Reinforcement Learning

CS 59300: RL1

September 25, 2025

Joseph Campbell

Department of Computer Science

PURDUE UNIVERSITY®

Today's lecture

1. Proximal policy optimization

2. Off-policy actor-critic

3. Model-based reinforcement learning

Some content inspired by Katerina Fragkiadaki's CMU 10-403

Project proposals

Discuss your project idea with me!

Due this Friday

Send a confirmation email, even if you have already talked with me

Recap: The effect of learning rate

In actor critic, we update model weights with

$$\theta = \theta + \alpha\gamma^t A(s_t, a_t)\nabla \ln \pi(a_t|s_t)$$

The learning rate α dictates our step size

How far should we step?

Reinforcement learning vs imitation learning

In supervised learning, we tune step size to fit “labels” on a dataset

In (on-policy) RL, the step size *changes the dataset we see next*

- Step size is too big: bad policy update which means we collect bad data for next gradient update
- Step size is too small: we don’t update policy enough, which means we collect very similar data each time

Recap: KL-constrained optimization

We want to improve our expected policy returns while subject to the constraint that we don't change the policy *too much* at once

- Constraint dictated by KL divergence

$$\max_{\Delta\theta} g^T \Delta\theta \quad \text{s.t.} \quad D_{\mathrm{KL}}\left(\pi_\theta || \pi_{\theta+\Delta\theta}\right) \leq \epsilon \quad \text{where } g = \nabla\theta J(\theta)$$

Maximize gains while ensuring KL $\leq \epsilon$

Recap: Natural Policy Gradient

$$\theta_{\text{new}} = \theta_{\text{old}} + \Delta\theta^*$$

Recalling back to our Taylor expansion:

$$D_{\text{KL}}(\pi_\theta || \pi_{\theta+\Delta\theta}) \approx \frac{1}{2}\Delta\theta^T F(\theta)\Delta\theta = \frac{1}{2}(\alpha\Delta\theta)^T F(\alpha\Delta\theta)$$

$$\theta_{\text{new}} = \theta_{\text{old}} + \sqrt{\frac{2\epsilon}{\Delta\theta F^{-1}\Delta\theta}} F^{-1}(\theta)\nabla\theta J(\theta)$$

Recap: Quadratic approximations are *local*

Since the quadratic is only local, in practice the actual empirical KL update may be larger than ϵ

Simple solution: perform a backtracking line search

1. Compute progressively smaller proposed steps
2. If step yields positive advantage and KL $\leq \epsilon$ then repeat

Recap: Trust region policy optimization

Algorithm 2 Line Search for TRPO

Compute proposed policy step $\Delta_k = \sqrt{\frac{2\epsilon}{\hat{g}_k^T \hat{F}_k^{-1} \hat{g}_k}} \hat{F}_k^{-1} \hat{g}_k$

for $j = 0, 1, 2, \ldots, L$ **do**

  Compute proposed update $\theta = \theta_k + \alpha^j \Delta_k$

  if $\bar{\mathbb{A}}_{\pi_{old}}(\pi) \geq 0$ and $\bar{D}_{KL}(\theta || \theta_k) \leq \delta$ **then**

    accept the update and set $\theta_{k+1} = \theta_k + \alpha^j \Delta_k$

    break

  end if

end for

Useful link:
https://spinningup.openai.com/en/latest/algorithms/trpo.html

Schulman et al, Trust Region Policy Optimization, 2015.

Proximal policy optimization

TRPO is not very practical

It requires performing second-order optimization with Fisher info

- Very computationally expensive!
- Even with conjugate gradient optimization

Difficult to implement

- Complex computations can result in subtle errors
- Algorithm is very brittle if not implemented correctly

Proximal policy optimization: practical TRPO

PPO avoids constrained optimization by using simple heuristics

- Goal is to approximate the trust region efficiently

How? Two options:

- Clip the surrogate objective function
- Use a first-order KL penalty rather than a constraint

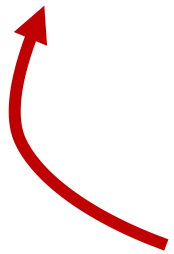
Result: effective and easy to implement

- A good “first try” baseline
- More sample efficient than TRPO (multiple gradient updates)

Clipped objective

As in TRPO, optimize the surrogate objective

$$\mathcal{L}(\theta) = \mathbb{E}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)} A(s,a)\right] = \mathbb{E}[r(\theta)A(s,a)]$$



Importance ratio.
Needed because we are estimating the advantage for each candidate θ using "old" data.

Goal is to maximize advantage under new candidate policy.

Clipped objective

As in TRPO, optimize the surrogate objective

$$\mathcal{L}(\theta) = \mathbb{E}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)} A(s,a)\right] = \mathbb{E}[r(\theta)A(s,a)]$$

But rather than explicitly computing candidates and checking constraints, form a lower bound by clipping the importance ratio

$$\mathcal{L}^{CLIP}(\theta) = \mathbb{E}[\min(r(\theta)A(s,a), \text{clip}(r(\theta), 1-\epsilon, 1+\epsilon)A(s,a)]$$

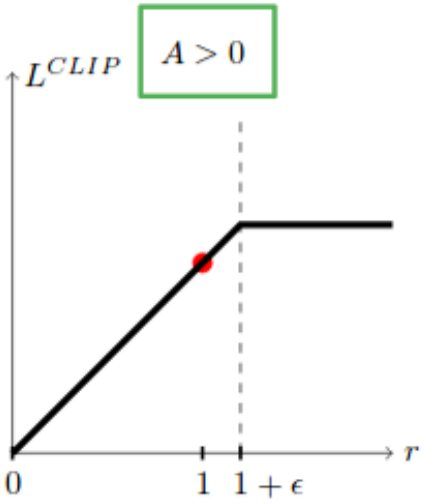
Clipped objective

$$\mathcal{L}^{CLIP}(\theta) = \mathbb{E}[\min(r(\theta)A(s,a), \text{clip}(r(\theta), 1-\epsilon, 1+\epsilon)A(s,a)]$$

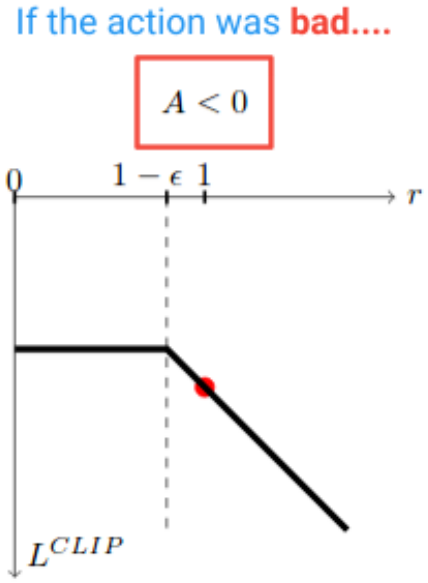
TRPO unclipped objective

Clipped objective

If the action was **good**....



If the action was **bad**....



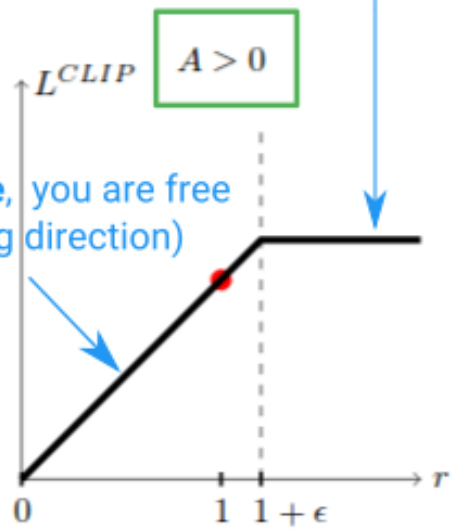
$$clip(f(x), a, b) = \begin{cases} f(x) & a \le f(x) \le b \\ a & f(x) \le a \\ b & f(x) \ge b \end{cases}$$

Clipped objective

If the action was **good**....

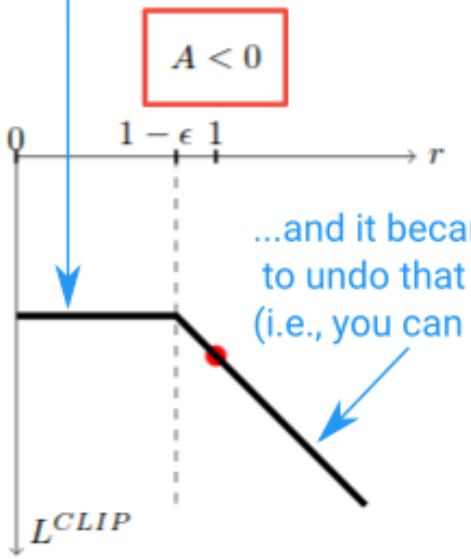
...and it became **more probable** the last time you took a gradient step, don't keep updating it too far or else the policy might get worse

...and it became **less probable**, you are free to undo that step (in the wrong direction) as much as you want



If the action was **bad**....

...and it became **less probable**, don't keep making it too much less probable or else the policy might get worse (i.e., don't step too far)



...and it became **more probable**, you are free to undo that step as much as you want (i.e., you can fix your mistakes)

https://stackoverflow.com/questions/46422845/what-is-the-way-to-understand-proximal-policy-optimization-algorithm-in-rl/50663200#50663200

Clipped objective

What is the purpose of the minimum term?

Recall that:

- If $A > 0$, the action is “better” than what we would currently take
- If $A < 0$, the action is “worse” than what we would currently take

Intuition: the minimum means the unclipped objective is taken when we increase the probability of taking a bad action. It is a penalty.

Adaptive KL penalty

$$\mathcal{L}^{KLPEN}(\theta) = \mathbb{E}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)} A(s,a) - \beta \mathrm{KL}[\pi_{\theta_{old}}(a|s)||\pi_\theta(a|s)]\right]$$

At each iteration, compute $d = \mathrm{KL}[\pi_{\theta_{old}}(a|s)||\pi_\theta(a|s)]$

- If $d < d_{targ}/1.5$ then $\beta = \frac{\beta}{2}$
- If $d > d_{targ} * 1.5$ then $\beta = \beta * 2$

Adapt penalty to approximately enforce d_{targ} KL divergence constraint

Algorithm 1 PPO, Actor-Critic Style

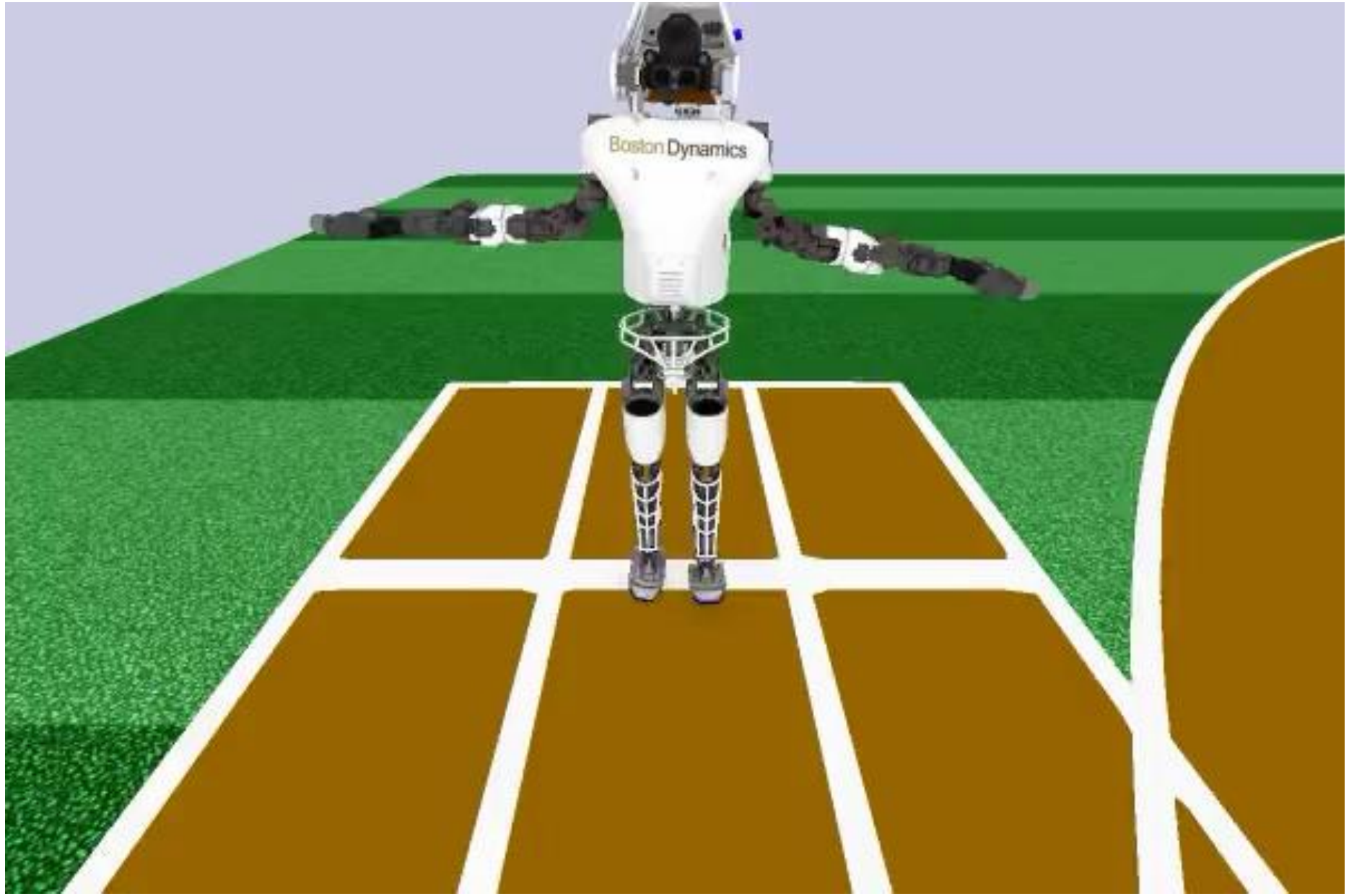
for iteration=$1, 2, \ldots$ **do**
  for actor=$1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t\left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)\right],$$

Clipped objective

Value function loss

Entropy bonus



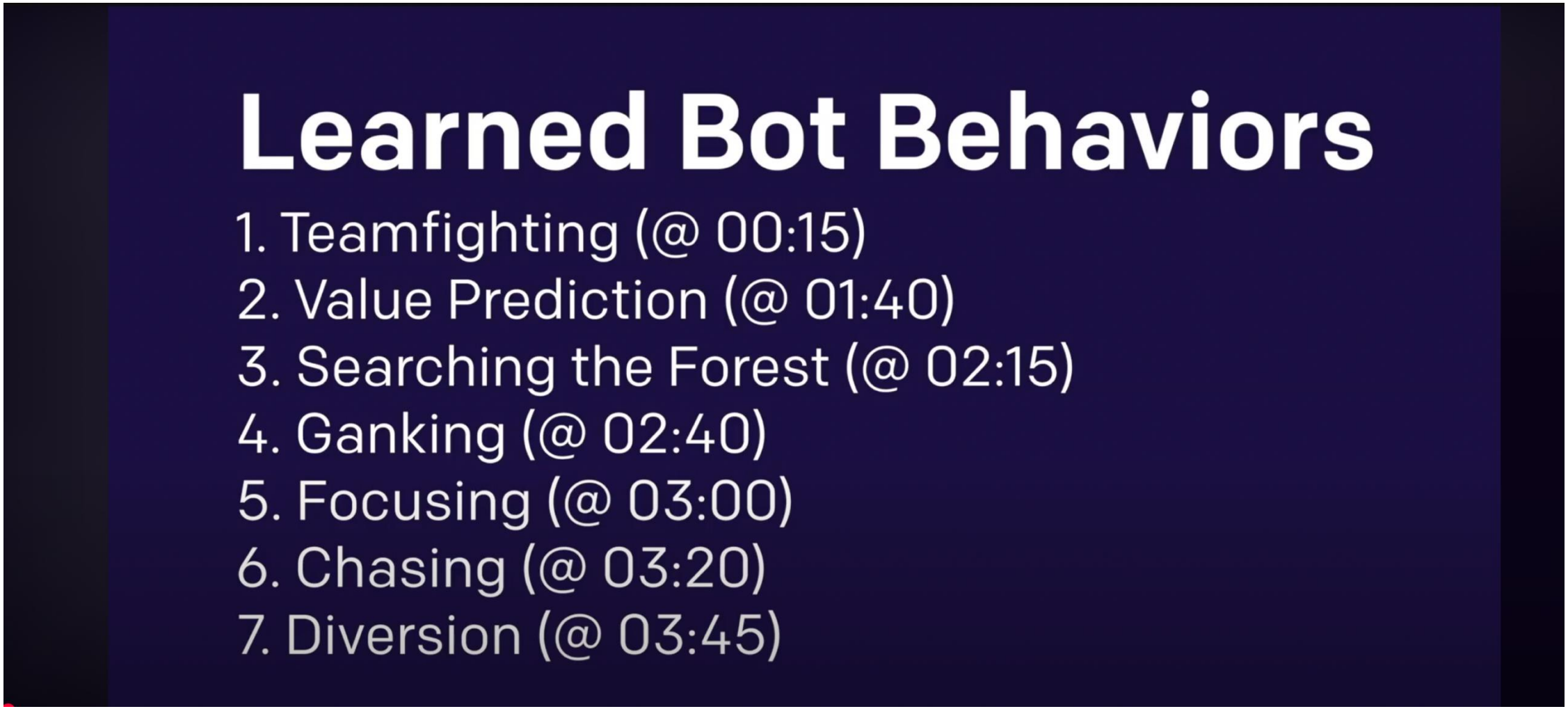
https://openai.com/index/openai-baselines-ppo/

https://openai.com/index/openai-baselines-ppo/

April 15, 2019 Milestone

OpenAI Five defeats Dota 2 world champions

Berner et al, Dota 2 with Large Scale Deep Reinforcement Learning, 2019.



https://www.youtube.com/watch?v=UZHTNBMAfAA

Off-policy actor-critic

Recall: What is “on-policy” learning?

On-policy learning:

- “Learn on the job”
- The policy learns from its own experience
- Improve policy π from episodes sampled from π

Off-policy learning:

- “Look over someone’s shoulder”
- The policy learns from another policy’s experience
- Improve policy π from episodes sampled from β

Recall: Off-policy learning

Goal: evaluate target policy $\pi(a|s)$ to compute $V_\pi(s)$ or $Q_\pi(s,a)$ while following behavior policy $\mu(a|s)$

- This means exploration is handled by μ!

$$s_1, a_1, r_2, \ldots s_t \sim \mu$$

Benefits:

- Learn from observing humans or other agents
- Re-use experience from old policies
- Learn optimal policy while following sub-optimal exploratory policy

NPG, TRPO, and PPO are on-policy

Can we come up with an *off-policy* actor-critic algorithm?

One possible solution: DQN + learned policy

- Benefits of actor critic
- Sample efficiency of off-policy learning

This is known as deep deterministic policy gradient

Deep deterministic policy gradient

As in DQN, we seek to learn state-action values

$$Q(s,a) = Q(s,a) + \alpha\left(r_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q(s_{t+1}, a') - Q(s,a)\right)$$

Lillicrap et al, Continuous Control with Deep Reinforcement Learning, 2016.

Deep deterministic policy gradient

As in DQN, we seek to learn state-action values

$$Q(s,a) = Q(s,a) + \alpha\left(r_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q(s_{t+1}, a') - Q(s,a)\right)$$

Step size

Magnitude of error

Direction of error

$$\Delta \mathbf{w} = \alpha\left(r_{t+1} + \gamma \max_{a' \in \mathcal{A}} \hat{Q}\left(s_{t+1}, a', \mathbf{w}\right) - \hat{Q}(s,a,\mathbf{w})\right) \nabla_{\mathbf{w}} \hat{Q}(s,a,\mathbf{w})$$

Lillicrap et al, Continuous Control with Deep Reinforcement Learning, 2016.

Adding a learned policy

We want to update policy parameters such that

$$\theta = \theta + \alpha\gamma^t Q(s_t, a_t)\nabla \ln \pi(a_t|s_t)$$

Which gives the objective function:

$$J(\theta) = \mathbb{E}_{s\sim\mathcal{D}}[Q\big(s, \pi(s)\big)]$$

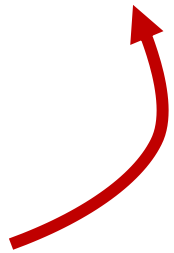
Adding a learned policy

We want to update policy parameters such that

$$\theta = \theta + \alpha\gamma^t Q(s_t, a_t)\nabla \ln \pi(a_t|s_t)$$

Which gives the objective function:

$$J(\theta) = \mathbb{E}_{s\sim\mathcal{D}}[Q(s, \pi(s))]$$



This is a deterministic policy!
Convention sometimes denotes as $\mu(s)$

Why use a deterministic policy?

The Bellman expectation tells us that…

$$Q_\pi(s,a) = \mathbb{E}[r_{t+1} + \gamma Q_\pi(s_{t+1}, a_{t+1}) | s_t = s, a_t = a]$$

This is also an expectation over actions!

$\mathbb{E}_{a_{t+1} \sim \pi} Q_\pi(s_{t+1}, a_{t+1})$

Why use a deterministic policy?

The Bellman expectation tells us that…

$$Q_\pi(s,a) = \mathbb{E}[r_{t+1} + \gamma Q_\pi(s_{t+1}, a_{t+1}) | s_t = s, a_t = a]$$

If the policy is deterministic, we remove the inner expectation

Computationally cheaper and more stable for continuous actions

- Otherwise need to sample actions from a distribution
- Or assume a particular distribution with reparameterization tricks

Optimization pseudocode

```
# s: (B, obs_dim) from replay
a = actor(s)                    # μθ(s)
q = critic(s, a)                # Qϕ(s, μθ(s))
actor_loss = -q.mean()          # maximize Q ⇒ minimize
negative
actor_optim.zero_grad()
actor_loss.backward()           # backprop through critic
into actor
actor_optim.step()
```

Optimization pseudocode

```
# s: (B, obs_dim) from replay
a = actor(s)                    # μθ(s)
q = critic(s, a)                # Qϕ(s, μθ(s))
actor_loss = -q.mean()          # maximize Q ⇒ minimize
negative
actor_optim.zero_grad()
actor_loss.backward()           # backprop through critic
into actor
actor_optim.step()
```

We need to backprop through the critic because the policy is deterministic.

The policy does not know how it should change to maximize Q.

Exploration

If our policy is deterministic, how do we explore?

Simply add noise when sampling actions:

$$\pi(s)' = \pi(s) + \epsilon$$

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

https://youtu.be/pOFli1Zlk4k?si=O-bpCq-byQURhv_7

Model-based reinforcement learning

Learning models from experience

Policy iteration and value iteration assume we know both…

- The transition function (aka environment dynamics)
- The reward function

Monte Carlo and TD learning assume we know neither

- This remains true even in today's lecture!

Today we focus on learning transition/reward models from experience

Model-based reinforcement learning

Given state, action, reward, state sequences…

- Predict the next state given the current state + action
- Predict the reward given the current state + action

Great! So now that we have learned models of the transition and reward functions, can we use policy/value iteration?

Model-based reinforcement learning

Given state, action, reward, state sequences…

- Predict the next state given the current state + action
- Predict the reward given the current state + action

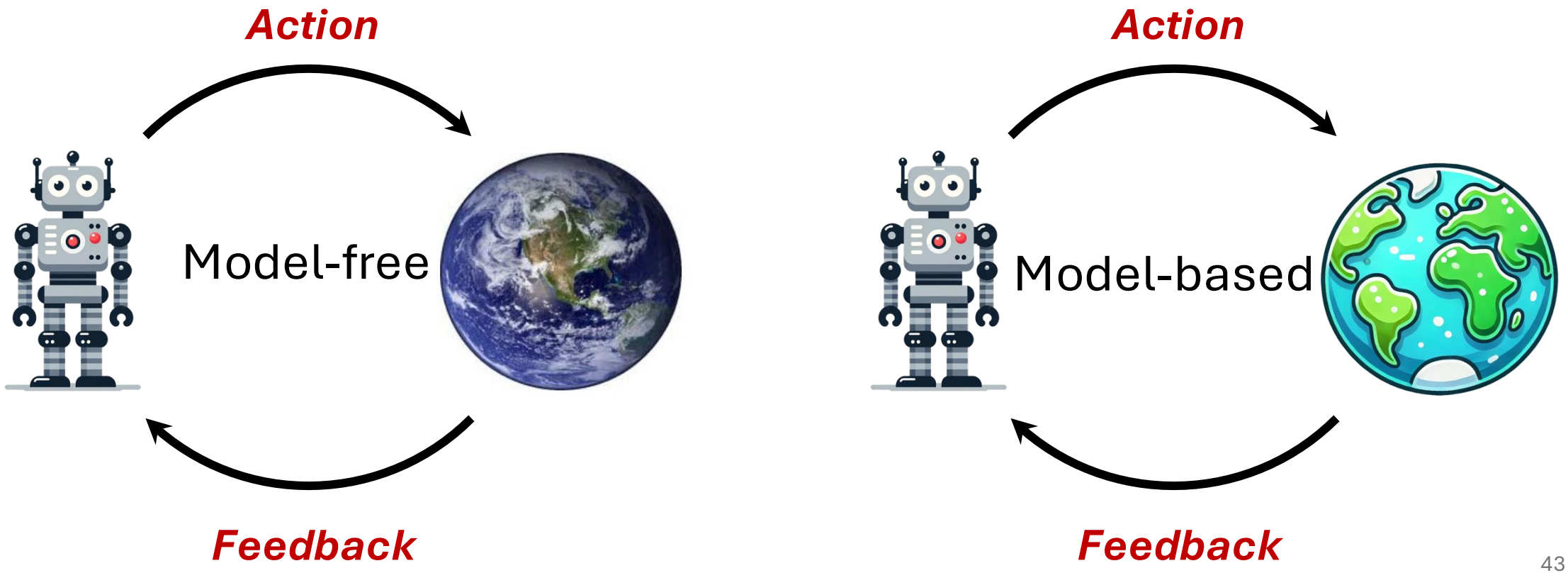
Great! So now that we have learned models of the transition and reward functions, can we use policy/value iteration?

In theory? Yes. In practice? No.

- Our state spaces are too high-dimensional for dynamic programming

Model-based reinforcement learning

Idea: In MBRL we learn a model of the environment and reward and use this to perform reinforcement learning



Why do we care? What are the benefits?

Why do we care? What are the benefits?

- In some cases, it is more sample-efficient to learn a model of the environment than to directly learn a policy in the environment
  - Especially important if interactions are safety-critical!
- Once a model is learned, it can be re-used many times
- We can plan ahead by simulating what will happen before acting
- Can be used to guide exploration and representation learning
  - Uncertain what next state looks like? Then we haven't been there enough!

What are the drawbacks?

What are the drawbacks?

- The learned model may not be accurate (e.g. high dimensions)
  - Particularly dangerous as errors propagate over time!
- If dynamics are hard to predict, may result in under-modeling
  - Faster learning at beginning, but non-optimal asymptotic performance
- If model is overfit, may lead to policy taking advantage of mistakes
  - Overfitting can induce spurious correlations in resulting policies
- Capturing stochasticity in future events is hard

Early MBRL: Dyna

Dyna, an Integrated Architecture for Learning, Planning, and Reacting

Richard S. Sutton
GTE Laboratories Incorporated
Waltham, MA 02254
sutton@gte.com

Abstract

Dyna is an AI architecture that integrates learning, planning, and reactive execution. Learning methods are used in Dyna both for compiling planning results and for updating a model of the effects of the agent's actions on the world. Planning is incremental and can use the probabilistic and ofttimes incorrect world models generated by learning processes. Execution is fully reactive in the sense that no planning intervenes between perception and action. Dyna relies on machine learning methods for learning from examples—these are among the basic building blocks making up the architecture—yet is not tied to any particular method. This paper briefly introduces Dyna and discusses its strengths and weaknesses with respect to other architectures.

1 Introduction to Dyna

The Dyna architecture attempts to integrate

- Trial-and-error learning of an optimal *reactive policy*, a mapping from situations to actions;
- Learning of domain knowledge in the form of an *action model*, a black box that takes as input a situation and action and outputs a prediction of the immediate next situation;
- Planning: finding the optimal reactive policy given do-

Figure 1: The Problem Formulation Used in Dyna. The agent's object is to maximize the total reward it receives over time.[1]

REPEAT FOREVER:

1. Observe the world's state and reactively choose an action based on it;
2. Observe resultant reward and new state;
3. Apply reinforcement learning to this experience;
4. Update action model based on this experience;
5. Repeat K times:
   - **5.1** Choose a hypothetical world state and action;
   - **5.2** Predict resultant reward and new state using action model;
   - **5.3** Apply reinforcement learning to this hypothetical experience.

Figure 2: A Generic Dyna Algorithm.

Early MBRL: Dyna

Early, influential approach

- While policy/value iteration and optimal control methods already existed, I would argue this is the first “modern” MBRL algo

Idea: combines model-free and model-based RL

1. Sample experience from real environment
2. Use experience to update policy (model-free RL)
3. Use experience to update environment model
4. Sample “synthetic experience” from environment model
5. Use “synthetic experience” to update policy (model-based RL)

Early MBRL: Dyna

An interesting read, even today!

- Short and well-written, I recommend going through it

“The main idea of Dyna is the old, commonsense idea that planning is 'trying things in your head,' using an internal model of the world. This suggests the existence of a more primitive process for trying things not in your head, but through direct interaction with the world. Reinforcement learning is the name we use for this more primitive, direct kind of trying, and Dyna is the extension of reinforcement learning to include a learned world model.”

What do we gain?

Dyna maximizes the efficiency of real interactions

By using it to learn an environment model:

- Faster credit assignment, as TD updates propagate further using the synthetic data
- Better coverage of state-action space (e.g. rare events)
- Trades compute for environment interactions

Another approach: explicitly planning ahead

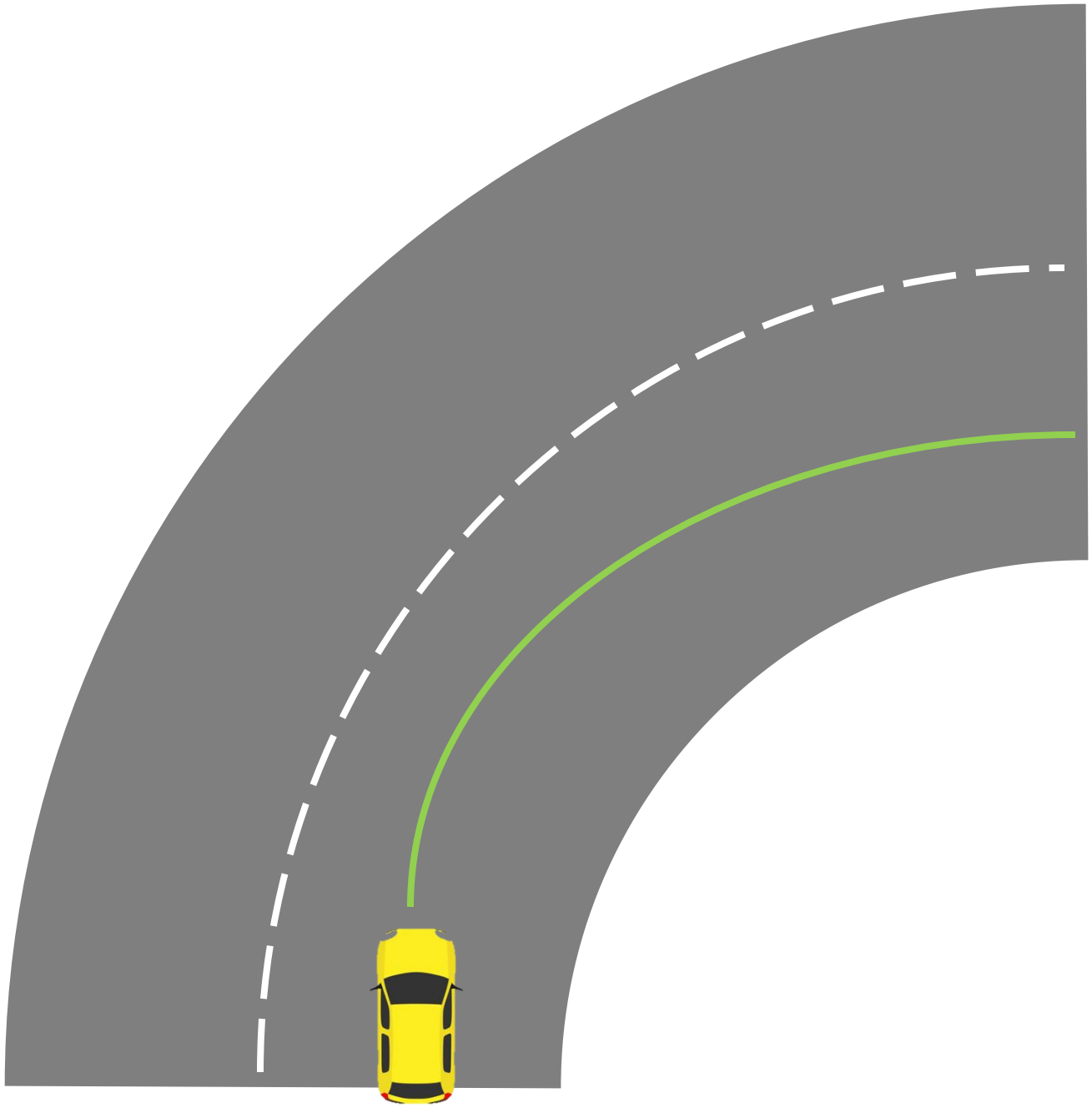
Value functions implicitly encode dynamics with respect to a policy

- What is the expected future reward if I follow my policy?

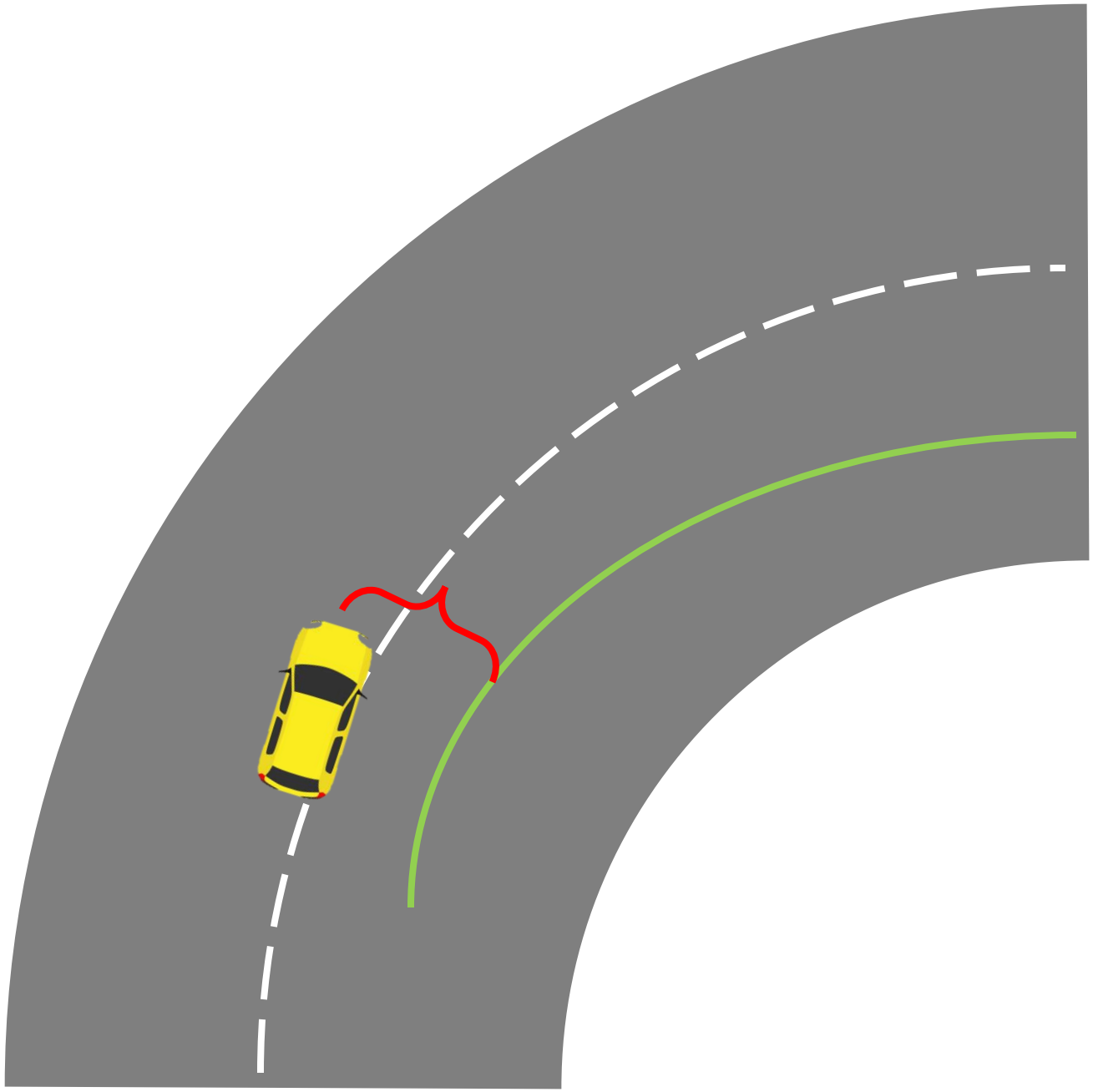
But if we have a model of the environment, we can explicitly simulate what will happen!

This means we can decouple dynamics from rewards

- Allows for planning optimal actions

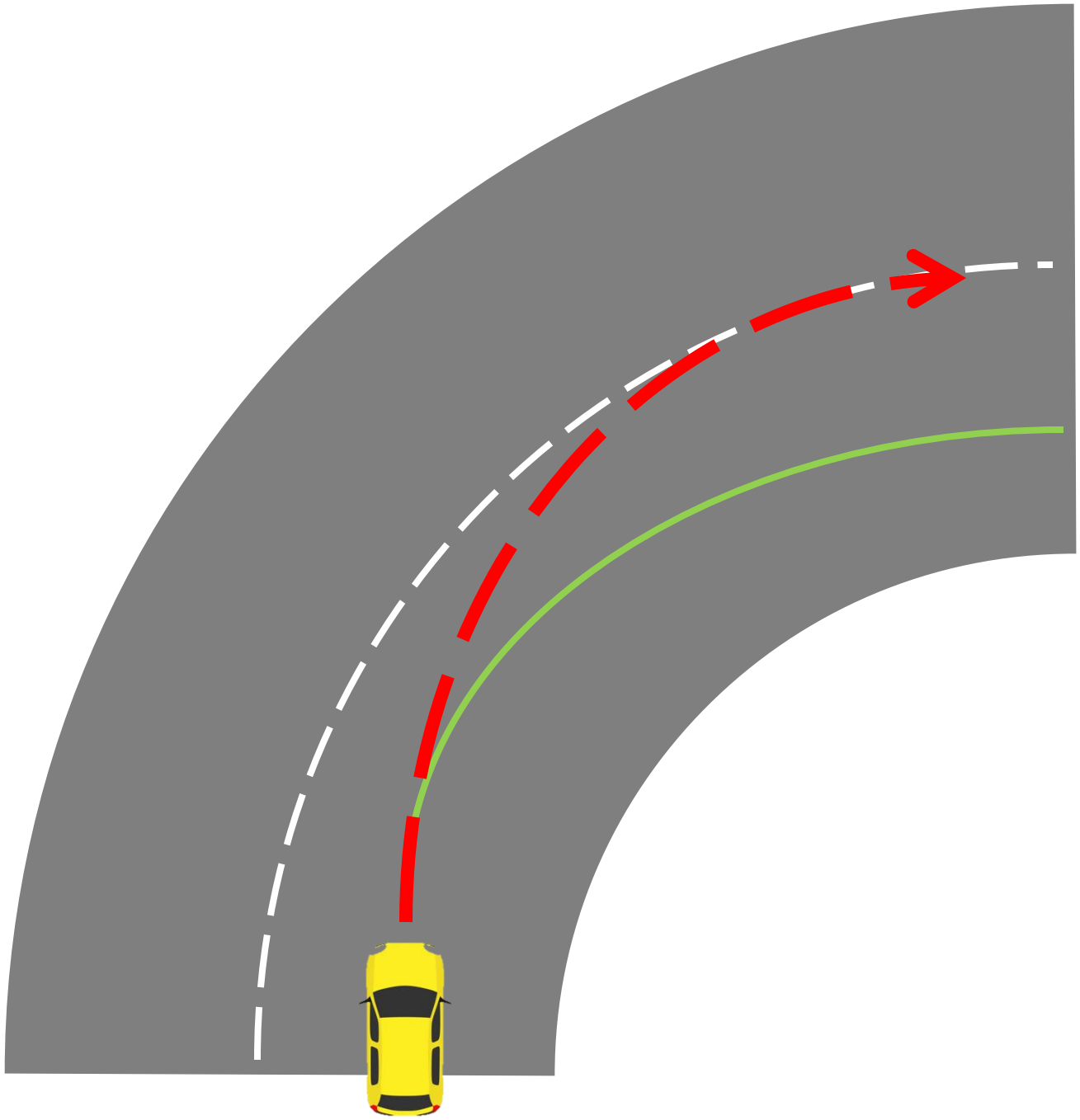


Desired trajectory through curve



Error is lateral distance from car to desired trajectory

- Cross-track error



Reactive controllers (PID) overshoot

In a sharp curve, the error increases very quickly

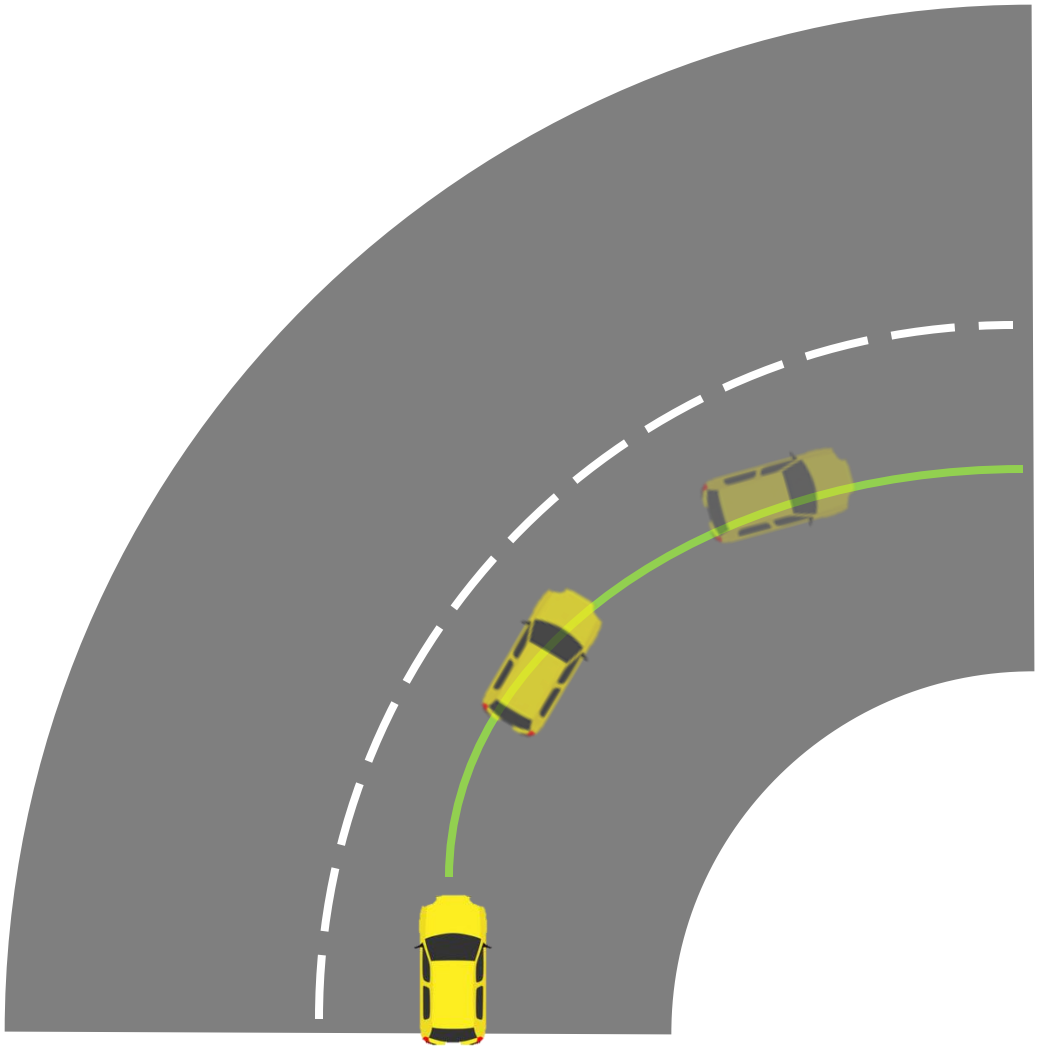
Due to mechanical and physical constraints, the controller cannot react fast enough and the car **overshoots**

After the car rounds the curve, it has over-corrected and now **oscillates** around the desired trajectory

What we need is a *proactive* controller

A controller that can predict the trajectory over a future horizon

Choose actions that produce an optimal **future** trajectory



Model predictive control

Definition: feedback control method that optimizes future behavior over finite time horizon

Three steps:

1. Predict future states using **system model**
2. Solve optimization problem to **minimize cost** over future states
3. Apply only the **first action** and repeat process next step

Model predictive control

System model:

$$x_{t+1} = f(x_t, u_t)$$

Next state Current state Current action

What does a system model look like?

The car is subject to dynamics

Consider a car…

State/configuration parameterized by…

x, y, θ

X-position, Y-position, heading (w.r.t x-axis)

What does a system model look like?

The car is subject to dynamics

Consider a car…

Control inputs defined as…
v, δ

Velocity, steering angle

What does a system model look like?

The car is subject to dynamics

Consider a car…

System parameters…

L

Wheelbase

Bicycle model equations of of motion

$$x_{t+1} = x_t + v_t \cos(\theta_t)\, \Delta t$$

$$y_{t+1} = y_t + v_t \sin(\theta_t)\, \Delta t$$

$$\theta_{t+1} = \theta_t + \frac{v_t}{L} \tan(\delta_t)\, \Delta t$$

Bicycle model equations of of motion

$$x_{t+1} = x_t + v_t \cos(\theta_t)\, \Delta t$$

$$y_{t+1} = y_t + v_t \sin(\theta_t)\, \Delta t$$

$$\theta_{t+1} = \theta_t + \frac{v_t}{L} \tan(\delta_t)\, \Delta t$$

Assumptions:

- Constant velocity during Δt
- Idealized -- no friction, slip, skid, or other disturbances

Model predictive control

Example (common) cost function:

$$J = \sum_{t=0}^{T} \left\| x_t - x_{target} \right\|^2 + \|u\|^2$$

T-step horizon

State error

Control penalty

Cost function minimizes state error over finite horizon while avoiding large control signals

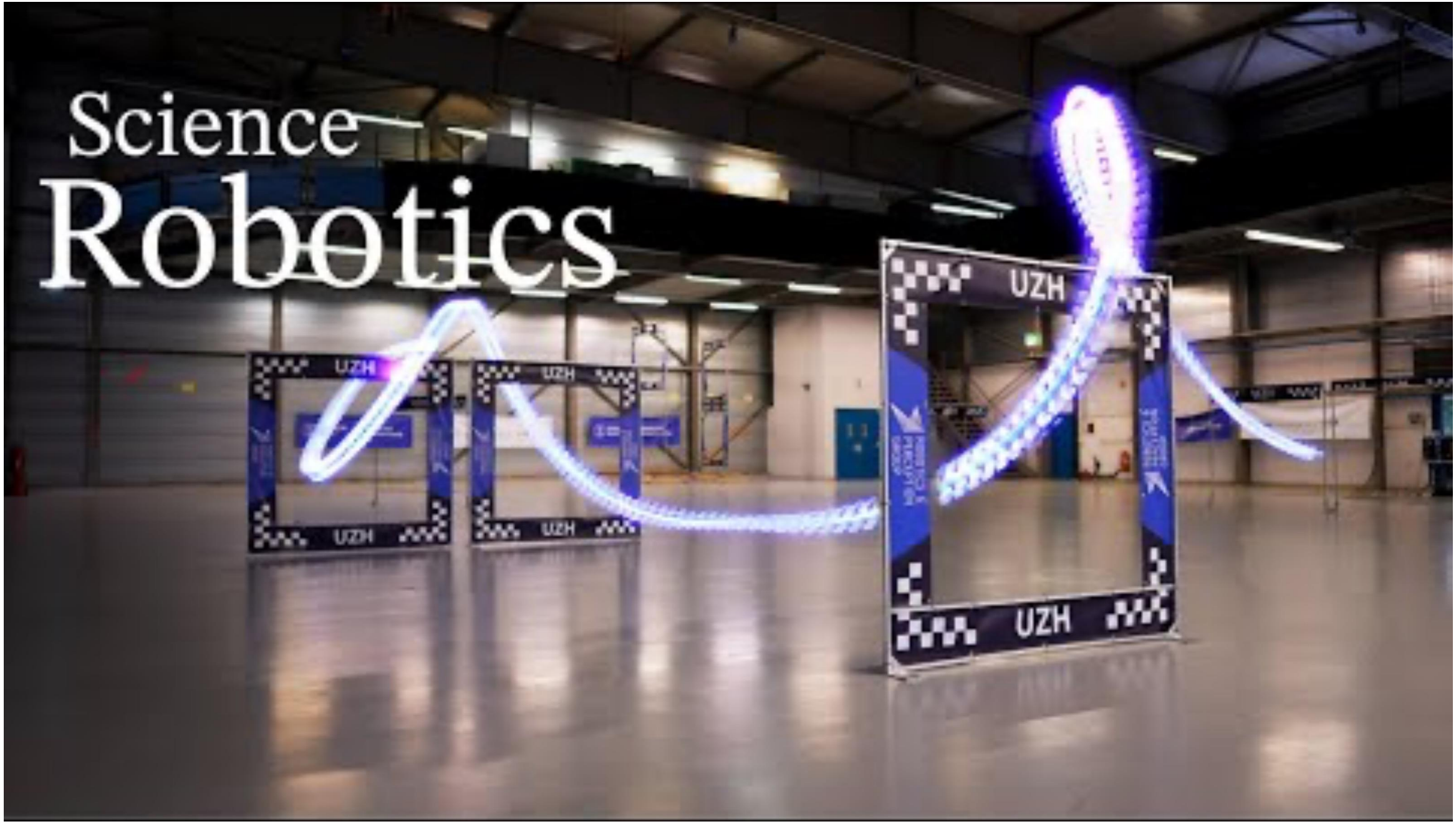
Model predictive control

Example constraints:

$$u_{min} \leq u_t \leq u_{max} \text{ and } x_{min} \leq x_t \leq x_{max}$$

Control input bounds

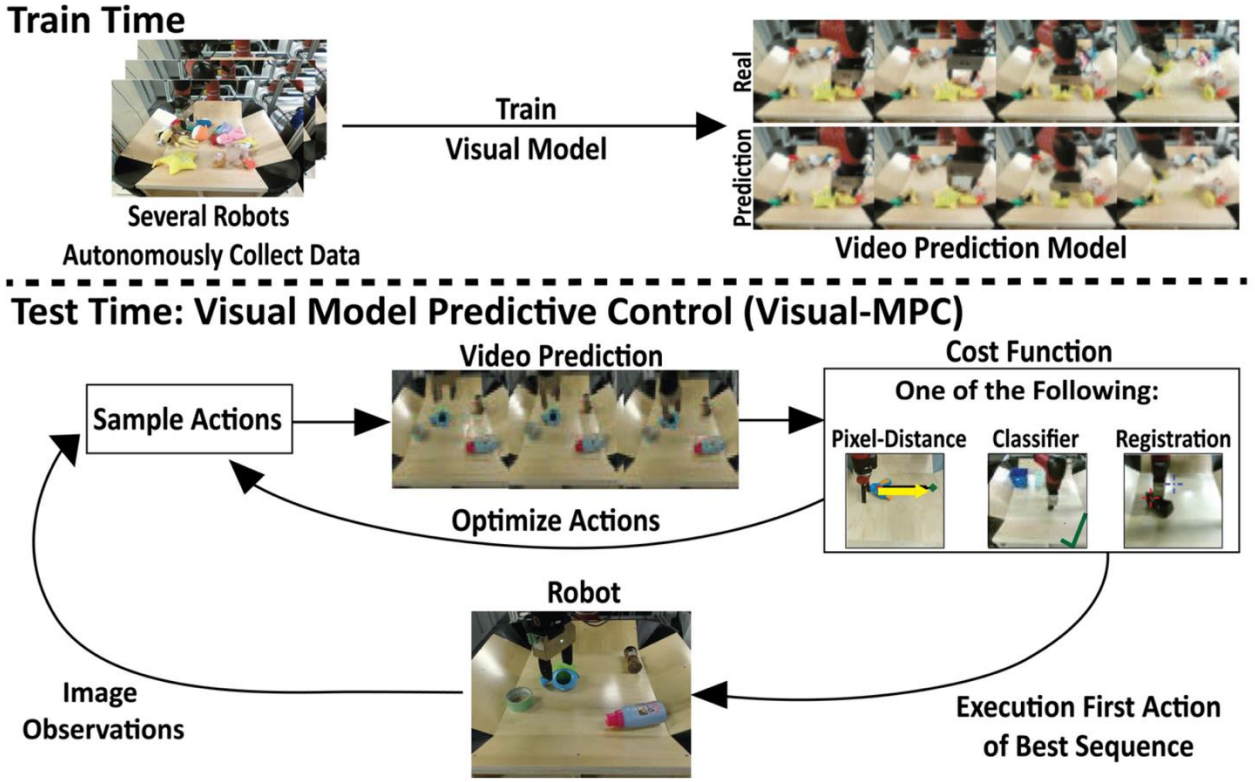
State bounds



https://youtu.be/HGULBBAo5lA?si=hN9A3gTiL58ES9Kn

Deep visual foresight

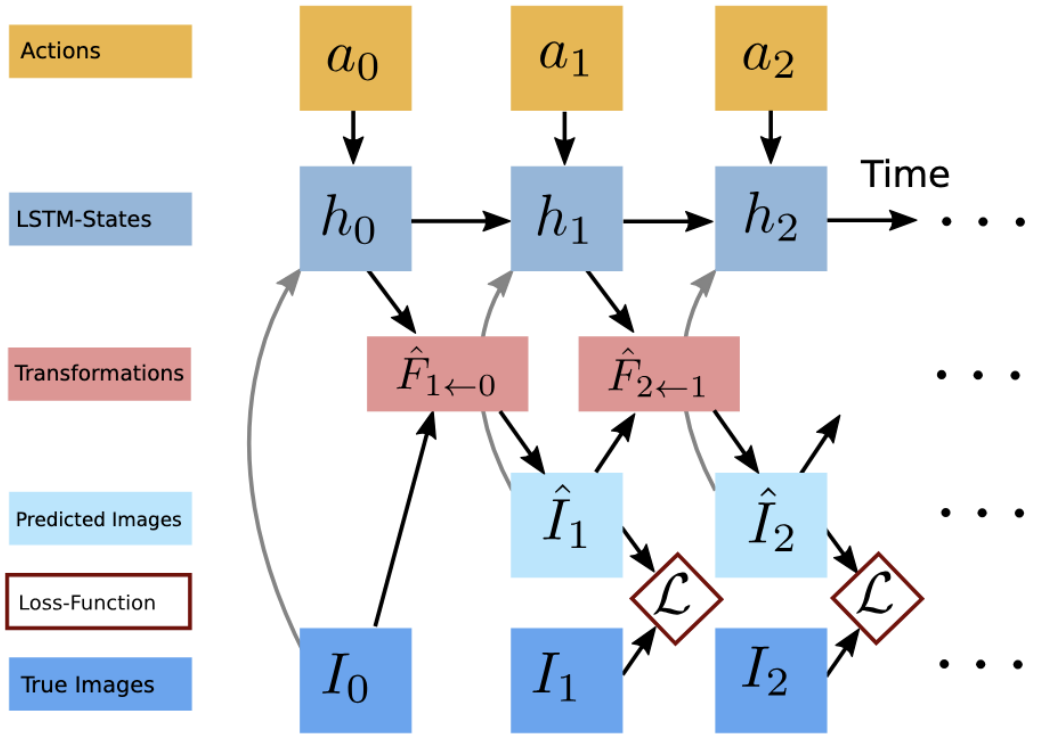
Idea: learn a video prediction model which predicts future image states and use this for model predictive control



Ebert et al, Visual Foresight: Model-Based Deep Reinforcement Learning for Vision-Based Robotic Control, 2018.

Video prediction model

Given the current RGB image and a sequence of actions, predict the corresponding sequence of future images



Visual model predictive control

Use model predictive control to choose the sequence of actions that maximize the probability of achieving the desired goal

- Goal in this work: pushing objects to desired location
- Note: this is planning not learning!

MPC in a nutshell: use (video) model to observe effect of actions such that chosen actions are optimal over a finite horizon

Since the policy is goal-conditioned and the prediction is in pixel-space, no need to learn a reward function!

https://youtu.be/6k7GHG4IUCY?si=04p3OqTHHF3fUCLI&t=33