

Trustworthy AI Autonomy

M2-2: Model-based decision making

Ding Zhao

Assistant Professor
Carnegie Mellon University

Plan for today

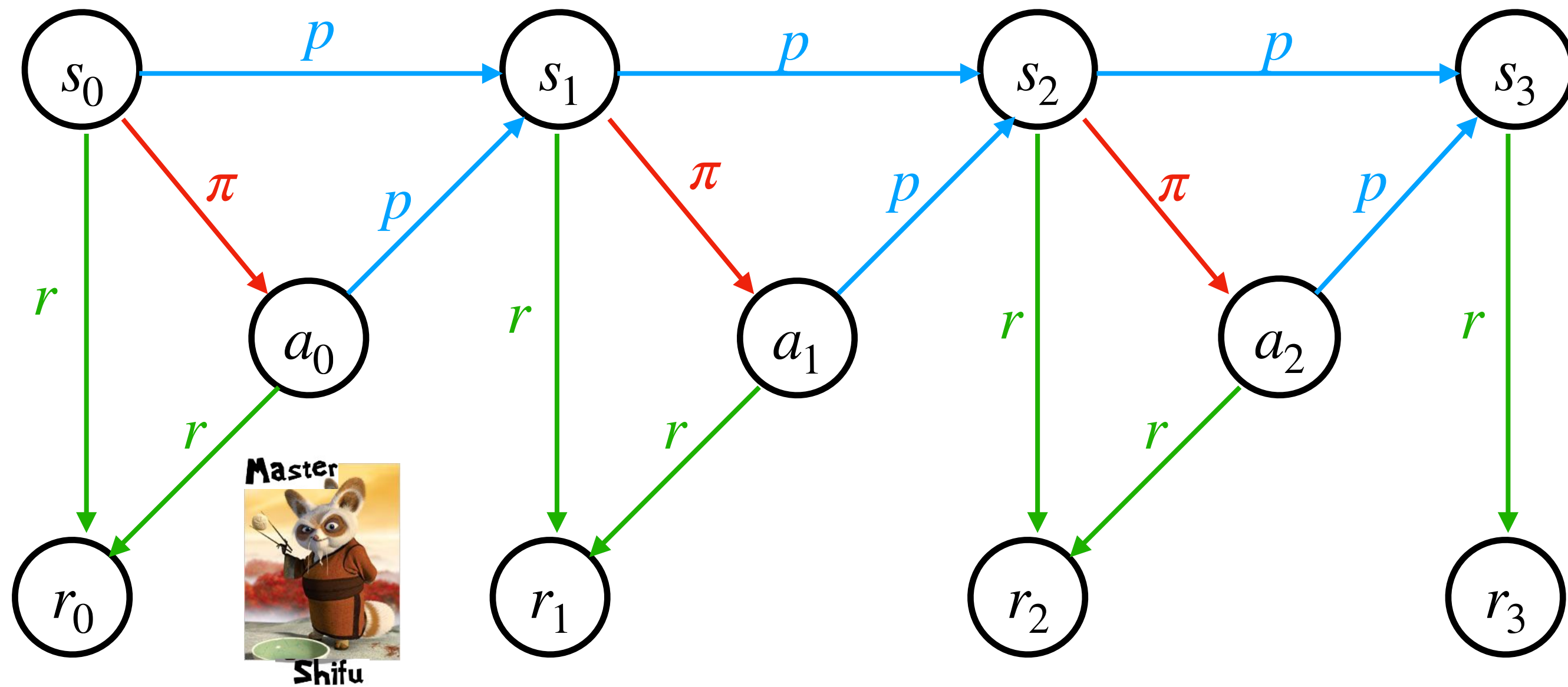
- Model-based control
 - LQR, iLQR, MPC
- Model-based reinforcement learning
 - Neural network based method
 - Local (linearized) model
 - Planning: Cross Entropy Method
 - Gaussian process-based Reinforcement learning (next lecture)

Recap: On-policy vs off-policy

- Policy optimization is almost always performed **on-policy**, which means that each update only uses data collected while acting according to the most recent version of the **policy**. The historical data collected with very old policy is not used. They can be used with both continuous and discrete states. Using gradient, they converge to a local minima of $J(\theta)$
- Q-learning, e.g., DQN, is almost always performed off-policy, which means that each update can use data collected during the whole training history, regardless of what policy the agent was choosing to explore the environment. Therefore, it is more sampling efficient. No guarantee of convergence.



Recap: MDP/Reinforcement Learning

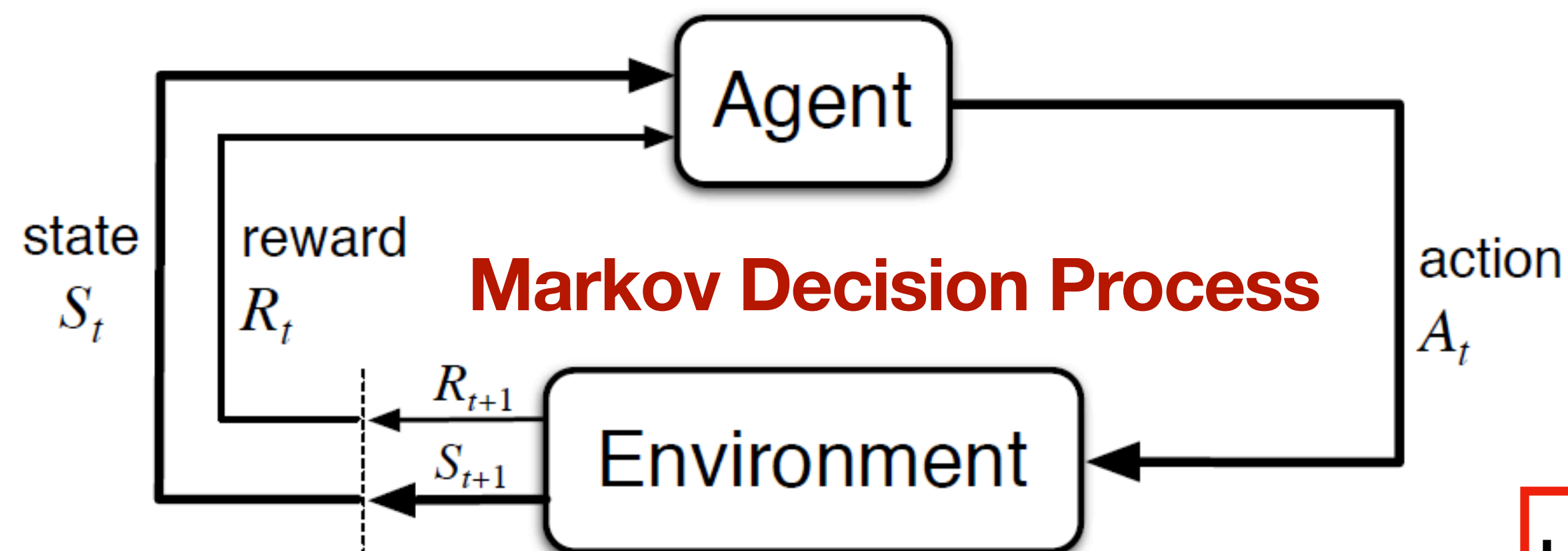


- Instead of asking for demos, we only request a single digit number r_t to indicate the level of happiness - reward.

$$s_{t+1} \sim p(\cdot | s_t, a_t)$$

$$a_t \sim \pi(\cdot | s_t)$$

$$r_t \sim r(\cdot | s_t, a_t)$$



Here $p(s_{t+1} | s_t, a_t)$ is called the **model**

How to get the model?

- Often we do know the dynamics
 - Well-studied systems, e.g., automotive
 - Optimal control

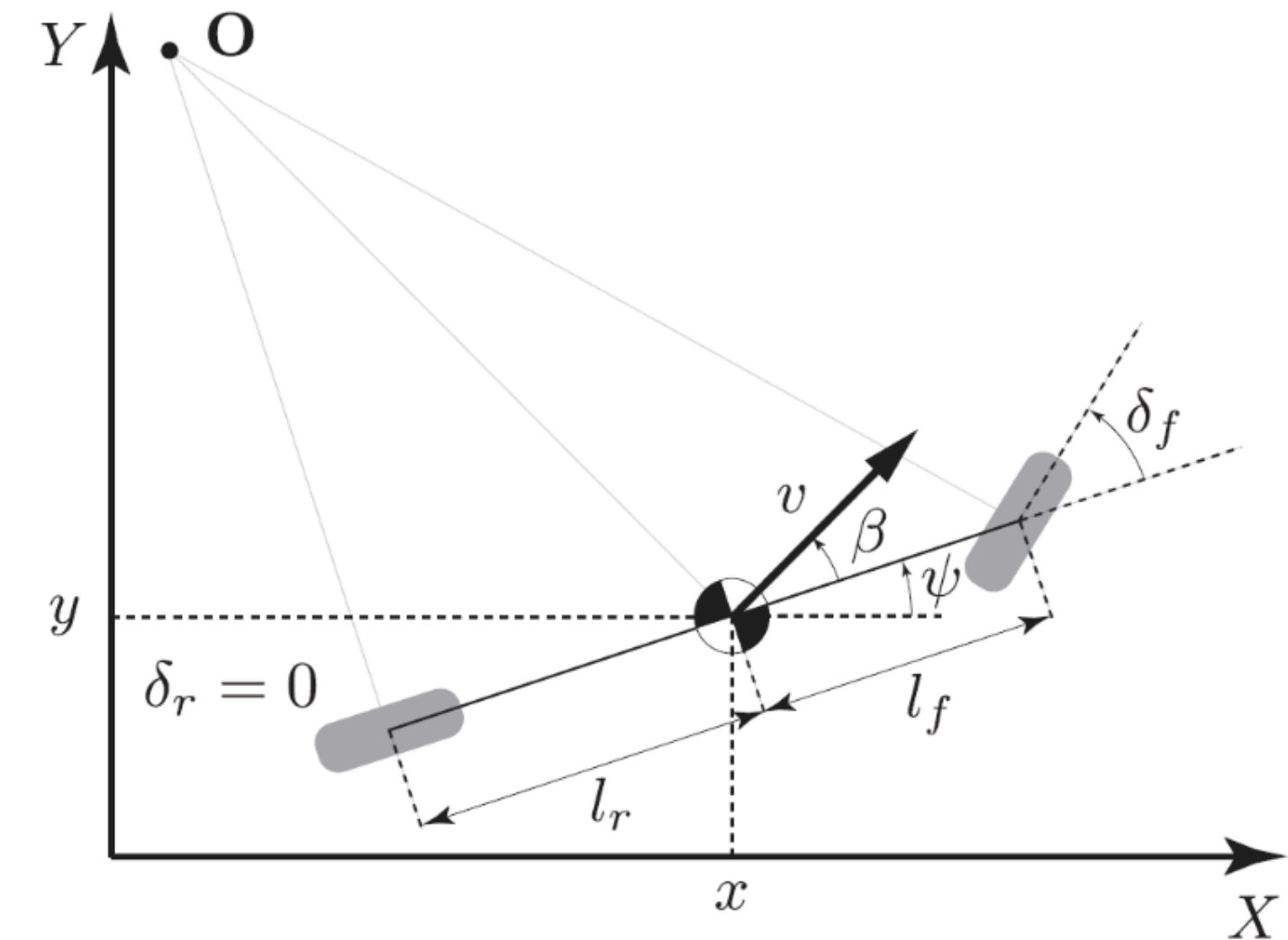


Figure 1: Bicycle model[2]

$$\dot{x} = f(x, u, t) \quad \dot{x} = Ax + Bu$$

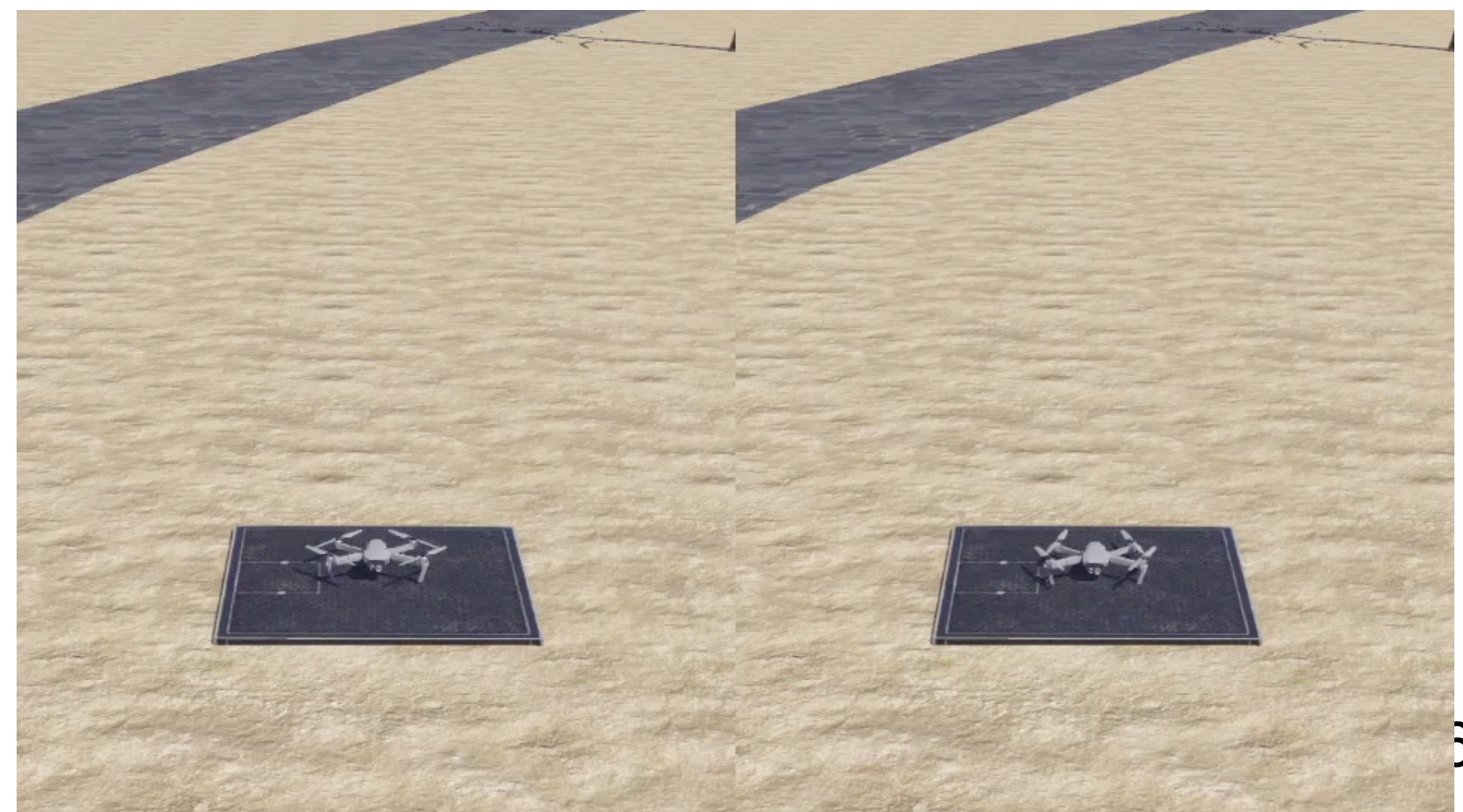
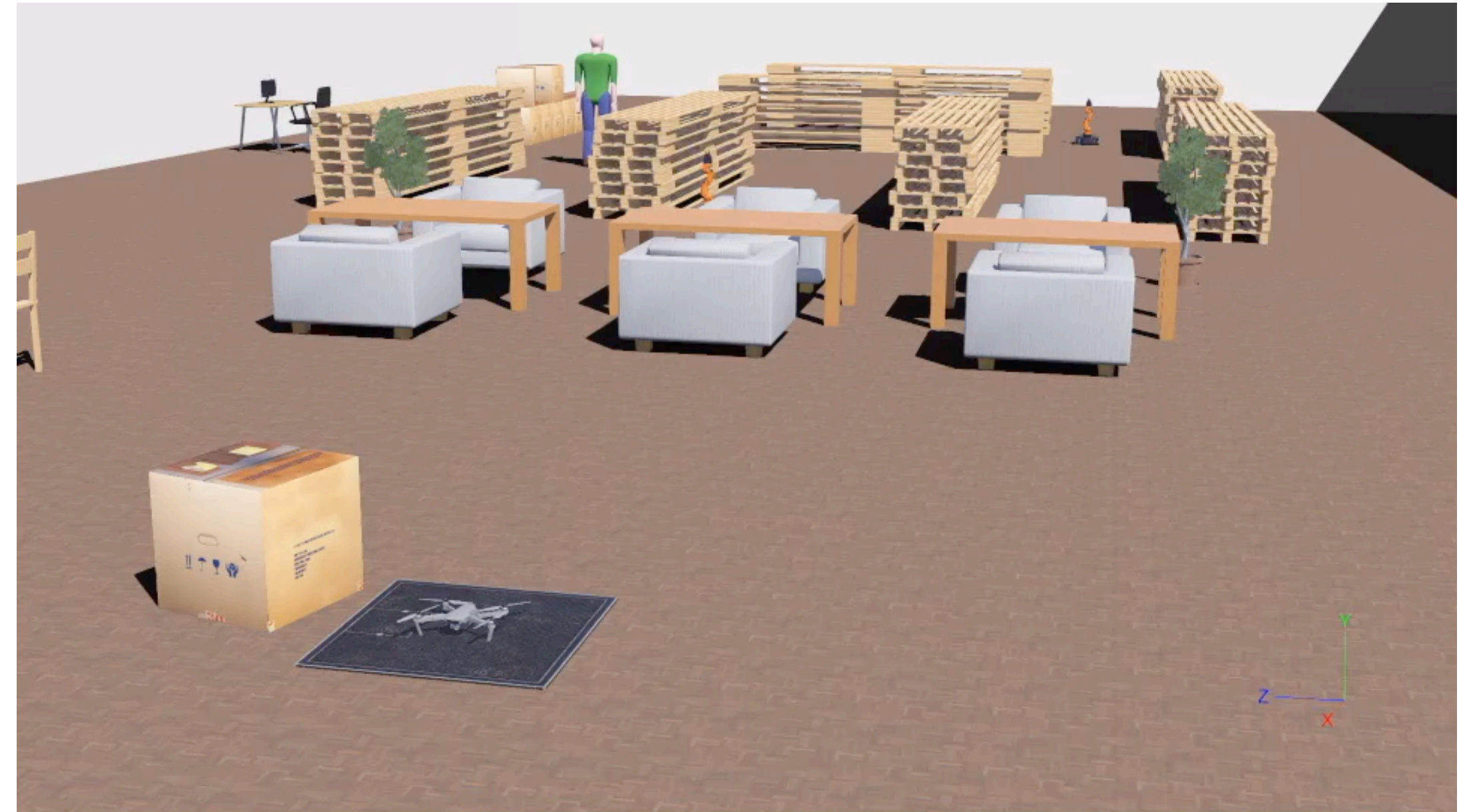
$$y = g(x, u, t) \quad y = Cx + Du$$

$$\frac{d}{dt} s_1 = \frac{d}{dt} \begin{bmatrix} y \\ \dot{y} \\ \psi \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & \frac{-4C_\alpha}{m\dot{x}} & 0 & -\dot{x} + \frac{2C_\alpha(l_r - l_f)}{m\dot{x}} \\ 0 & 0 & 0 & 1 \\ 0 & \frac{2(l_r - l_f)C_\alpha}{I_z\dot{x}} & 0 & -\frac{2(l_f^2 + l_r^2)C_\alpha}{I_z\dot{x}} \end{bmatrix} \begin{bmatrix} y \\ \dot{y} \\ \psi \\ \dot{\psi} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ \frac{2C_\alpha}{m} & 0 \\ 0 & 0 \\ \frac{2l_f C_\alpha}{I_z} & 0 \end{bmatrix} \begin{bmatrix} \delta \\ F \end{bmatrix}$$

$$\frac{d}{dt} s_2 = \frac{d}{dt} \begin{bmatrix} x \\ \dot{x} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & \frac{1}{m} \end{bmatrix} \begin{bmatrix} \delta \\ F \end{bmatrix} + \begin{bmatrix} 0 \\ \psi\dot{y} - fg \end{bmatrix}$$

Where to get the model?

- **Often we do know the dynamics**
 - Well-studied systems, e.g., automotive
 - Optimal control
- **We know the structure of the dynamics but need to fit some parameters**
 - System identification: fit unknown parameters of a known model structure, e.g., estimation of the road friction, abrupt changes
 - Adaptive control: the model may not be accurately estimated but the control error vanishes



Where to get the model?

- **We do know the dynamics**
 - Well-studied systems, e.g., automotive
 - Optimal control
- **We know the structure of the dynamics but need to fit some parameters**
 - System identification – fit unknown parameters of a known model, e.g. estimation of the road friction, abrupt changes
 - Adaptive control: the model may not be accurate but the control error vanishes
- **We can learn the dynamics**
 - Model-based reinforcement learning: Fit a general-purpose model for $p(s_{t+1} | s_t, a_t)$

Aside: notation

\mathbf{s}_t – state

\mathbf{a}_t – action

$r(\mathbf{s}, \mathbf{a})$ – reward function



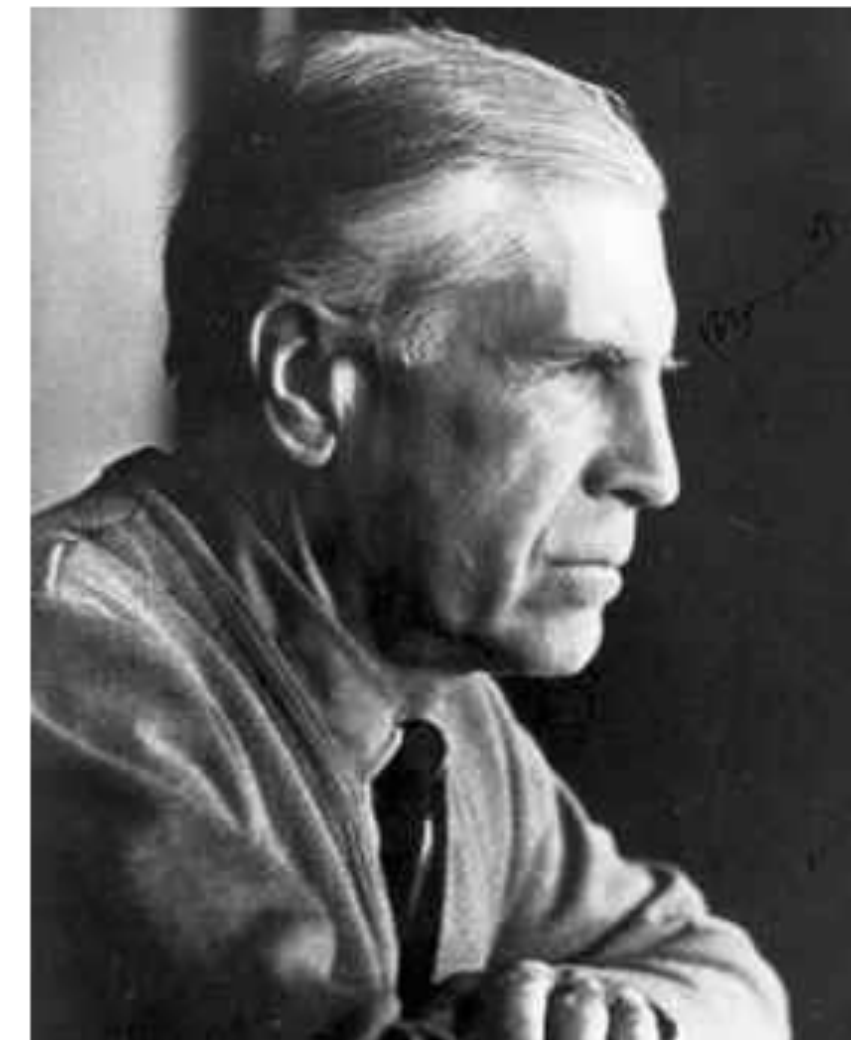
Richard Bellman

$$r(\mathbf{s}, \mathbf{a}) = -c(\mathbf{x}, \mathbf{u})$$

\mathbf{x}_t – state

\mathbf{u}_t – action

$c(\mathbf{x}, \mathbf{u})$ – cost function



Lev Pontryagin

(Finite Horizon Discrete-Time) Linear Quadratic Regulator (LQR)

- Design control policy to minimize the cost function.

$$J_{0,N} = \frac{1}{2}x(N)^T S_N x(N) + \frac{1}{2} \sum_{k=0}^{N-1} (x(k)^T Q x(k) + u(k)^T R u(k))$$

where $S_N, Q, R \geq 0$, subject to the system dynamics

$$x(k+1) = Ax(k) + Bu(k)$$

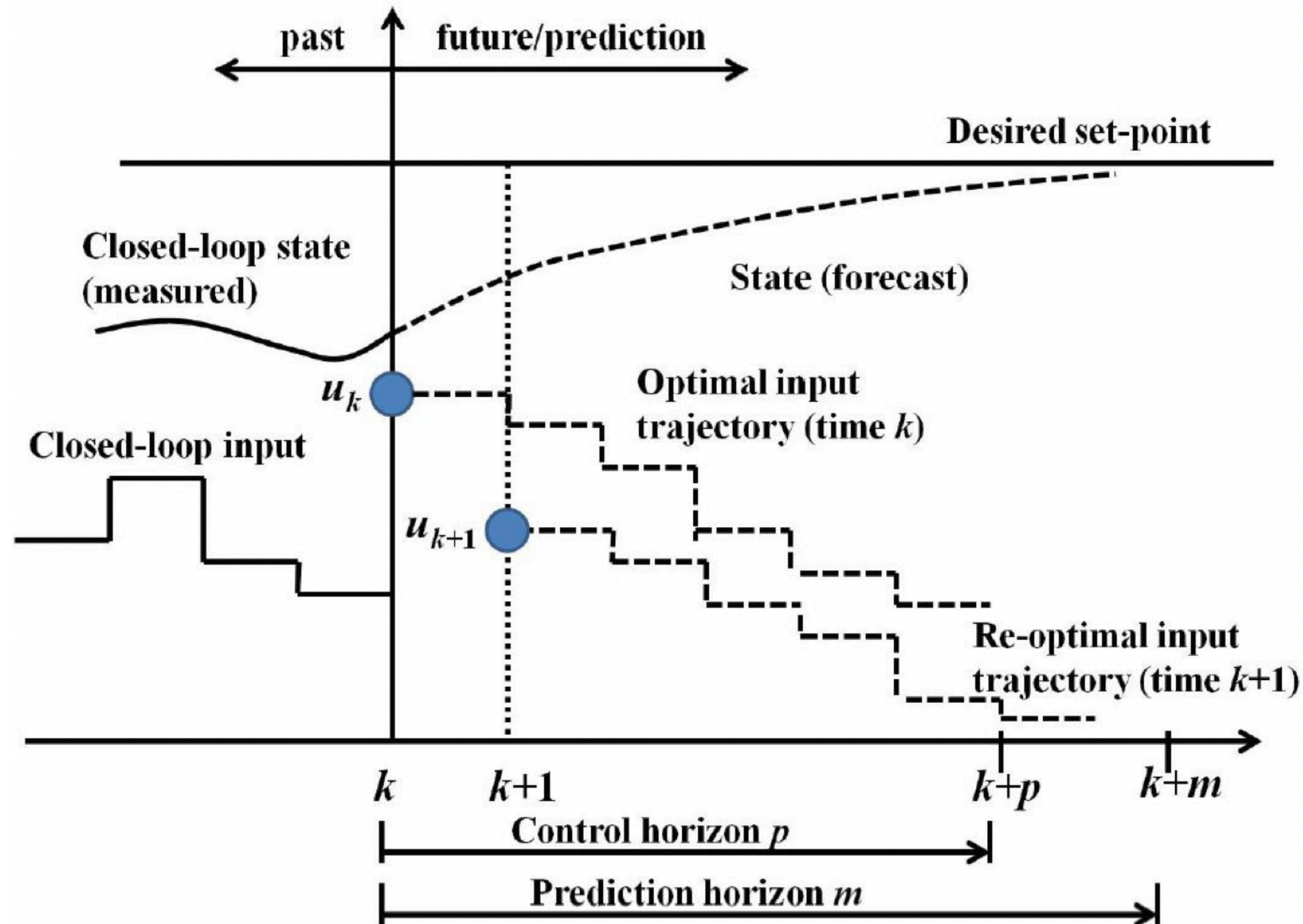
- It is found that the optimal control solution follows an elegant format

$$u^*(k) = K_k x(k) \qquad \min J_{k,N} = J_{k,N}^* = \frac{1}{2}x(k)^T S_k x(k)$$

- where K_k is a constant only dependent on A, B, S, Q, R ,
 $S_N \Rightarrow K_{N-1} \Rightarrow S_{N-1} \Rightarrow K_{N-2} \Rightarrow S_{N-2} \Rightarrow \dots \Rightarrow S_0 (= J_{0,N}^*)$


$$K_k = -(R + B^T S_{k+1} B)^{-1} B^T S_{k+1} A, S_k = (A + BK_k)^T S_{k+1} (A + BK_k) + Q + K_k^T R K_k$$

(FH-DT) LQR vs MPC

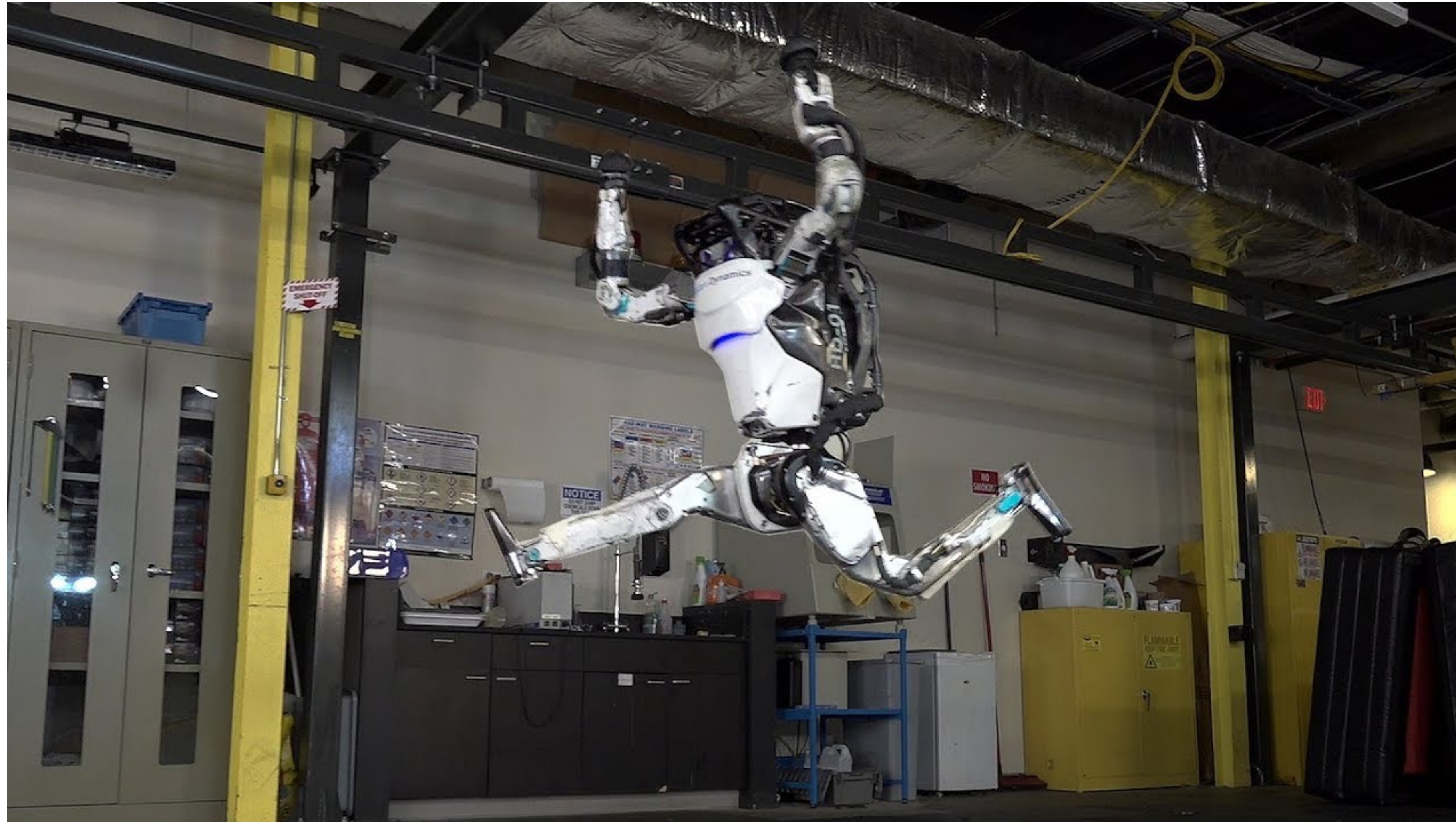


- (Linear) Modal Predictive Control or "Receding Horizon Control"
- Calculate $u^*(k : k + N)$, but only use $u^*(k)$ and recalculate $u^*(k + 1 : k + N + 1)$ in the next step. Essentially, it is a closed loop version of LQR, therefore, it could be more robust by increasing computation budget.

Model Predictive Control

 Boston Dynamics ©
2.47M subscribers

Atlas uses its whole body -- legs, arms, torso -- to perform a sequence of dynamic maneuvers that form a gymnastic routine. We created the maneuvers using new techniques that streamline the development process. First, an optimization algorithm transforms high-level descriptions of each maneuver into dynamically-feasible reference motions. Then Atlas tracks the motions using a model predictive controller that smoothly blends from one maneuver to the next. Using this approach, we developed the routine significantly faster than previous Atlas routines, with a performance success rate of about 80%. For more information visit us at



Atlas uses its whole body -- legs, arms, torso -- to perform a sequence of dynamic maneuvers that form a gymnastic routine. We created the maneuvers using new techniques that streamline the development process. First, an optimization algorithm transforms high-level descriptions of each maneuver into dynamically-feasible reference motions. Then Atlas tracks the motions using a **model predictive controller** that smoothly blends from one maneuver to the next. Using this approach, we developed the routine significantly faster than previous Atlas routines, with a performance success rate of about 80%.

iterative LQR (iLQR)



iterative LQR (iLQR)

- Approximate a nonlinear system as a linear-quadratic system at \tilde{x}_t, \tilde{u}_t with Taylor expansion

$$x_{t+1} = f(x_t, u_t) \approx f(\tilde{x}_t, \tilde{u}_t) + \nabla_{x_t, u_t} f(\tilde{x}_t, \tilde{u}_t) \begin{bmatrix} x_t - \tilde{x}_t \\ u_t - \tilde{u}_t \end{bmatrix}$$

$$c(x_t, u_t) \approx c(\tilde{x}_t, \tilde{u}_t) + \nabla_{x_t, u_t} c(\tilde{x}_t, \tilde{u}_t) \begin{bmatrix} x_t - \tilde{x}_t \\ u_t - \tilde{u}_t \end{bmatrix} + \frac{1}{2} \begin{bmatrix} x_t - \tilde{x}_t \\ u_t - \tilde{u}_t \end{bmatrix}^T \nabla_{x_t, u_t}^2 c(\tilde{x}_t, \tilde{u}_t) \begin{bmatrix} x_t - \tilde{x}_t \\ u_t - \tilde{u}_t \end{bmatrix}$$

$$\delta x_t = x_t - \tilde{x}_t, \quad \delta x_{t+1} = f(x_t, u_t) - f(\tilde{x}_t, \tilde{u}_t)$$

$$\delta u_t = u_t - \tilde{u}_t$$

- Run LQR with state δx_t and action δu_t . Then rerun the linearization to update the model.

Case study: nonlinear model-predictive control with iLQR

Synthesis and Stabilization of Complex Behaviors through Online Trajectory Optimization

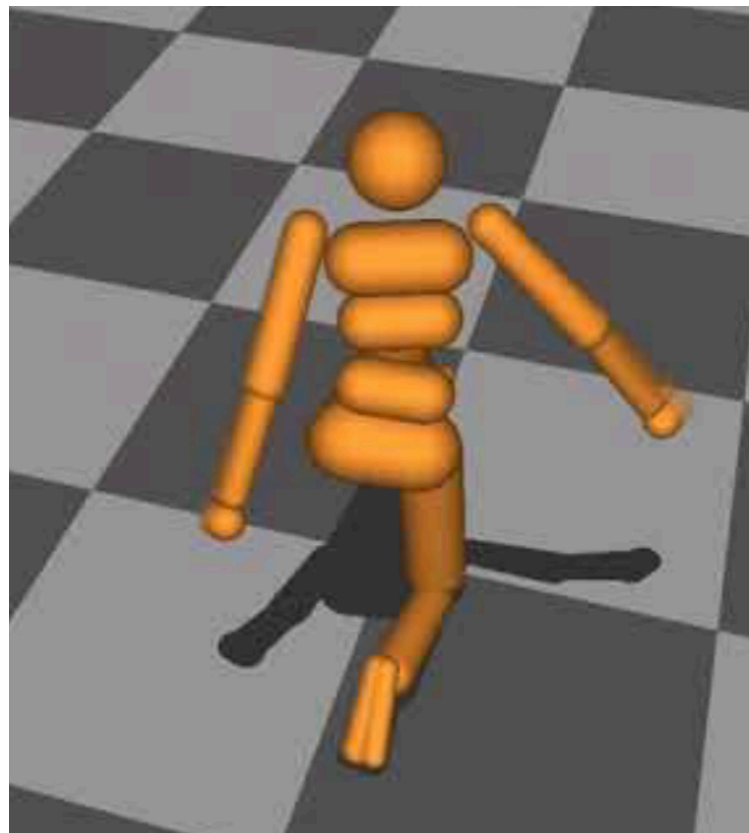
Yuval Tassa, Tom Erez and Emanuel Todorov
University of Washington

every time step:

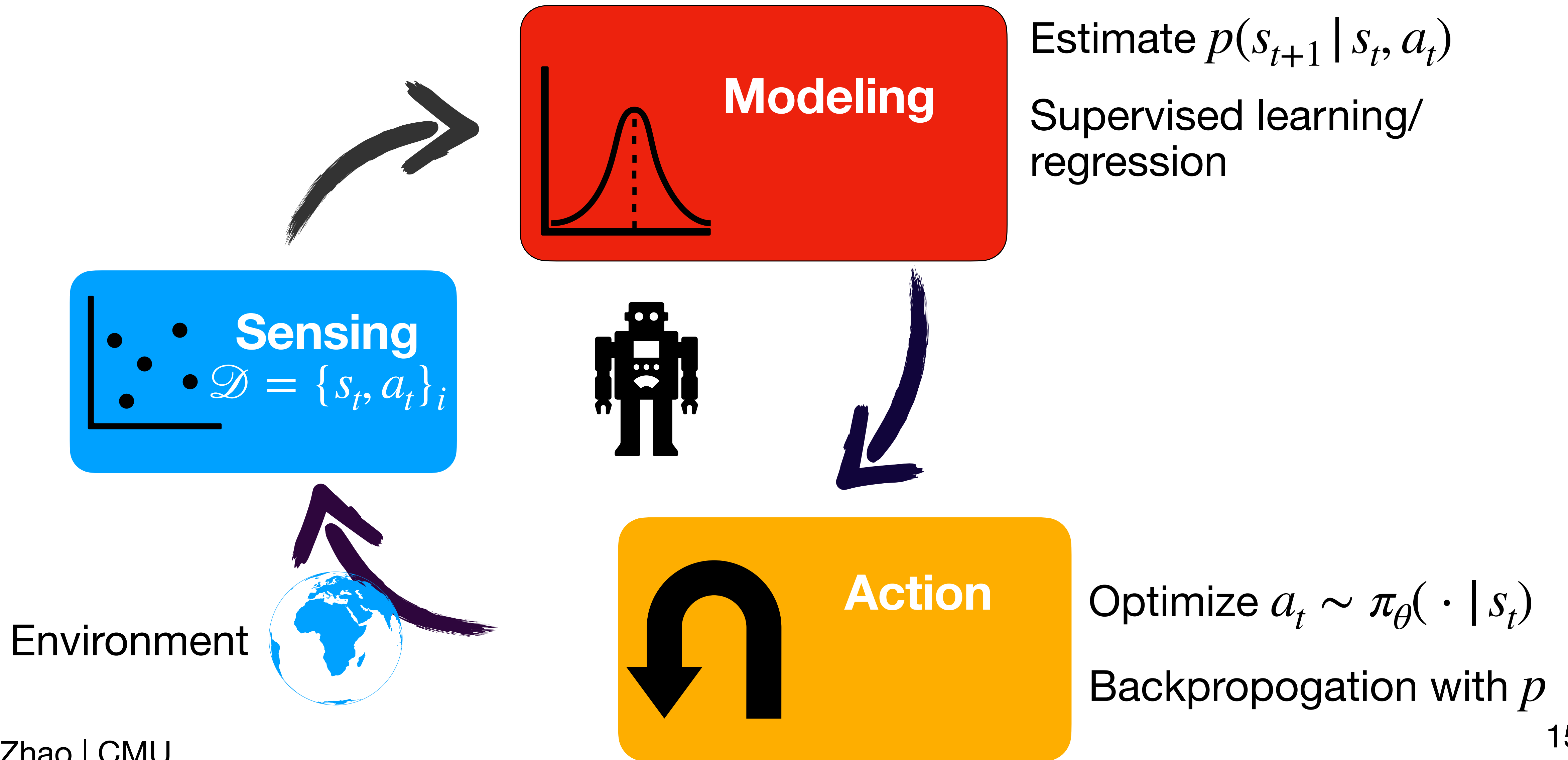
observe the state \mathbf{x}_t

use iLQR to plan $\mathbf{u}_t, \dots, \mathbf{u}_T$ to minimize $\sum_{t'=t}^{t+T} c(\mathbf{x}_{t'}, \mathbf{u}_{t'})$

execute action \mathbf{u}_t , discard $\mathbf{u}_{t+1}, \dots, \mathbf{u}_{t+T}$

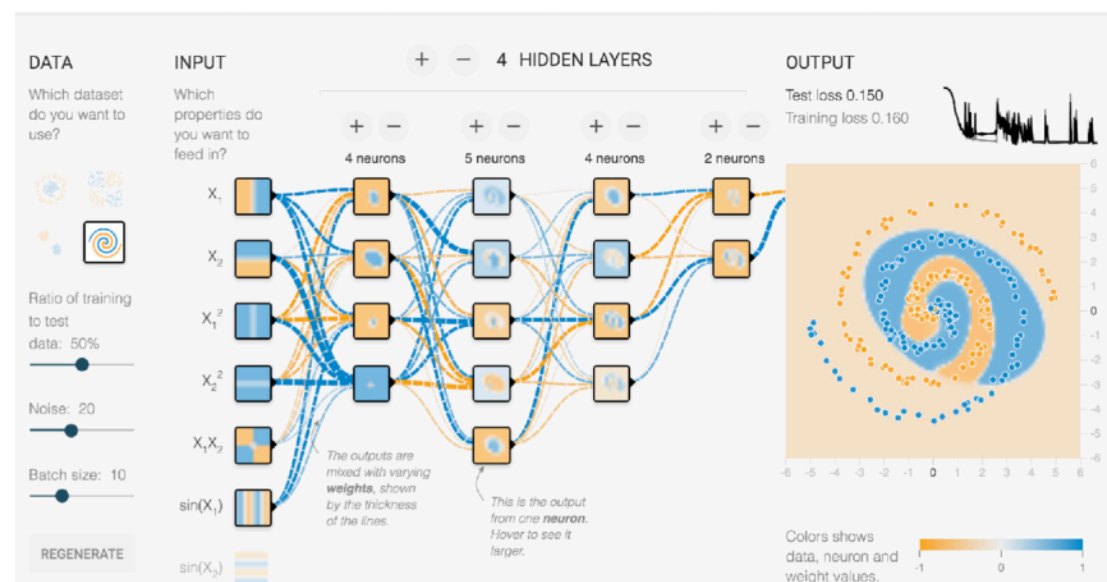


Model-based Reinforcement Learning



What kind of models can we learn?

Neural networks



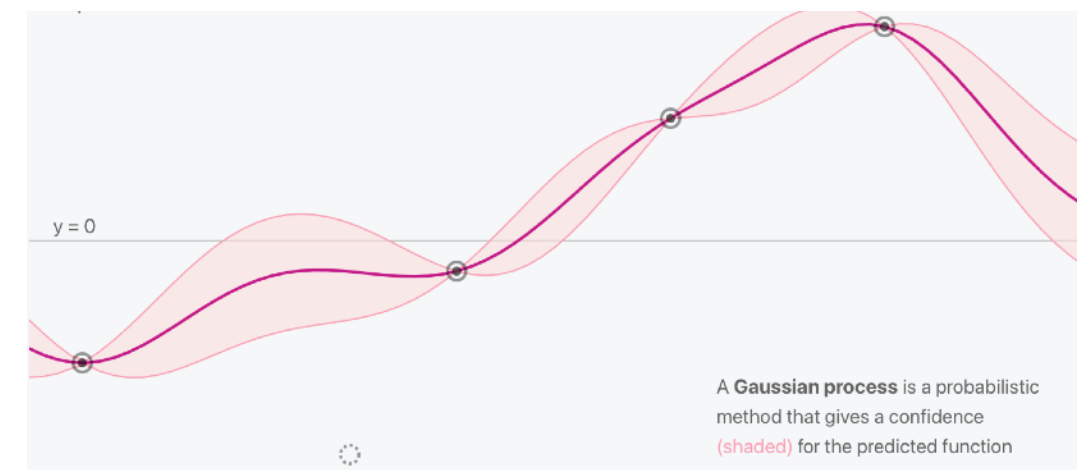
$$s_{t+1} = f_{\phi}(s_t, a_t)$$

Pro: very expressive, can take the advantage of rich data

Con: not so good in low data regimes/rare events, lack of interpretation

Parametric

Stochastic functions (Gaussian Processes)



$$s_{t+1} \sim \mathcal{N}(\cdot | s_t, a_t, \mathcal{D})$$

Pro: data efficient

Con: hard to model non-smooth dynamics, slower than NN when dataset is big

Nonparametric

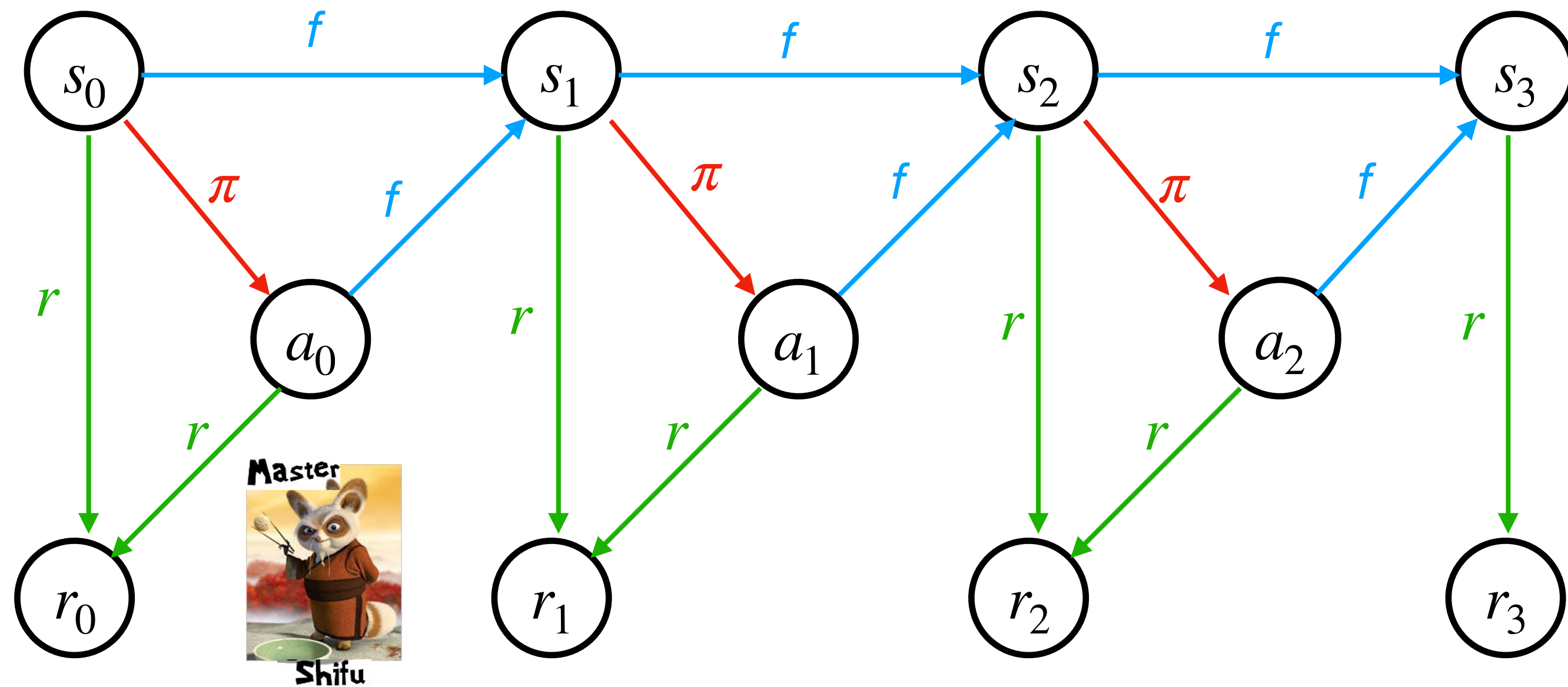
Hierarchical /modular structures



Pro: good interpretation, data efficient

Con: hard to train

Reinforcement Learning - NN model-based



MB-NN-RL-1.0

Issue: we may over-rely on the model, which could have safety issues.

Planning helps to make the model more trustworthy

1. Run base policy $\pi_{\theta^{(0)}}(a_t | s_t)$ (e.g., random policy) to collect $\mathcal{D} = \{(s_t, a_t, s_{t+1})_{t=1:D}\}_k$
2. Learn model $f_{\phi^{(i)}}$ by minimizing $\sum_t \|f_{\phi^{(i)}}(s_t, a_t) - s_{t+1}\|^2$
3. Optimize $\pi_{\theta^{(k)}}(a_t | s_t)$ using $f_{\phi^{(i)}}$ via backpropagate
4. Execute with policy $\pi_{\theta^{(k)}}(a_t | s_t)$, append new data \mathcal{D}_{k+1} to \mathcal{D}

Reinforcement Learning - NN model-based

MB-NN-RL-2.0

1. Run base policy $\pi_{\theta^{(0)}}(a_t | s_t)$ (e.g., random policy) to collect $\mathcal{D} = \{(s_t, a_t, s_{t+1})_{t=1:D}\}_k$
2. Learn model $f_{\phi^{(i)}}$ by minimizing $\sum_t \|f_{\phi^{(i)}}(s_t, a_t) - s_{t+1}\|^2$
3. Plan through $f_{\phi^{(i)}}(s_t, a_t)$ to choose actions
4. Execute the first planned action, observe results states s_{t+1}
5. Append (s_t, a_t, s_{t+1}) to \mathcal{D}

How to do planning (for multi-steps)?

- Planning with linearized models (local model)
 - i-LQR
- Planning with sampling based methods
 - CEM, PETS

Case study: local models and iLQR



Learning Contact-Rich Manipulation Skills with Guided Policy Search

Sergey Levine, Nolan Wagener, Pieter Abbeel

Abstract—Autonomous learning of object manipulation skills can enable robots to acquire rich behavioral repertoires that scale to the variety of objects found in the real world. However, current motion skill learning methods typically restrict the behavior to a compact, low-dimensional representation, limiting its expressiveness and generality. In this paper, we extend a recently developed policy search method [1] and use it to learn a range of dynamic manipulation behaviors with highly general policy representations, without using known models or example demonstrations. Our approach learns a set of trajectories for the desired motion skill by using iteratively refitted time-varying linear models, and then unifies these trajectories into a single control policy that can generalize to new situations. To enable this method to run on a real robot, we introduce several improvements that reduce the sample count and automate parameter selection. We show that our method can acquire fast, fluent behaviors after only minutes of interaction time, and can learn robust controllers for complex tasks, including putting together a toy airplane, stacking tight-fitting lego blocks, placing wooden rings onto tight-fitting pegs, inserting a shoe tree into a shoe, and screwing bottle caps onto bottles.

I. INTRODUCTION

Autonomous acquisition of manipulation skills has the potential to dramatically improve both the ease of deployment of robotic platforms, in domains ranging from manufacturing to household robotics, and the fluency and speed of the robot's motion. It is often much easier to specify *what* a robot should do, by means of a compact cost function, than

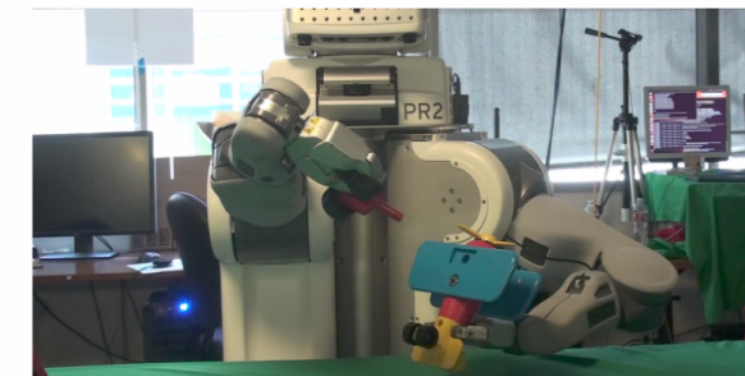


Fig. 1: PR2 learning to attach the wheels of a toy airplane.

In this paper, we show that a range of motion skills can be learned using only general-purpose policy representations. We use our recently developed policy search algorithm [1], which combines a sample-efficient method for learning linear-Gaussian controllers with the framework of guided policy search, which allows multiple linear-Gaussian controllers (trained, for example, from several initial states, or under different conditions) to be used to train a single nonlinear policy with any parameterization, including complex, high-dimensional policies represented by large neural networks. This policy can then generalize to a wider range of conditions than the individual linear-Gaussian controllers.

We present several modifications to this method that make it practical for deployment on a robotic platform.

Cross Entropy Method (Random Shooting)

Optimal planning:

$$a_1, \dots, a_T = \arg \max J(a_1, \dots, a_T), \mathbf{A} = \arg \max J(\mathbf{A})$$

Simplest method: randomly sample and pick the top actions

1. Pick $\mathbf{A}_1, \dots, \mathbf{A}_N$ from some distribution (e.g., uniform)
2. Choose \mathbf{A}_i based on $\arg \max J(\mathbf{A})$

Case study: CEM with MPC

Deep Reinforcement Learning in a Handful of Trials using Probabilistic Dynamics Models

Kurtland Chua

Roberto Calandra

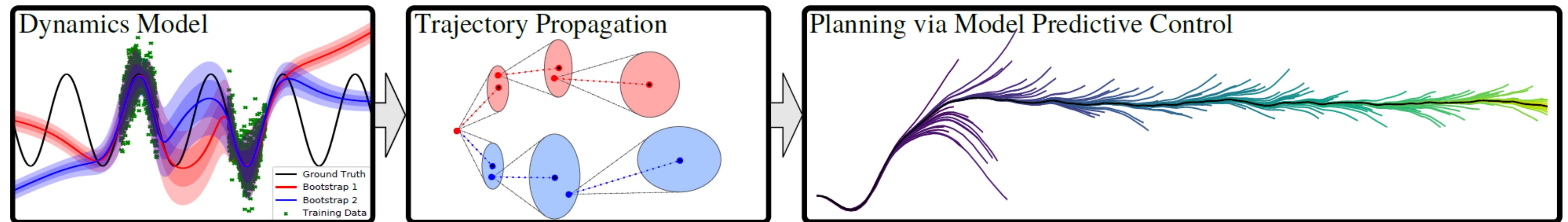
Rowan McAllister

Sergey Levine

Berkeley Artificial Intelligence Research

University of California, Berkeley

{kchua, roberto.calandra, rmcallister, svlevine}@berkeley.edu



Algorithm 1 Our model-based MPC algorithm ‘*PETS*’:

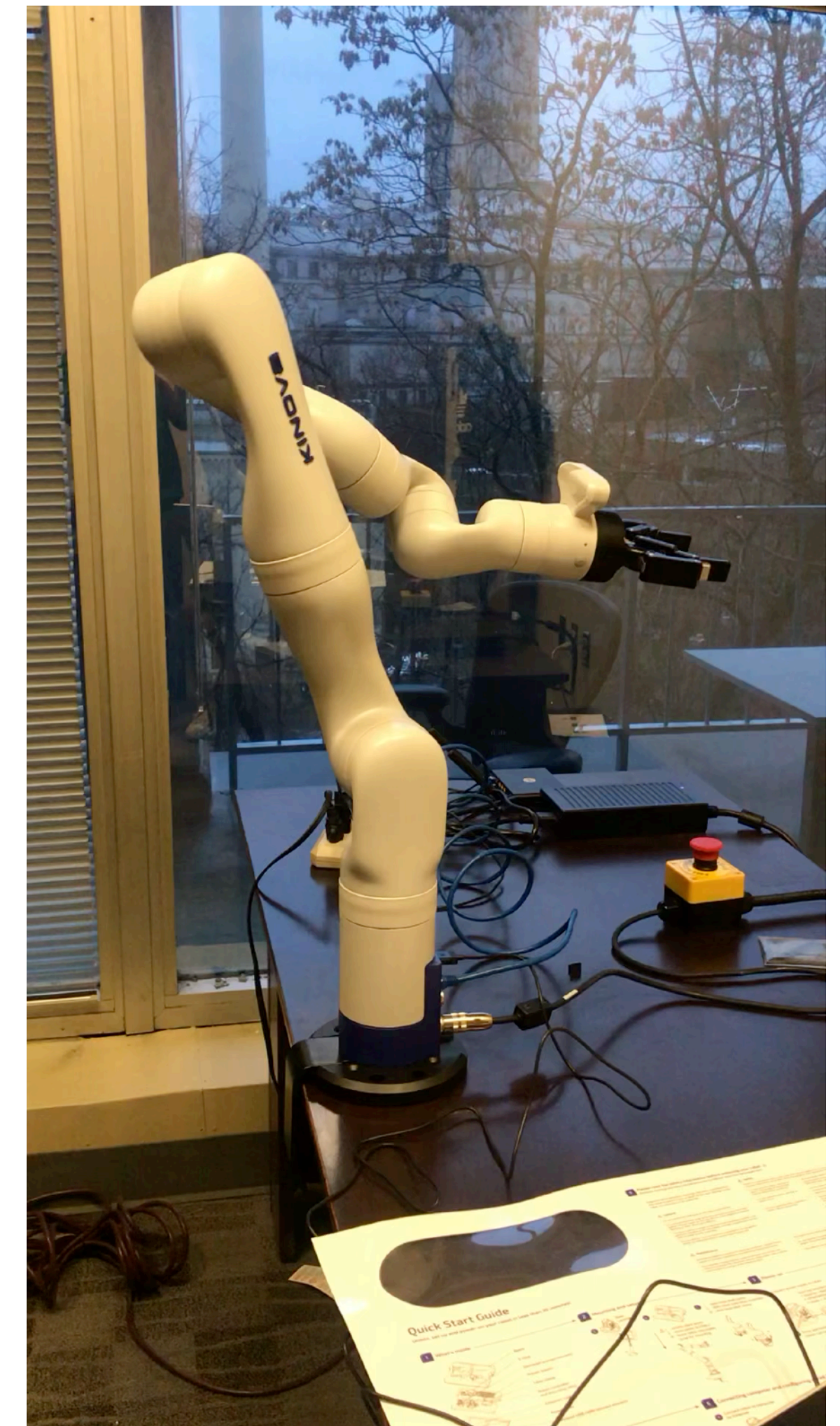
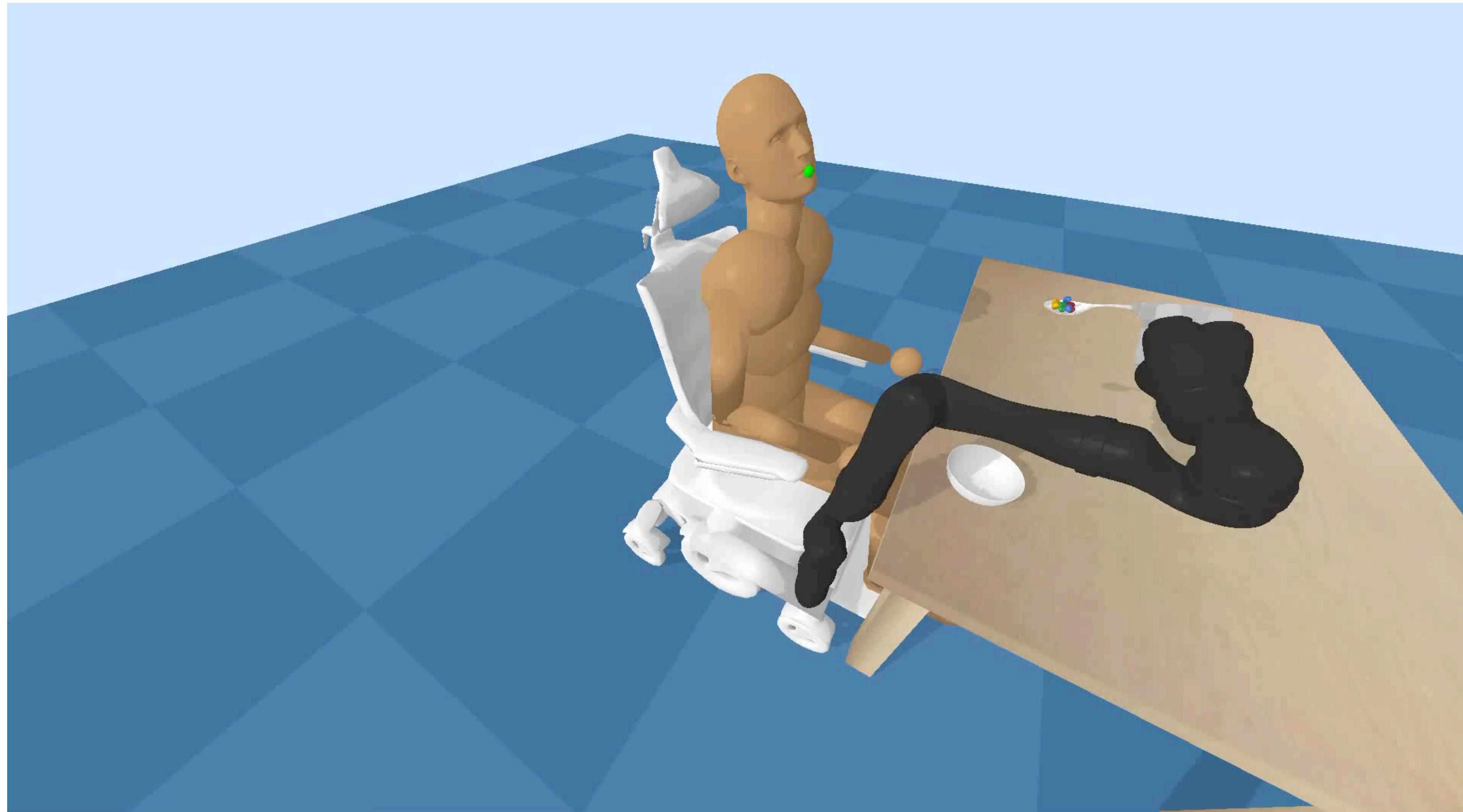
- 1: Initialize data \mathbb{D} with a random controller for one trial.
- 2: **for** Trial $k = 1$ to K **do**
- 3: Train a *PE* dynamics model \tilde{f} given \mathbb{D} .
- 4: **for** Time $t = 0$ to TaskHorizon **do**
- 5: **for** Actions sampled $\mathbf{a}_{t:t+T} \sim \text{CEM}(\cdot)$, 1 to N_{Samples} **do**
- 6: Propagate state particles \mathbf{s}_τ^p using *TS* and $\tilde{f} | \{\mathbb{D}, \mathbf{a}_{t:t+T}\}$.
- 7: Evaluate actions as $\sum_{\tau=t}^{t+T} \frac{1}{P} \sum_{p=1}^P r(\mathbf{s}_\tau^p, \mathbf{a}_\tau)$
- 8: Update $\text{CEM}(\cdot)$ distribution.
- 9: Execute first action \mathbf{a}_t^* (only) from optimal actions $\mathbf{a}_{t:t+T}^*$.
- 10: Record outcome: $\mathbb{D} \leftarrow \mathbb{D} \cup \{\mathbf{s}_t, \mathbf{a}_t^*, \mathbf{s}_{t+1}\}$.

PE: Probabilistic Ensembles

TS: Trajectory Sampling

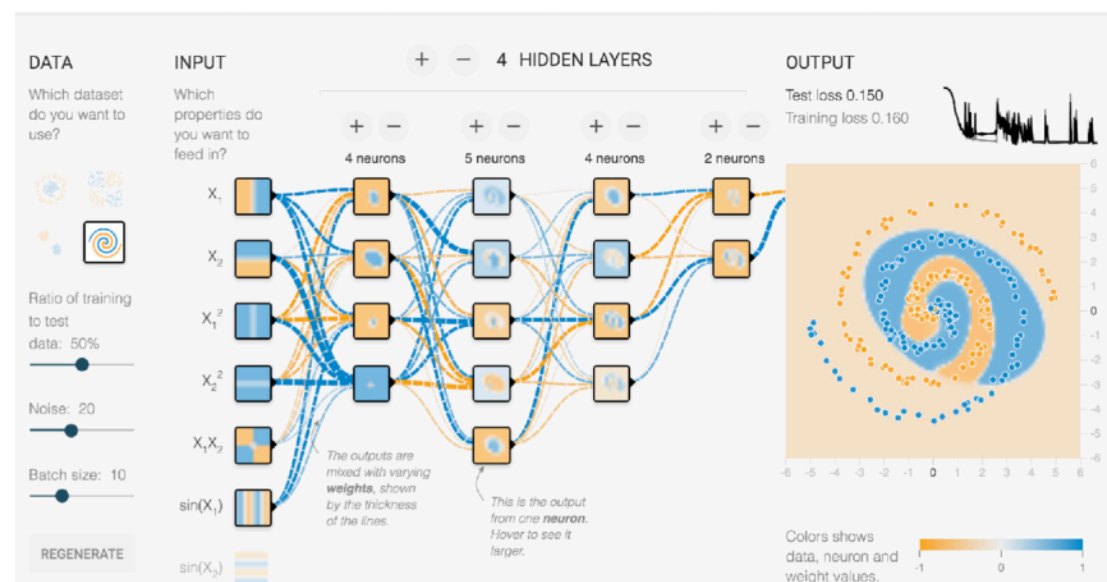
Case study: planning with CEM

Safe RL with non-stationary environment (a shaking head)



What kind of models can we learn?

Neural networks



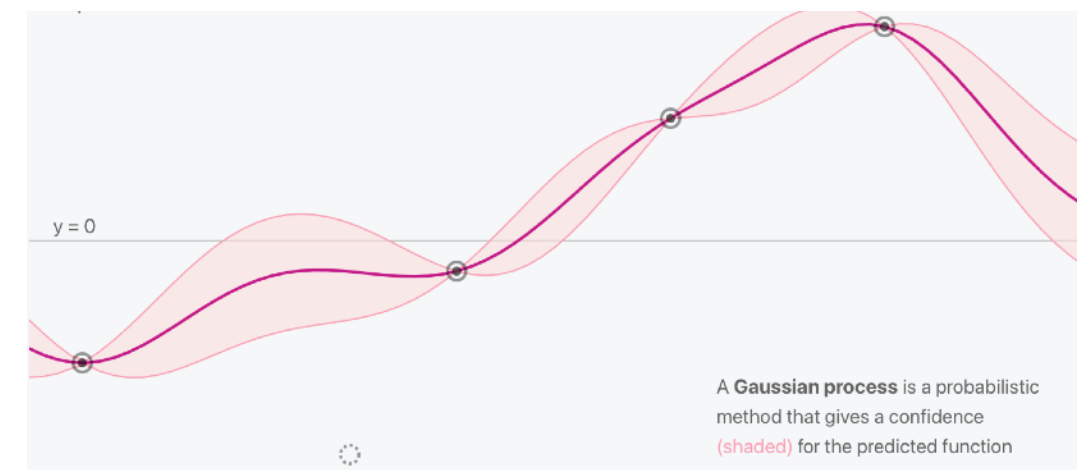
$$s_{t+1} = f_{\phi}(s_t, a_t)$$

Pro: very expressive, can take the advantage of rich data

Con: not so good in low data regimes/rare events, lack of interpretation

Parametric

Stochastic functions (Gaussian Processes)



$$s_{t+1} \sim \mathcal{N}(\cdot | s_t, a_t, \mathcal{D})$$

Pro: data efficient

Con: hard to model non-smooth dynamics, slower than NN when dataset is big

Nonparametric

Hierarchical /modular structures



Pro: good interpretation, data efficient

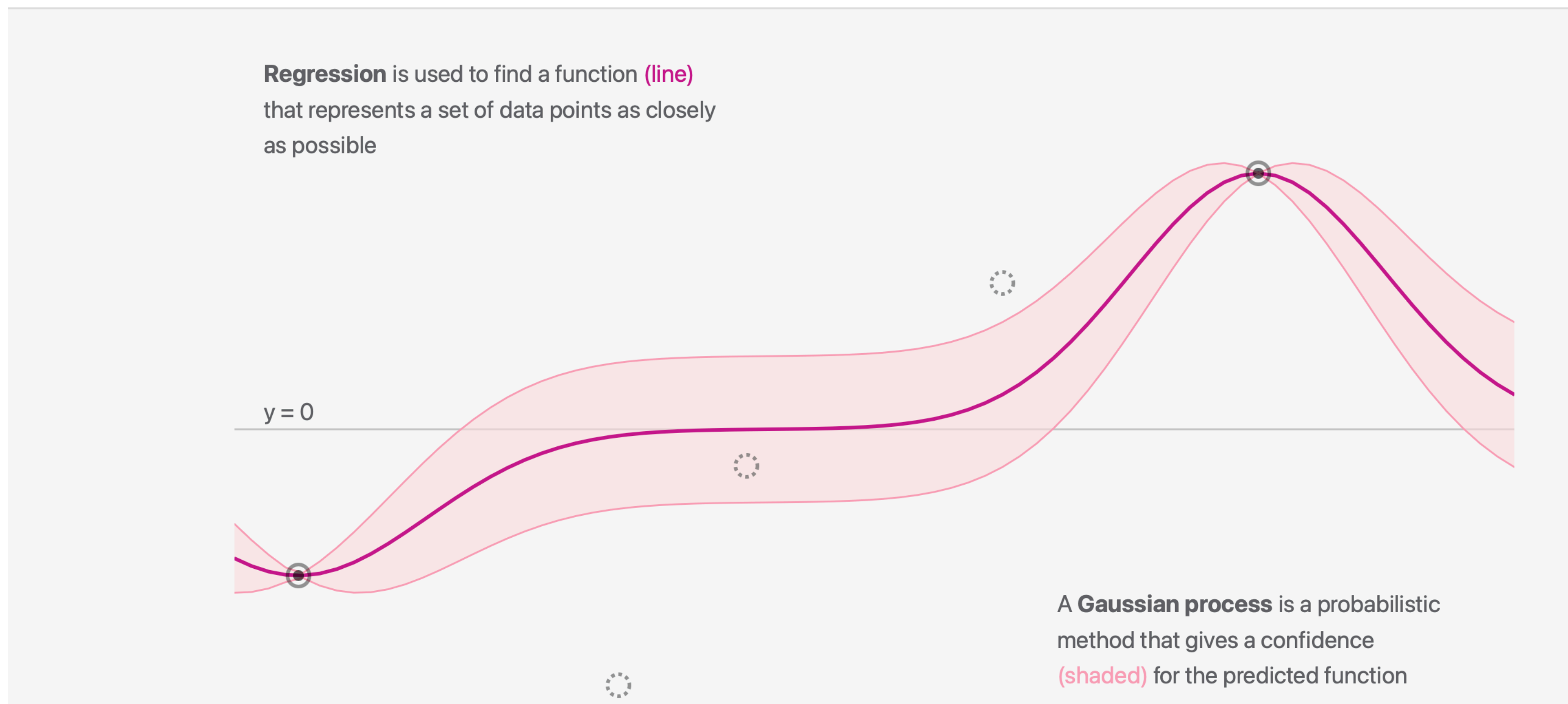
Con: hard to train

Neural network vs Gaussian processes

- Neural network
 - very powerful to approximate nonlinear functions
 - Efficient training
 - overfitting issues
- Gaussian processes
 - approximate nonlinear functions
 - provide sensible uncertainties
 - a probability distribution upon a set of functions
 - adjust complexity with data size: nonparametric
 - may suffer from the curse of dimension

A Visual Exploration of Gaussian Processes

How to turn a collection of small building blocks into a versatile tool for solving regression problems.



Effect of model errors and benefit of GP

- The main reason why model-based RL are not widely used in real-world application is that they can suffer severely from model errors, i.e., they inherently assume that the learned model resembles the real environment sufficiently accurately.
- Given a small data set of observed transitions (left), multiple transition functions plausibly could have generated them (center).

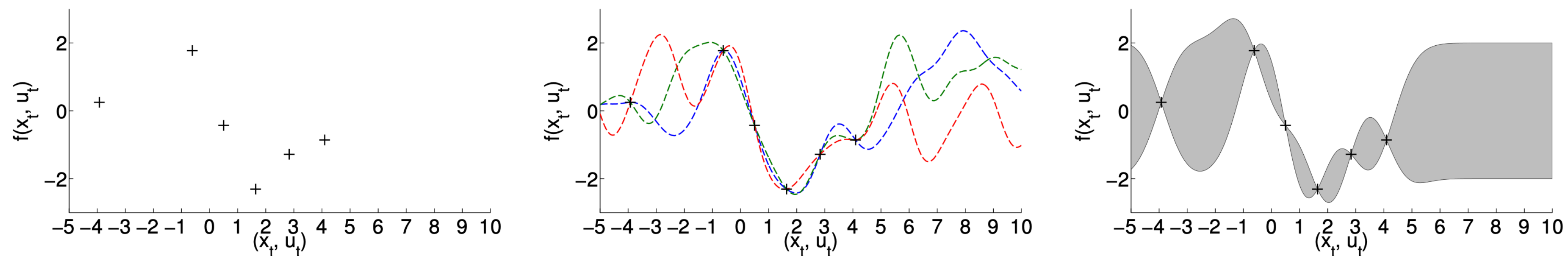
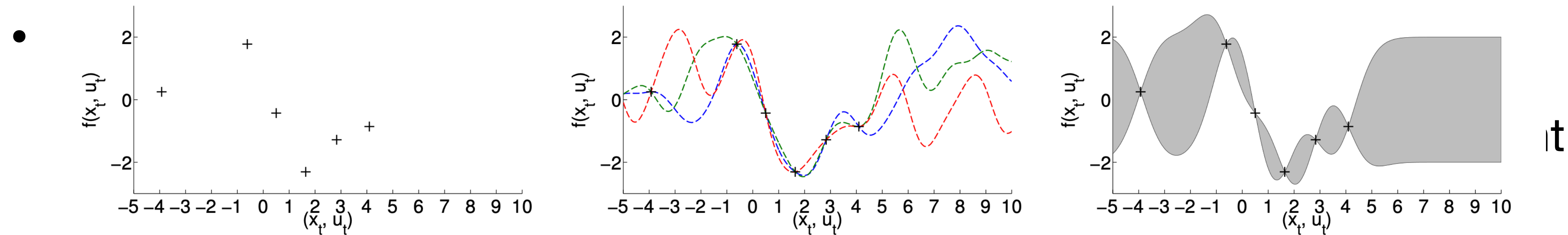


Fig. 1. Effect of model errors. Left: Small data set of observed transitions from an idealized one-dimensional representations of states and actions (x_t, u_t) to the next state $x_{t+1} = f(x_t, u_t)$. Center: Multiple plausible deterministic models. Right: Probabilistic model. The probabilistic model describes the uncertainty about the latent function by a probability distribution on the set of all plausible transition functions. Predictions with deterministic models are claimed with full confidence, while the probabilistic model expresses its predictive uncertainty by a probability distribution.

Effect of model errors and benefit of GP

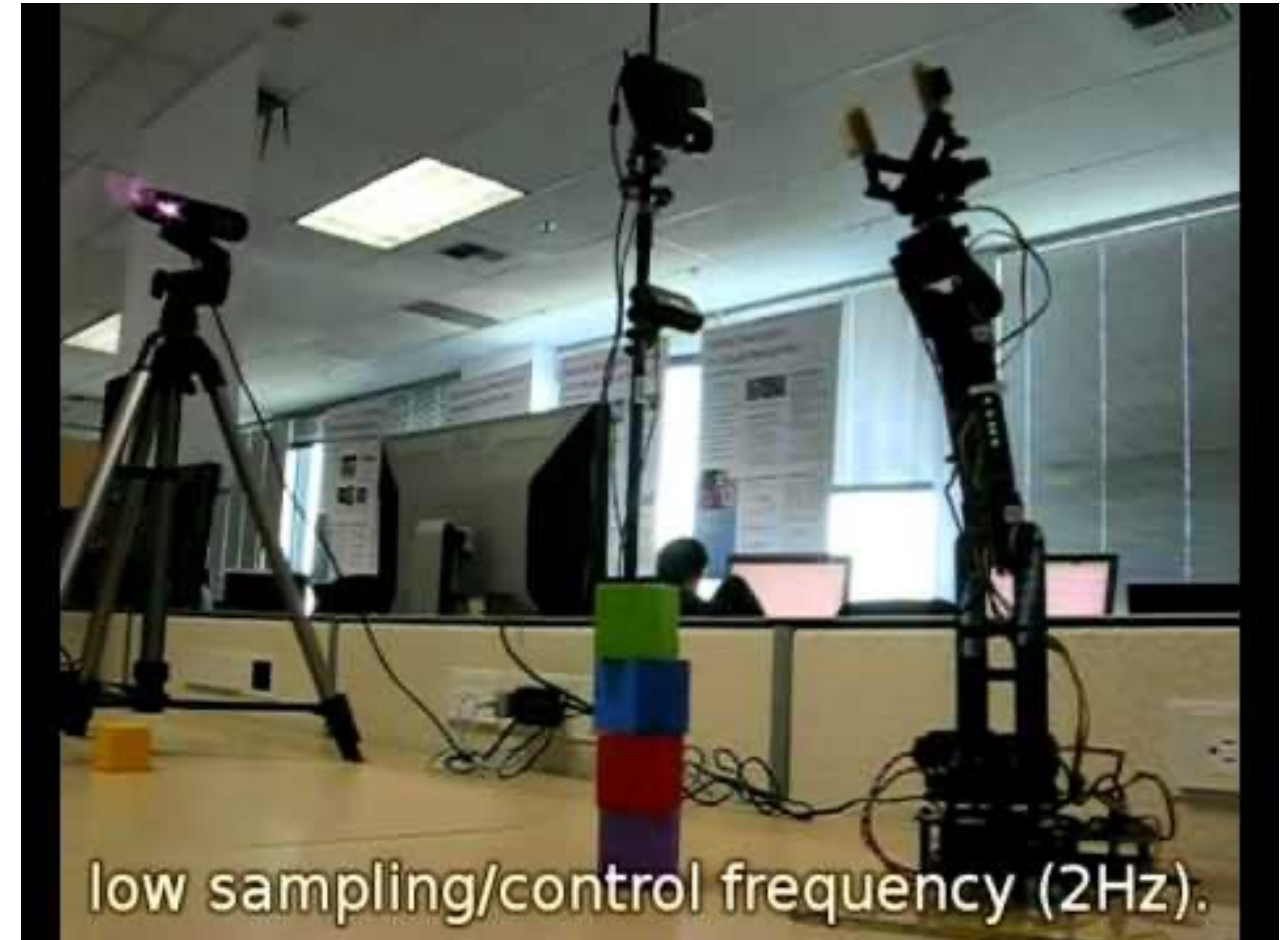


- Fig. 1. Effect of model errors. Left: Small data set of observed transitions from an idealized one-dimensional representations of states and actions (x_t, u_t) to the next state $x_{t+1} = f(x_t, u_t)$. Center: Multiple plausible deterministic models. Right: Probabilistic model. The probabilistic model describes the uncertainty about the latent function by a probability distribution on the set of all plausible transition functions. Predictions with deterministic models are claimed with full confidence, while the probabilistic model expresses its predictive uncertainty by a probability distribution.
- Choosing a single deterministic model has severe consequences: Long-term predictions often leave the range of the training data in which case the predictions become essentially arbitrary. However, the deterministic model claims them with full confidence! By contrast, a probabilistic model places a posterior distribution on plausible transition functions (right) and expresses the level of uncertainty about the model itself.

PILCO algorithm

Algorithm 1 PILCO

- 1: **init:** Sample controller parameters $\theta \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. Apply random control signals and record data.
 - 2: **repeat**
 - 3: Learn probabilistic (GP) dynamics model, see Sec. 3.1, using all data
 - 4: **repeat**
 - 5: Approximate inference for policy evaluation, see Sec. 3.2: get $J^\pi(\theta)$, Eq. (9)–(11)
 - 6: Gradient-based policy improvement, see Sec. 3.3: get $dJ^\pi(\theta)/d\theta$, Eq. (12)–(16)
 - 7: Update parameters θ (e.g., CG or L-BFGS).
 - 8: **until** convergence; **return** θ^*
 - 9: Set $\pi^* \leftarrow \pi(\theta^*)$
 - 10: Apply π^* to system and record data
 - 11: **until** task learned
-



- PILCO: Design policy to minimize the cost function $J^\pi(\theta) = \sum_{t=0}^T \mathbb{E}_{x_t}[c(x_t)]$, $\mathbf{x}_0 \sim \mathcal{N}(\boldsymbol{\mu}_0, \boldsymbol{\Sigma}_0)$ subject to the unknown system dynamics f and noise w : $\mathbf{x}_{t+1} = f(\mathbf{x}_t, \mathbf{u}_t) + \mathbf{w}$, $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{\Sigma}_w)$
- Model Learning: we use tuples $(\mathbf{x}_t, \mathbf{u}_t) \in \mathbb{R}^{D+F}$ denoted as $\tilde{\mathbf{x}} := [\mathbf{x}^\top \mathbf{u}^\top]^\top$ as training inputs and differences $\boldsymbol{\Delta}_t = \mathbf{x}_{t+1} - \mathbf{x}_t \in \mathbb{R}^D$ as training outputs (targets). The posterior GP is a one-step prediction model, and the predicted successor state \mathbf{x}_{t+1} is Gaussian distribute

$$p(\mathbf{x}_{t+1} | \mathbf{x}_t, \mathbf{u}_t) = \mathcal{N}(\mathbf{x}_{t+1} | \boldsymbol{\mu}_{t+1}, \boldsymbol{\Sigma}_{t+1}), \boldsymbol{\mu}_{t+1} = \mathbf{x}_t + \mathbb{E}_f[\boldsymbol{\Delta}_t], \boldsymbol{\Sigma}_{t+1} = \text{var}_f[\boldsymbol{\Delta}_t]$$

where the mean and variance of the GP prediction are

$$\mathbb{E}_f[\boldsymbol{\Delta}_t] = m_f(\tilde{\mathbf{x}}_t) = \mathbf{k}_*^\top (\mathbf{K} + \sigma_w^2 \mathbf{I})^{-1} \mathbf{y}, \quad \text{var}_f[\boldsymbol{\Delta}_t] = k_{**} - \mathbf{k}_*^\top (\mathbf{K} + \sigma_w^2 \mathbf{I})^{-1} \mathbf{k}_*$$

respectively, with $\mathbf{k}_* := k(\tilde{\mathbf{X}}, \tilde{\mathbf{x}}_t)$, $k_{**} := k(\tilde{\mathbf{x}}_t, \tilde{\mathbf{x}}_t)$, $\tilde{\mathbf{X}} = [\tilde{\mathbf{x}}_1, \dots, \tilde{\mathbf{x}}_n]$, $\mathbf{y} = [\boldsymbol{\Delta}_1, \dots, \boldsymbol{\Delta}_n]^\top$, \mathbf{K} is the kernel matrix with entries $K_{ij} = k(\tilde{\mathbf{x}}_i, \tilde{\mathbf{x}}_j)$.

- Kernel: a positive semidefinite covariance function

$$k(\tilde{\mathbf{x}}_p, \tilde{\mathbf{x}}_q) = \sigma_f^2 \exp(-\frac{1}{2}(\tilde{\mathbf{x}}_p - \tilde{\mathbf{x}}_q)^\top \boldsymbol{\Lambda}^{-1}(\tilde{\mathbf{x}}_p - \tilde{\mathbf{x}}_q)) + \delta_{pq} \sigma_w^2$$

With parameters length-scales ℓ_i , signal variance σ_f^2 , and noise variance σ_w^2 learned by max likelihood

GP-based RL in a real-world application



Challenges in model learning

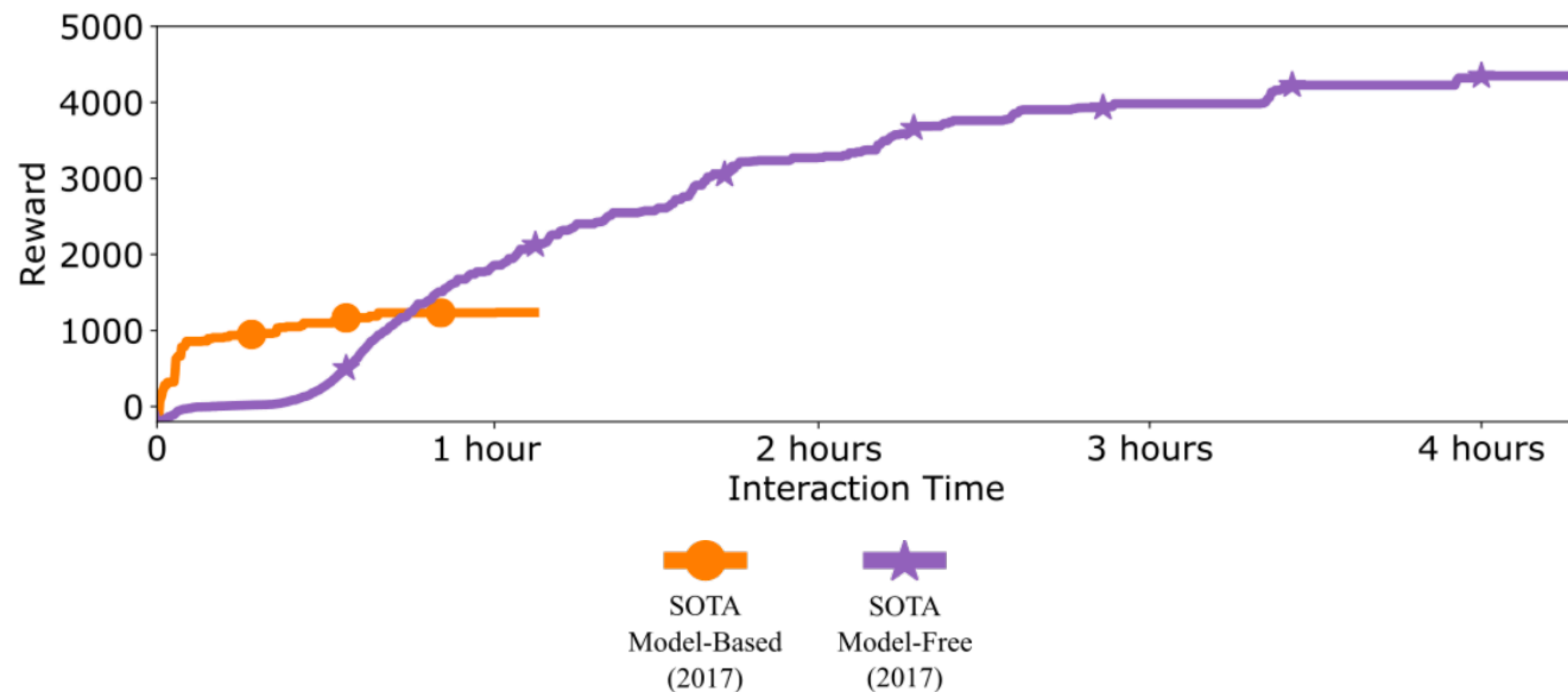
- Under-fitting: If the model class is restricted (e.g., linear function or gaussian process) we have under-modeling: we cannot represent complex dynamics, e.g., contact dynamics that are not smooth. As a result, though we learn faster than model free in the beginning, MBRL ends up having worse asymptotic performance than model-free methods, that do not suffer from model bias.
- Over-fitting: If the model class is very expressive (e.g., neural networks) the model will overfit, especially in the beginning of training, where we have very few samples
- Uncertainty/errors propagated and amplified through planning

Model-based vs Model free

- Model-based
 - + data efficient in training
 - + Possibility to transfer across tasks
 - + Increase interoperability
 - Do not optimize directly over performance
 - Usually need domain knowledge (overfitting/under-fitting)
 - Maybe hard to learn policy
- Model-free
 - + Need little assumption
 - + Efficient for learning complex policy
 - Require a lot training data
 - Not transferable and lack of interoperability

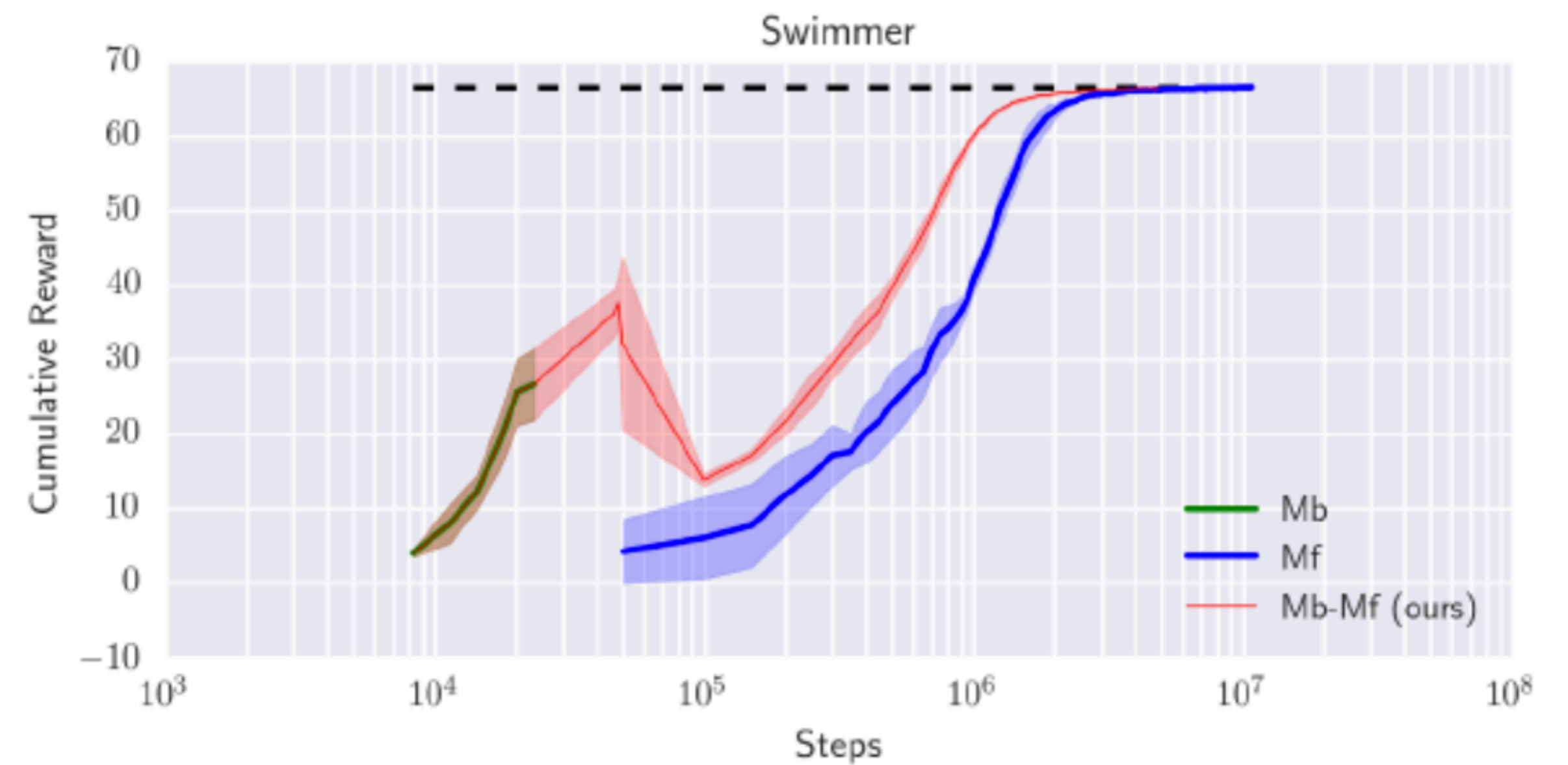
Widely used in safety-critical applications

Combine model-based and model-free

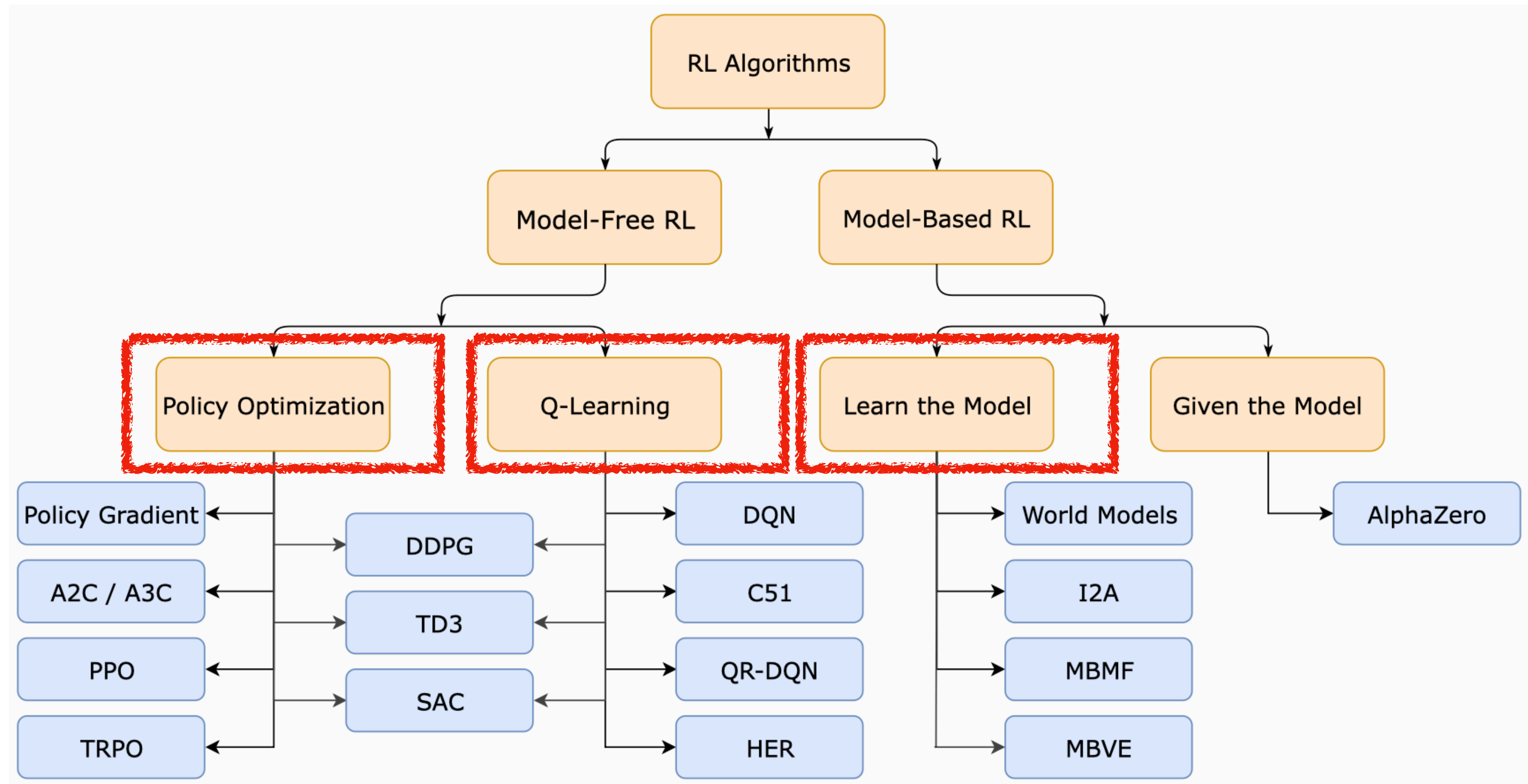


Neural Network Dynamics for Model-Based Deep Reinforcement Learning with Model-Free Fine-Tuning

Anusha Nagabandi, Gregory Kahn, Ronald S. Fearing, Sergey Levine
University of California, Berkeley

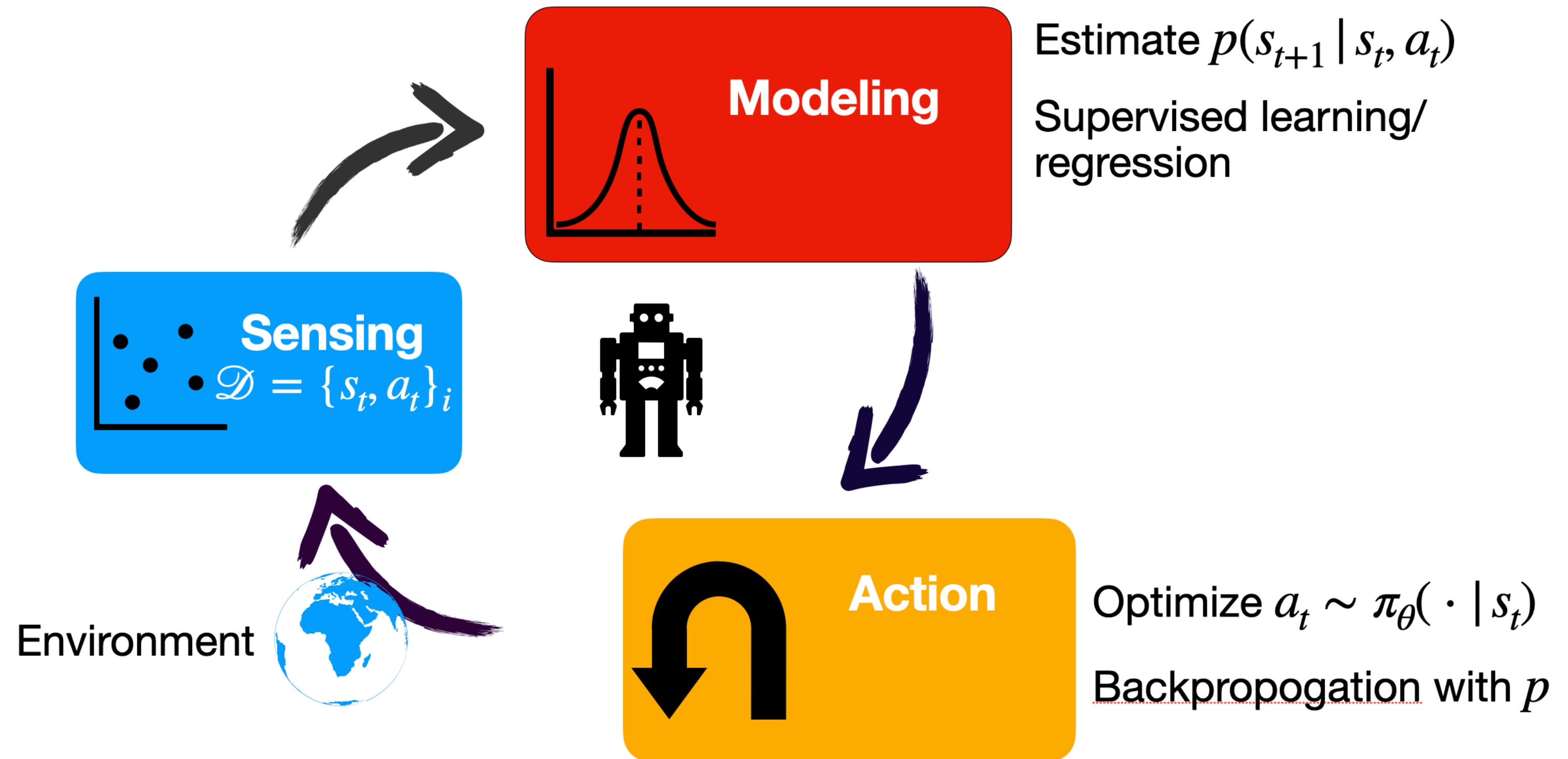


Popular RL algorithms



Summary

- Model-based RL
 - Choose the format of the models
 - Learn $p(s_{t+1} | s_t, a_t)$ with new data and pre-defined models
 - Replanning at each step (iLQR or CEM)



Worth reading

- Levine, Abbeel. (2014). Learning Neural Network Policies with Guided Policy Search under Unknown Dynamics.
- a github repo <https://github.com/anassinator/ilqr>
- A Visual Exploration of Gaussian Processes
 - <https://distill.pub/2019/visual-exploration-gaussian-processes/>
- M. P. Deisenroth, D. Fox and C. E. Rasmussen, "Gaussian Processes for Data-Efficient Learning in Robotics and Control," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 37, no. 2, pp. 408-423, Feb. 2015.