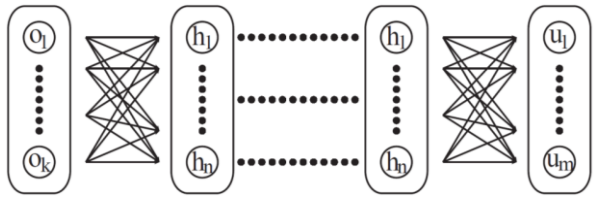
learn a separate local policy for every lego starting place

use local policy as data. then create global model using all of this data

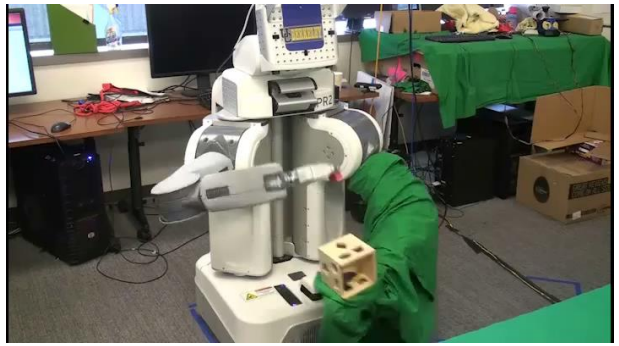


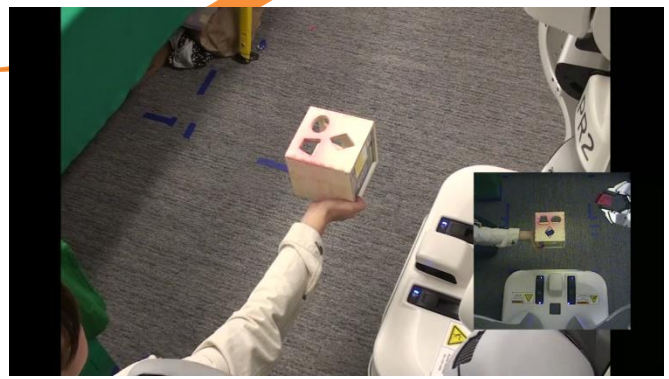


trajectory-centric RL



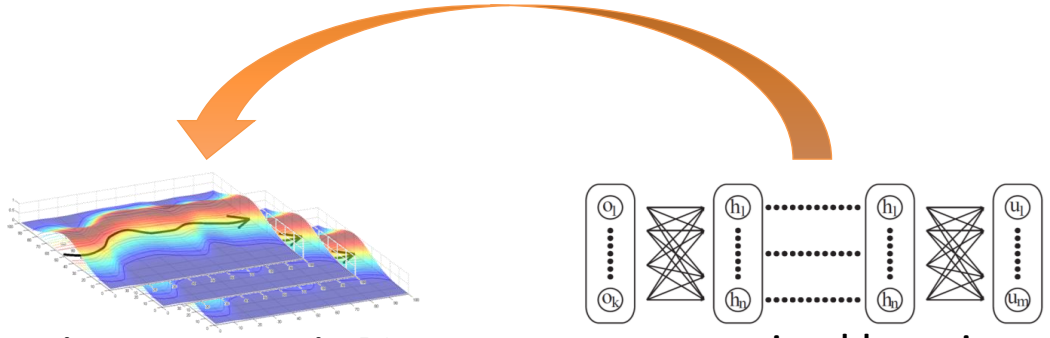
supervised learning





Guided policy search: algorithm sketch

idea: it's easier to create simpler, local solutions, then synthesize these solutions together to get a global solution



where i is the ith local policy

global policy

modified cost to keep $\pi_{\mathrm{LQR},i}$ close to π_θ

1. optimize each local policy $\pi_{\mathrm{LQR},i}(\mathbf{u}_t|\mathbf{x}_t)$ on initial state $\mathbf{x}_{0,i}$ w.r.t. $\tilde{c}_{k,i}(\mathbf{x}_t, \mathbf{u}_t)$
2. use samples from step (1) to train $\pi_\theta(\mathbf{u}_t|\mathbf{x}_t)$ to mimic each $\pi_{\mathrm{LQR},i}(\mathbf{u}_t|\mathbf{x}_t)$
3. update cost function $\tilde{c}_{k+1,i}(\mathbf{x}_t, \mathbf{u}_t) = c(\mathbf{x}_t, \mathbf{u}_t) + \lambda_{k+1,i} \log \pi_\theta(\mathbf{u}_t|\mathbf{x}_t)$

Repeating this process, every step of the way each local policy will improve b/c it takes another step in the LQR-FM algorithm. The global policy improves because it learns on more data, and the pi_LQR solutions will learn to be closer to pi_theta

Lagrange multiplier

this enforces a KL divergence constraint

For details, see: "End-to-End Training of Deep Visuomotor Policies"

Underlying principle: distillation

Ensemble models: single models are often not the most robust – instead train many models and average their predictions

this is how most ML competitions (e.g., Kaggle) are won

this is very expensive at test time

Can we make a single model that is as good as an ensemble?

Distillation: train on the ensemble's predictions as "soft" targets

logit

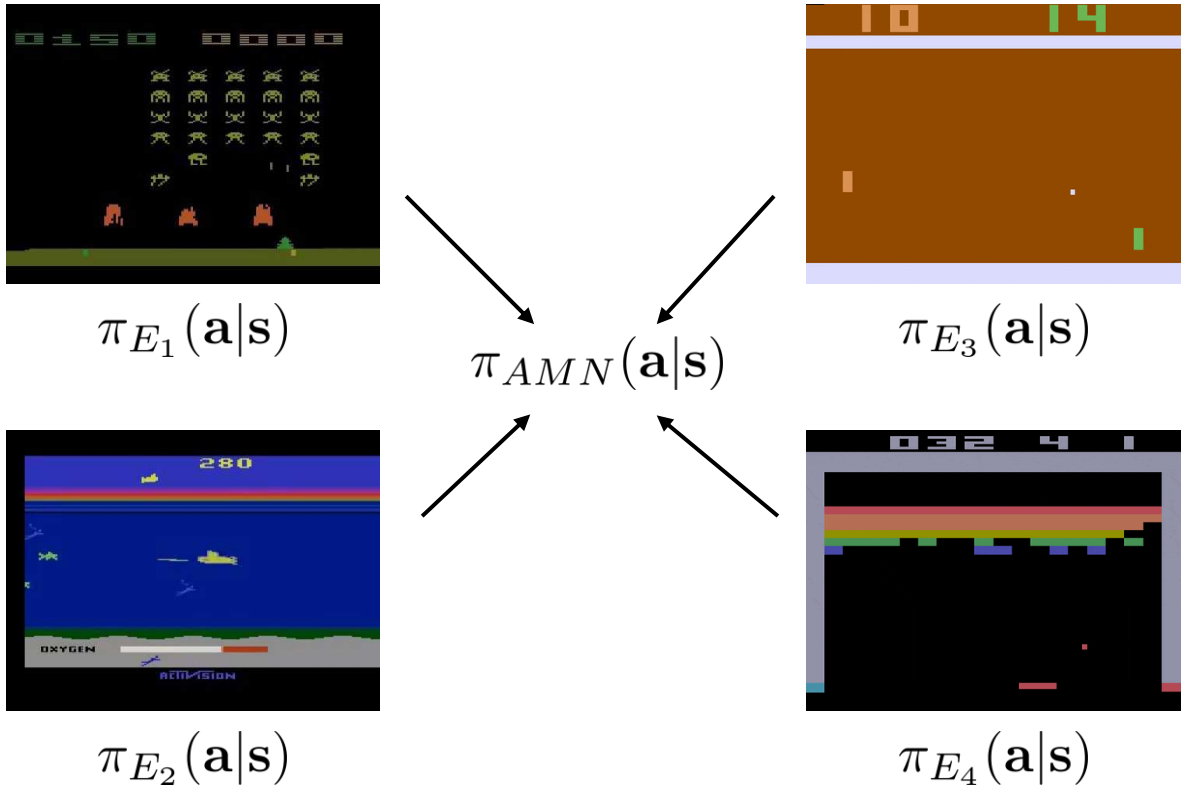
$$p_i = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}$$

temperature

Intuition: more knowledge in soft targets than hard labels!

Slide adapted from G. Hinton, see also Hinton et al. "Distilling the Knowledge in a Neural Network"

Distillation for Multi-Task Transfer



$$\mathcal{L} = \sum_{\mathbf{a}} \pi_{E_i}(\mathbf{a}|\mathbf{s}) \log \pi_{AMN}(\mathbf{a}|\mathbf{s})$$

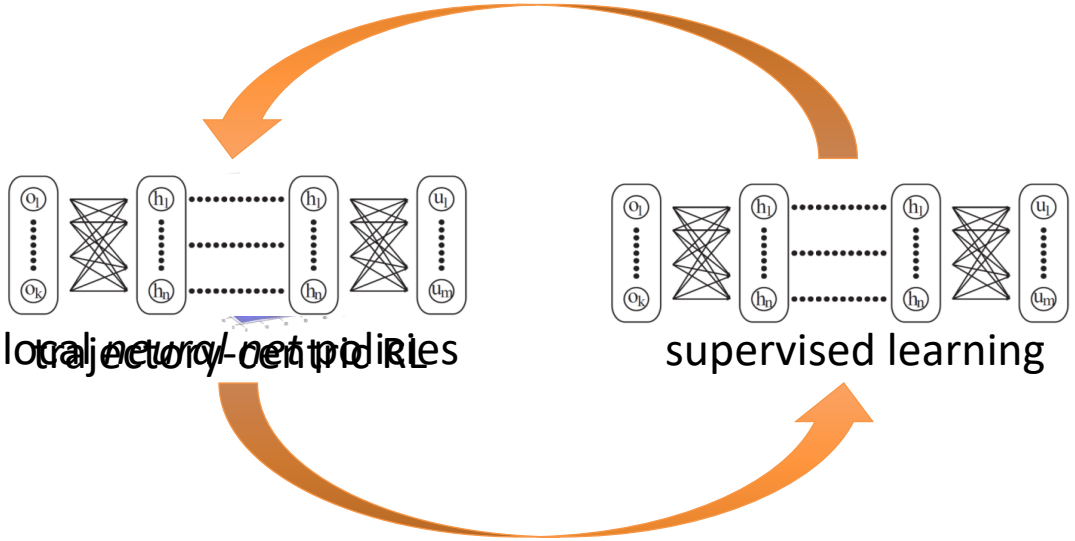
(just supervised learning/distillation)

analogous to guided policy search, but for multi-task learning

some other details
(e.g., feature regression objective)
– see paper

Parisotto et al. “Actor-Mimic: Deep Multitask and Transfer Reinforcement Learning”

Combining weak policies into a strong policy



Divide and Conquer Reinforcement Learning

Divide and conquer reinforcement learning algorithm sketch:

1. optimize each local policy $\pi_{\theta_i}(\mathbf{a}_t|\mathbf{s}_t)$ on initial state $\mathbf{s}_{0,i}$ w.r.t. $\tilde{r}_{k,i}(\mathbf{s}_t, \mathbf{a}_t)$
2. use samples from step (1) to train $\pi_\theta(\mathbf{u}_t|\mathbf{x}_t)$ to mimic each $\pi_{\theta_i}(\mathbf{u}_t|\mathbf{x}_t)$
3. update reward function $\tilde{r}_{k+1,i}(\mathbf{x}_t, \mathbf{u}_t) = r(\mathbf{x}_t, \mathbf{u}_t) + \lambda_{k+1,i} \log \pi_\theta(\mathbf{u}_t|\mathbf{x}_t)$

For details, see: “Divide and Conquer Reinforcement Learning”

Readings: guided policy search & distillation

- L.*, Finn*, et al. End-to-End Training of Deep Visuomotor Policies. 2015.
- Rusu et al. Policy Distillation. 2015.
- Parisotto et al. Actor-Mimic: Deep Multitask and Transfer Reinforcement Learning. 2015.
- Ghosh et al. Divide-and-Conquer Reinforcement Learning. 2017.
- Teh et al. Distral: Robust Multitask Reinforcement Learning. 2017.