Model-Based Reinforcement Learning

CS 285

Instructor: Sergey Levine
UC Berkeley

Today's Lecture

1. Basics of model-based RL: learn a model, use model for control
    - Why does naïve approach not work?
    - The effect of distributional shift in model-based RL
2. Uncertainty in model-based RL
3. Model-based RL with complex observations
4. Next time: **policy learning** with model-based RL

- Goals:
    - Understand how to build model-based RL algorithms
    - Understand the important considerations for model-based RL
    - Understand the tradeoffs between different model class choices

this time, we'll discuss how to learn a model. once you know the model, you can learn algorithms from the previous lecture on how to plan with them

Why learn the model?

If we knew $f(\mathbf{s}_t, \mathbf{a}_t) = \mathbf{s}_{t+1}$, we could use the tools from last week.

(or $p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$ in the stochastic case)

So let's learn $f(\mathbf{s}_t, \mathbf{a}_t)$ from data, and *then* plan through it!

model-based reinforcement learning version 0.5:

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

note that we don't care about rewards yet. we just want to learn the dynamics

learn a model (using a NN) that takes state s and action a, and predicts state s'. Do supervised learning to fit

using algorithms we created last week

Does it work?

Yes!

take data and use it to identify unknown parameters in a model. Typically, you actually know the model, but you don't know the weights for some parameters in that

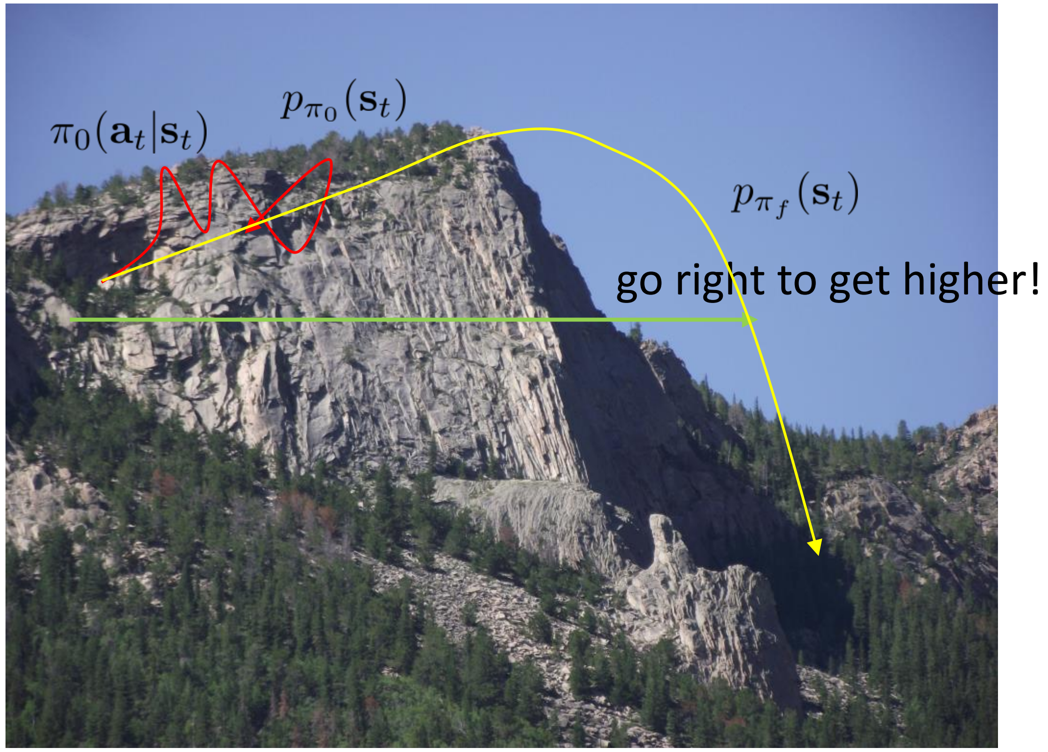
you need to explore a lot of different states and actions

- Essentially how system identification works in classical robotics
- Some care should be taken to design a good base policy
- Particularly effective if we can hand-engineer a dynamics representation using our knowledge of physics, and fit just a few parameters

it doesn't really work using large models like neural networks

Does it work?

No!


$\pi_0(\mathbf{a}_t|\mathbf{s}_t)$
$p_{\pi_0}(\mathbf{s}_t)$
$p_{\pi_f}(\mathbf{s}_t)$
go right to get higher!

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

the data was generated by a distribution under the random policy. the planning was done by a distribution under the policy we create after learning the dynamics model. in other words, the data we obtained using our random policy is distributionally different the states we get from rolling out our planning algorithm. So we go to states that had low probability of going to using our random policy. So then our model makes erroneous predictions at those states. Then we might get to other states where our model makes even more erroneous predictions, and it gets out of hand

$$\underbrace{p_{\pi_f}}_{\text{planned policy}}(\mathbf{s}_t) \neq \underbrace{p_{\pi_0}}_{\text{random policy}}(\mathbf{s}_t)$$

- Distribution mismatch problem becomes exacerbated as we use more expressive model classes

because more expressive models fit more tightly to the data we got from the random policy

however, system identification works in robotics because our model only has like 3 parameters, so it's harder to overfit

Can we do better?

can we make $p_{\pi_0}(\mathbf{s}_t) = p_{\pi_f}(\mathbf{s}_t)$?

where have we seen that before? need to collect data from $p_{\pi_f}(\mathbf{s}_t)$

model-based reinforcement learning version 1.0:

this is the simplest model-based RL method that generally works, at least conceptually

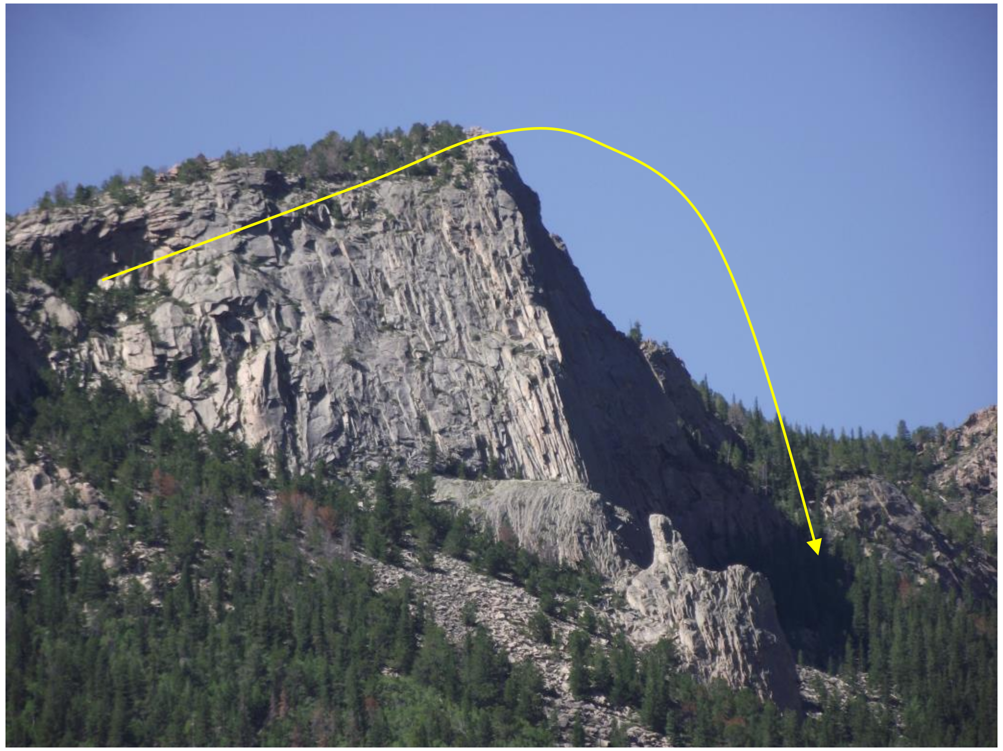
1. run base policy $\pi_0(\mathbf{a}_t|\mathbf{s}_t)$ (e.g., random policy) to collect $\mathcal{D} = \{(\mathbf{s}, \mathbf{a}, \mathbf{s}')_i\}$
2. learn dynamics model $f(\mathbf{s}, \mathbf{a})$ to minimize $\sum_i \|f(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{s}'_i\|^2$
3. plan through $f(\mathbf{s},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

this loop mitigates distributional shift

this is essentially like DAGGER but for model-based RL

What if we make a mistake?

in a case of walking off a cliff, there's not much you can do. but for a car, there is

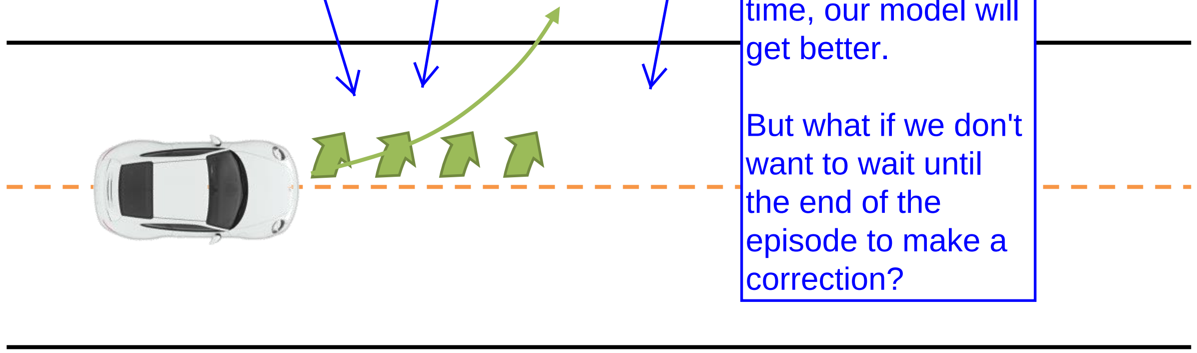


let's say our model isn't great yet, and predicts that you'll go straight if you steer slightly to the left

when we execute our model, we will actually go left

using our RL algo 1.0, we will continue to fail, but we will add all this to the dataset. This means that next time, our model will get better.

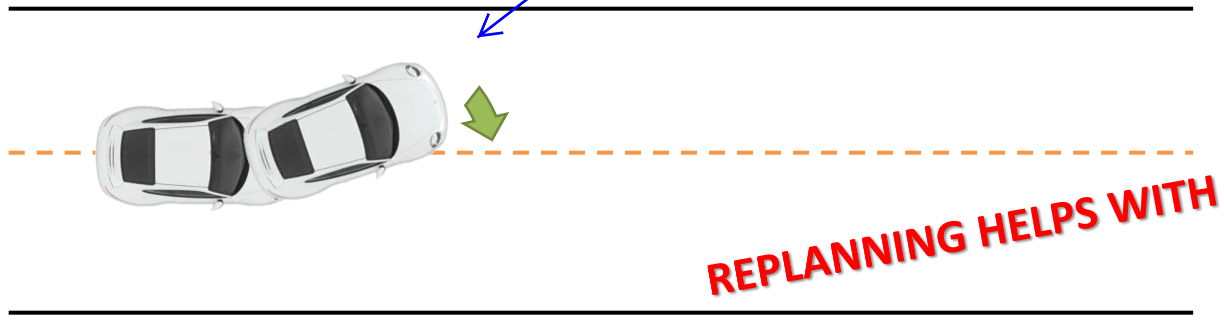
But what if we don't want to wait until the end of the episode to make a correction?



Can we do better?

when we make a mistake, we can look at our new state, then ask our model what we should do next. so instead of executing our old plan, we find a new plan. This is known as model-predictive-control

this is more computationally expensive because you have to complete the planning at every timestep. but, you can do much better with a worse model



REPLANNING HELPS WITH MODEL ERRORS

this always works better than version 1.0, but is more computationally expensive

model-based reinforcement learning version 1.5:

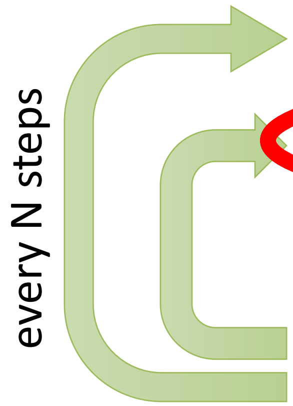
1. run base policy $\pi_0(\mathbf{a}_t|\mathbf{s}_t)$ (e.g., random policy) to collect $\mathcal{D} = \{(\mathbf{s}, \mathbf{a}, \mathbf{s}')_i\}$
2. learn dynamics model $f(\mathbf{s}, \mathbf{a})$ to minimize $\sum_i \|f(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{s}'_i\|^2$
3. plan through $f(\mathbf{s}, \mathbf{a})$ 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 steps

This will be on HW4!

How to replan?

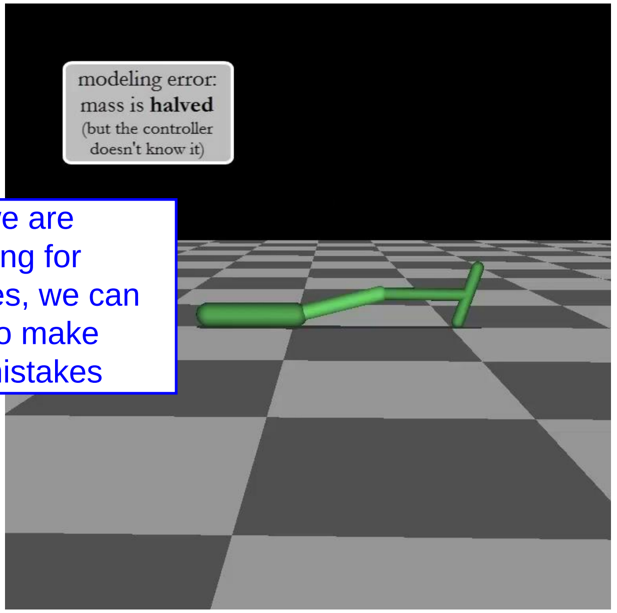
model-based reinforcement learning version 1.5:



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 $\mathcal{D}$

- The more you replan, the less perfect each individual plan needs to be
- Can use shorter horizons
- Even random sampling can often work well here!

since we are correcting for mistakes, we can afford to make more mistakes

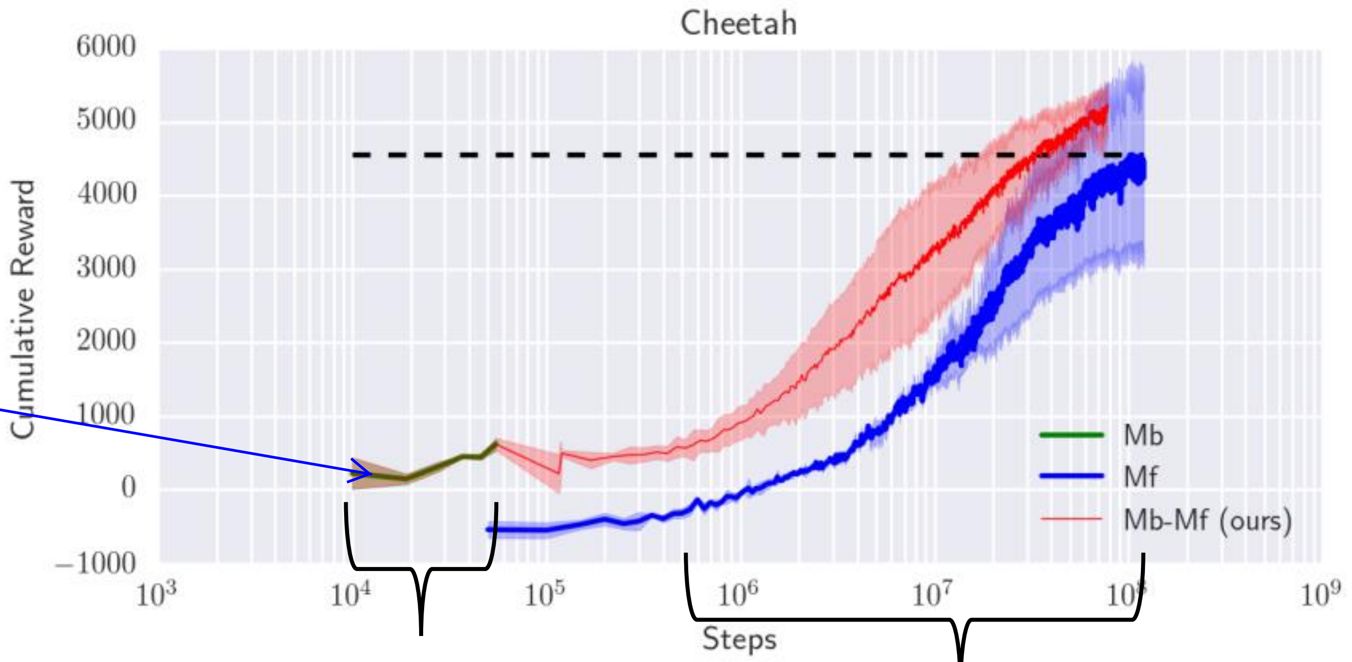

modeling error: mass is **halved** (but the controller doesn't know it)

Uncertainty in Model-Based RL

in principle, model-based RL v1.0
can solve the RL problem, in
practice it has major issues

A performance gap in model-based RL

they ran the basic model-based RL v1.5 starting from scratch. then they used this to bootstrap a model-free RL learner.

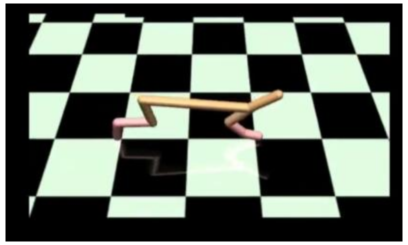


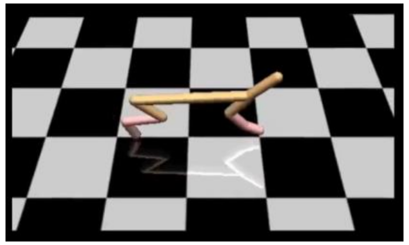
model-free gets better performance

model-based can improve very quickly. notice that the x-axis is logarithmic...so the model-free was trained much longer

pure model-based
(about 10 minutes real time)

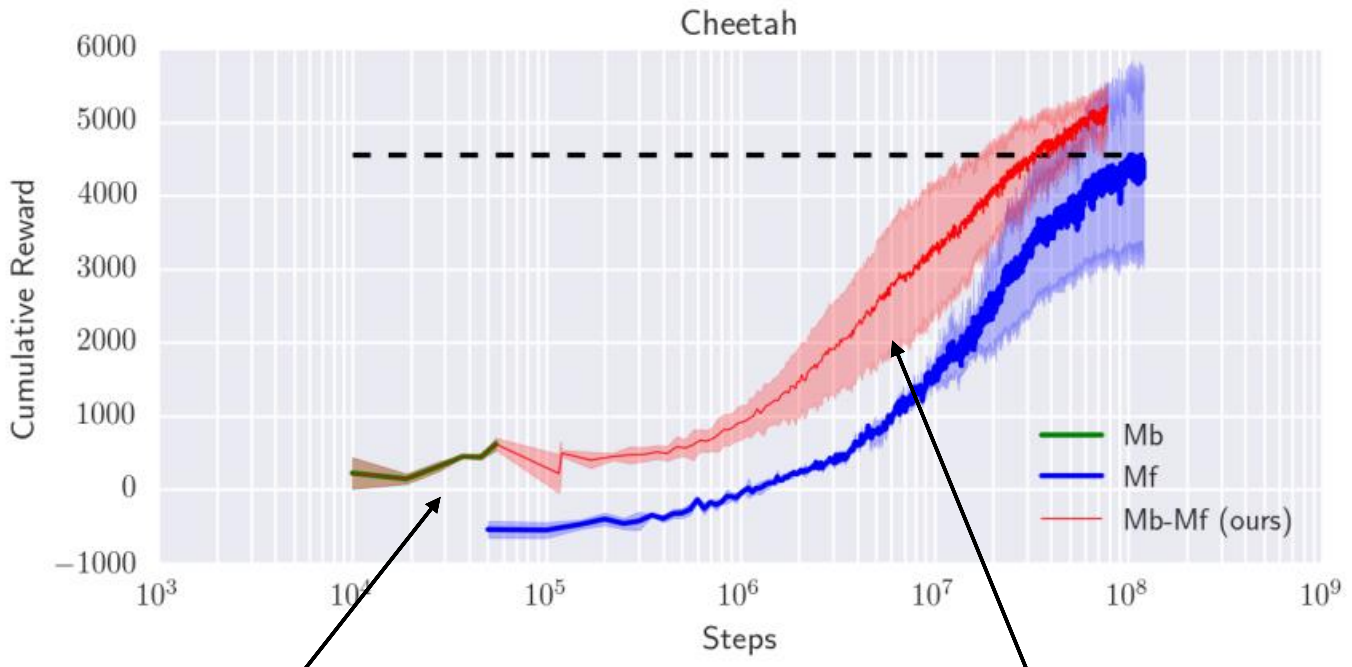
model-free training
(about 10 days...)





Nagabandi, Kahn, Fearing, L. ICRA 2018

Why the performance gap?



need to not overfit here…

…but still have high capacity over here

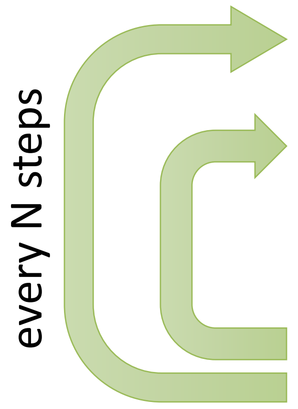
we're mitigating the distributional shift issue by using our model to collect additional data, but that means our model needs to be pretty good early on even when it doesn't have much data.

NN do well with large data, but struggle with low data. Often times they over fit small datasets. So NN does poorly in the initial stages, and they don't produce a good enough exploration, so they get stuck

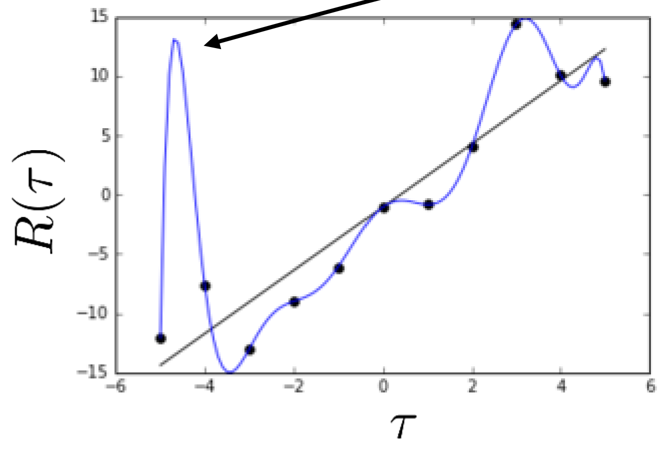
Why the performance gap?

so we have a performance gap due to overfitting exacerbated by distributional shift

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 $\mathcal{D}$

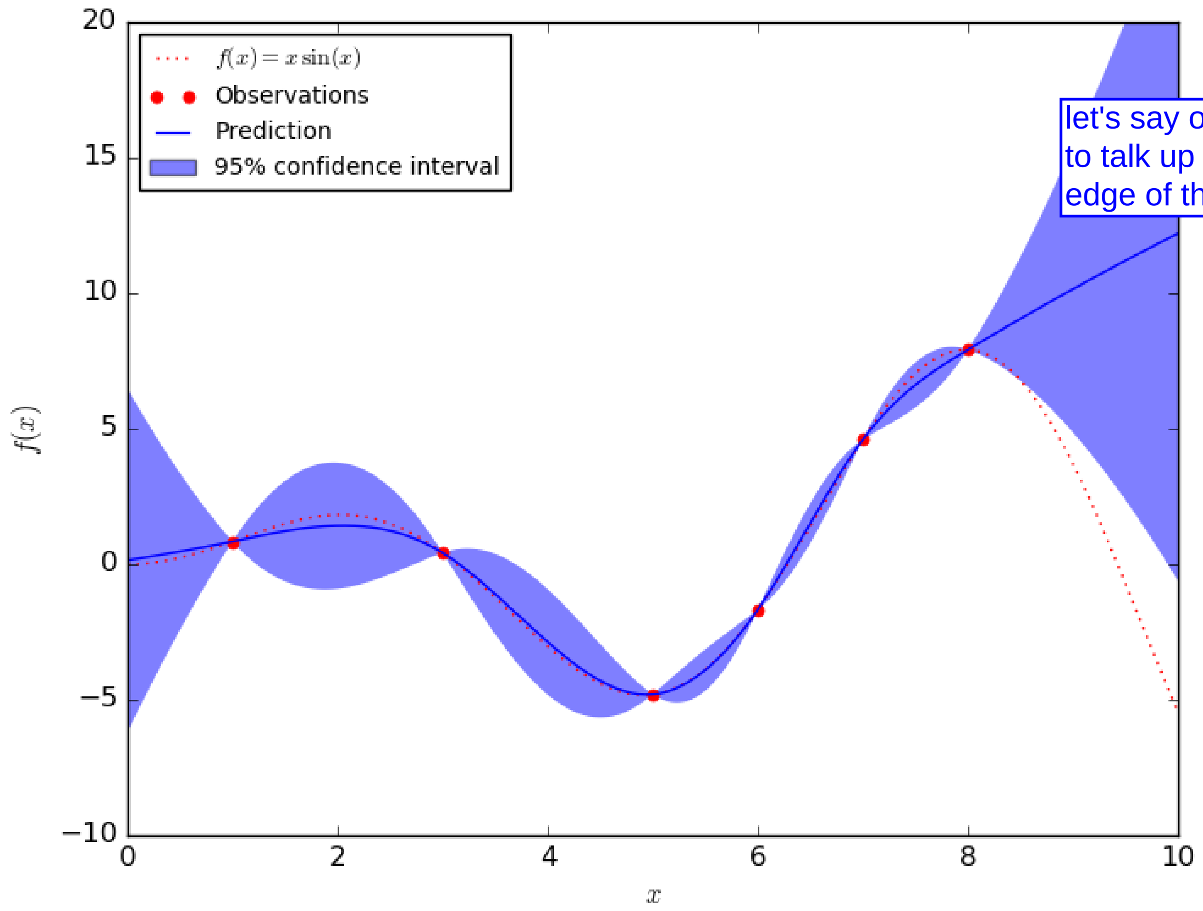


very tempting to go here...

our model will have some error. Sometimes this error will make it look like a trajectory has higher reward than it actually does. The planner uses that projection (and it doesn't know the model is wrong) to select trajectories that result in the largest (positive) mistakes in the model. So our planner exploits errors in our over-fitted model

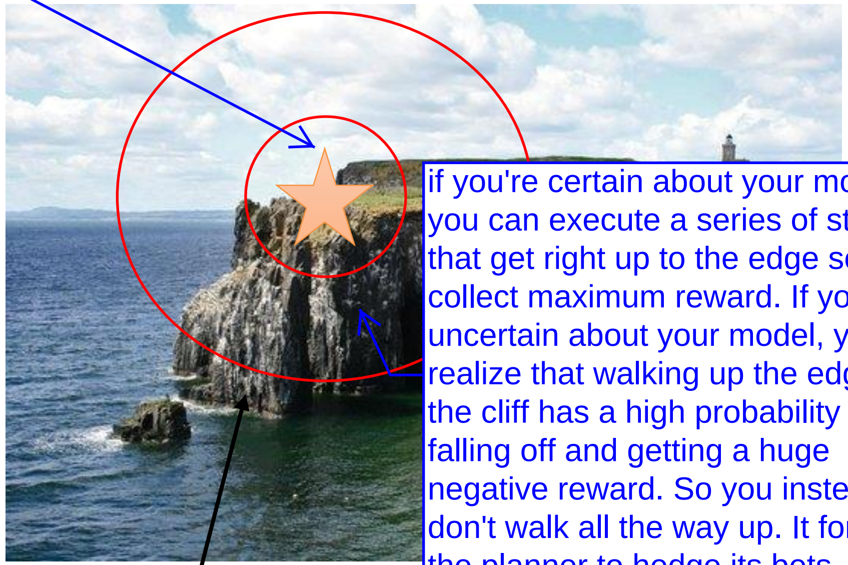
How can uncertainty estimation help?

Uncertainty estimation, for every state-action pair, we don't just predict the next state s', but instead we predict a distribution over all the next states we could reach under our uncertainty about the model.



$$p_{\pi_f}(\mathbf{s}_t) \neq p_{\pi_0}(\mathbf{s}_t)$$

let's say our goal is to talk up to the edge of the cliff



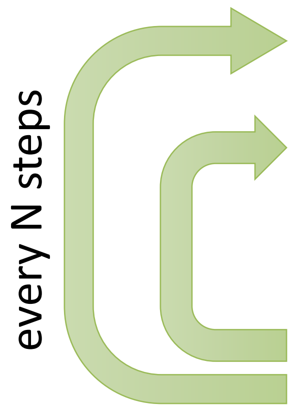
if you're certain about your model, you can execute a series of steps that get right up to the edge so you collect maximum reward. If you're uncertain about your model, you realize that walking up the edge of the cliff has a high probability of falling off and getting a huge negative reward. So you instead don't walk all the way up. It forces the planner to hedge its bets. Note that this phenomena could emerge without us doing anything special.

expected reward under high-variance prediction is **very** low, even though mean is the same!

Intuition behind uncertainty-aware RL

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 $\mathcal{D}$



only take actions for which we think we'll get high reward in expectation (w.r.t. uncertain dynamics)

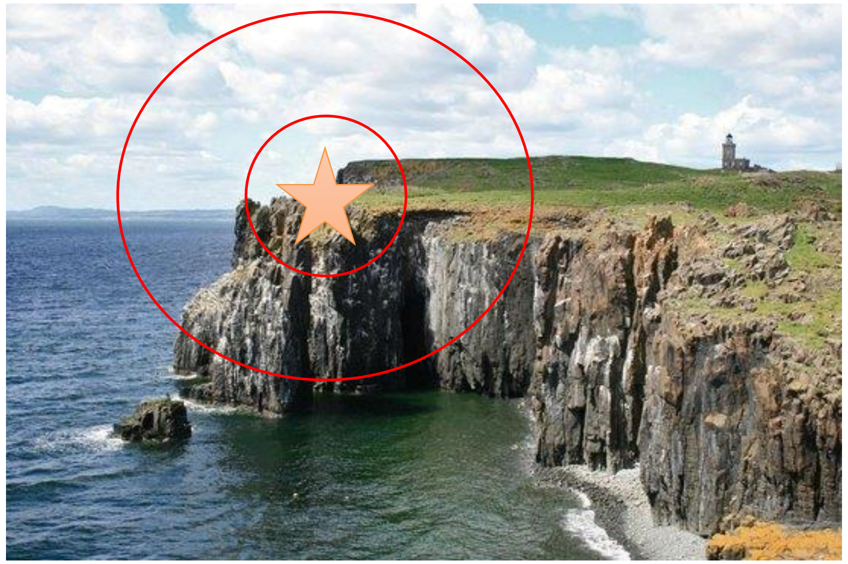
This avoids "exploiting" the model

The model will then adapt and get better

this will help us especially in the early stages of training when our model isn't very good and our uncertainty is high

intuition: at first you don't walk that close to the cliff. you collect data and refine your model. over time your uncertainty decreases and you walk closer and closer to the cliff

There are a few caveats…



if you're very cautious, you might never end up going near higher-reward regions, so you don't explore

Need to explore to get better

Expected value is not the same as pessimistic value

Expected value is not the same as optimistic value

…but expected value is often a good start

Uncertainty-Aware Neural Net Models

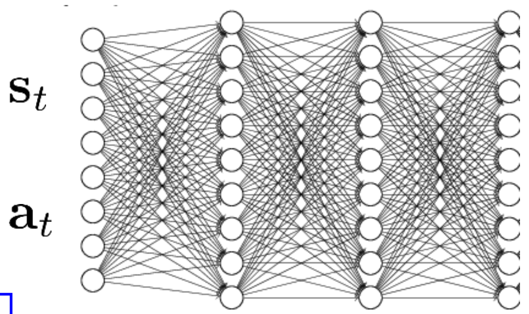
let's talk about how we can train uncertainty-aware NN models that
can serve as our uncertainty-aware dynamics models

How can we have uncertainty-aware models?

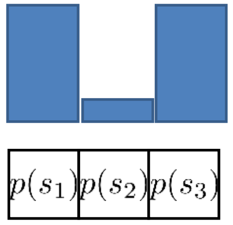
Idea 1: use output entropy

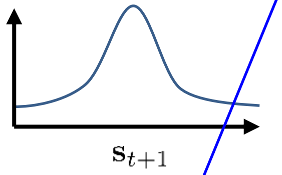
this is a bad idea and doesn't work.

softmax in discrete-setting, or multivariate gaussian distribution in continuous setting



$p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$

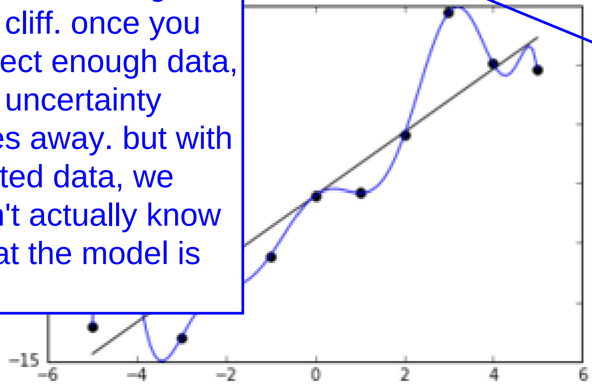




why is this not enough?

The problem we're having: the planner exploits the errors. Ultimately, it finds out-of-distribution actions that lead to out-of-distribution states, that lead to more out-of-distribution states. This means that our model is forced to make predictions for actions and states it wasn't trained on. The problem is that if the model is outputting the uncertainty and it's being trained with the maximum likelihood, the uncertainty itself is also not accurate for out-of-distribution inputs! Thus it will output erroneous means, and also, erroneous variance! It will output extremely overconfident predictions that are good on the training data, but are incorrect on test points. The NN is outputting the wrong type of incertainty.

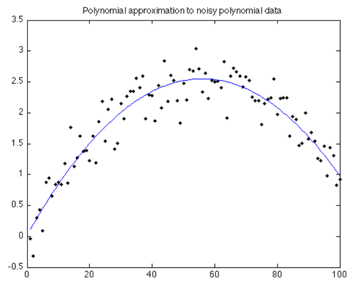
this is the setting where we're walking around the edge of the cliff. once you collect enough data, the uncertainty goes away. but with limited data, we don't actually know what the model is



what is the variance here?

Two types of uncertainty:

aleatoric or *statistical* uncertainty $\longrightarrow$



the function itself is noisy. this doesn't necessarily go down with more data if the true function is noisy. for example, if you learn the model for a game of "chance", seeing more data won't make it deterministic. The game itself is random.

we don't know what the right function is. This decreases with more data.

$\longleftarrow$ *epistemic* or *model* uncertainty

"the model is certain about the data, but we are not certain about the model"

this is what we want and MLE training doesn't give us this

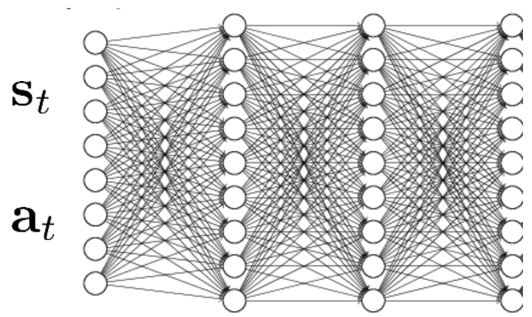
How can we have uncertainty-aware models?

Idea 2: estimate model uncertainty

to not be certain about the model, we need to represent a distribution over models

being uncertain about the model really means being uncertain about the parameters theta

"the model is certain about the data, but we are not certain about the model"



$p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$

parameters θ

usually, we estimate

$$\arg\max_\theta \log p(\theta|\mathcal{D}) = \arg\max_\theta \log p(\mathcal{D}|\theta)$$

can we instead estimate $p(\theta|\mathcal{D})$?

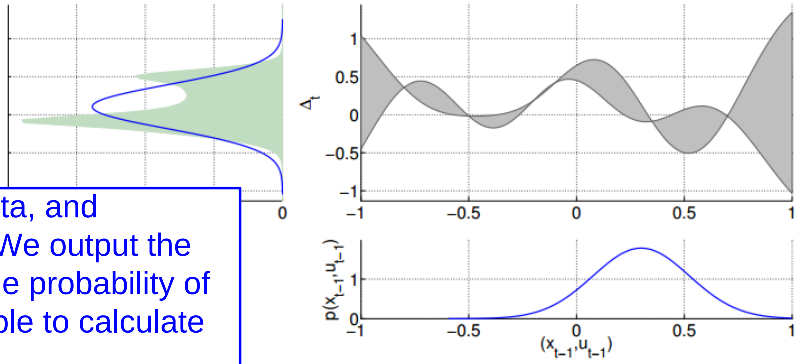
the entropy of this tells us the model uncertainty!

instead of estimating the most likely theta (via the argmax), what if we tried to estimate the distribution of theta, and then use this to get our uncertainty. This is the right kind of uncertainty we want to get.

predict according to:

$$\int p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t, \theta)p(\theta|\mathcal{D})d\theta$$

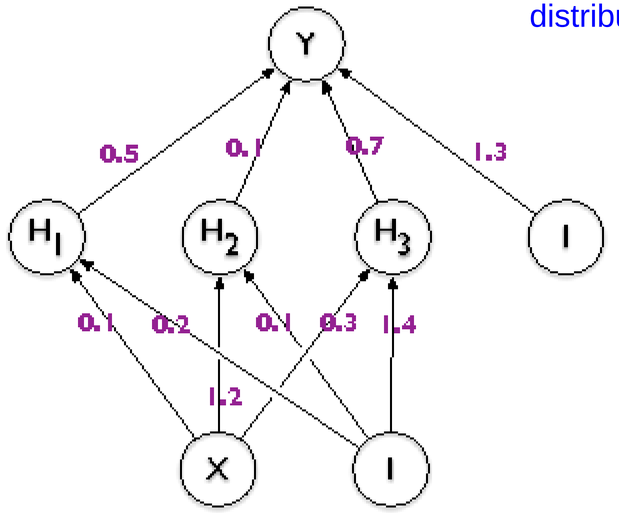
instead of taking the most likely theta, and outputting the probability of s_t+1. We output the parameters and multiply them by the probability of next states. For NN, this is intractable to calculate exactly

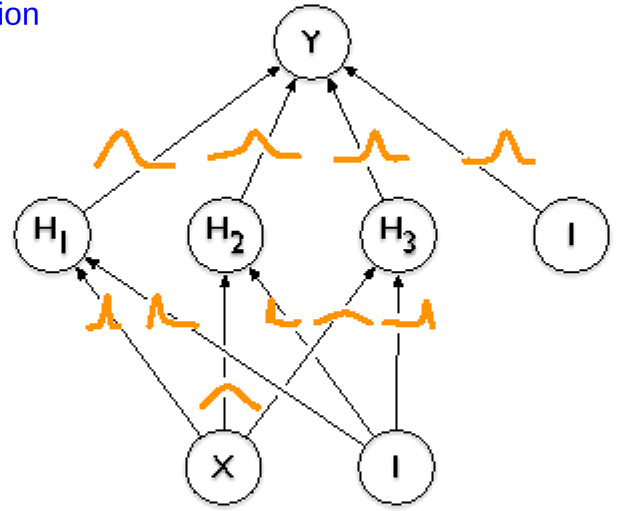


Quick overview of Bayesian neural networks

in a standard NN, the weights are just numbers.

in bayesian NN, there's a distribution over every weight. if you want to make a prediction, you sample a neural net over the distribution of NN and ask for its prediction. if you want to get a posterior distribution over predictions...if you want to sample from the posterior distribution, you'd sample a neural net and then sample a Y given your NN. you could do this many times if you wanted to get many samples to get a general impression of the true posterior distribution





common approximation:

$$p(\theta|\mathcal{D}) = \prod_i p(\theta_i|\mathcal{D})$$

$$p(\theta_i|\mathcal{D}) = \mathcal{N}(\mu_i, \sigma_i)$$

expected weight

uncertainty about the weight

it's common to represent each independent marginals with a guassian distribution. For every model parameter, we learn its mean and uncertainty

the posterior distribution is the product of marginals. this means that each parameter distributed randomly but is independent of the other weights. In practice this isn't good because the weights aren't independent and they have tightly interacting effects. So if you varied the weights independently, they the NN could change quite a lot. So using a product of independent marginals to estimate the parameter posterior is a very crude approximation, but is simple and tractable. So it's used quite often

For more, see:

Blundell et al., Weight Uncertainty in Neural Networks

Gal et al., Concrete Dropout

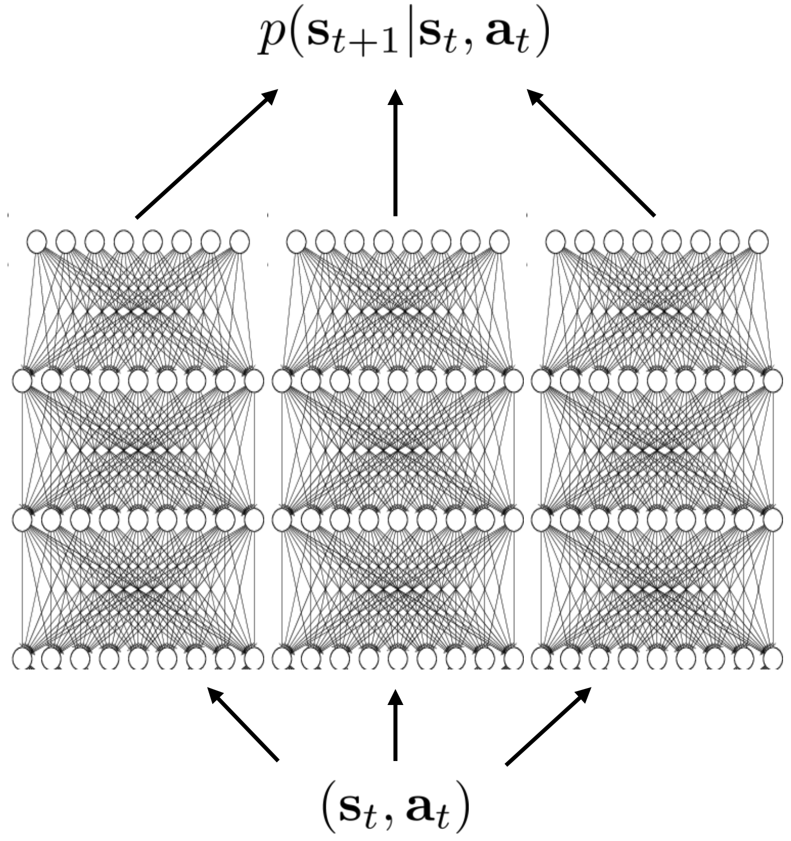
We'll learn more about variational inference later!

Bootstrap ensembles

today we'll talk about an even similar method, which from Sergey Levine's experience, works even better for model-based RL

Instead of training one NN to get us a distribution over the next state, we instead train many different NN (and we make sure they're a bit different). Ideally, they all are accurate on the training data, but would make different mistakes on the test data.

by training an ensemble of models, we can have them vote on what the next state will be, and we can estimate their uncertainty using the spread of their different predictions


$p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$
$(\mathbf{s}_t, \mathbf{a}_t)$

Train multiple models and see if they agree!

formally: $p(\theta|\mathcal{D}) \approx \frac{1}{N}\sum_i \delta(\theta_i)$

the posterior can be estimated by a mixture of dirac delta distributions. This is similar to a mixture of gaussian distributions, except instead of a gaussian, we have deltas, which are narrow spikes where each element has no variance. each spike is centered at the parameter vector for the corresponding network in the ensemble

$$\int p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t, \theta)p(\theta|\mathcal{D})d\theta \approx \frac{1}{N}\sum_i p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t, \theta_i)$$

so what we do is simply average over our models. so we create a mixture distribution, where each mixture element is the prediction of the corresponding models. Critically, for continuous models, we don't average our means and then average our variance. So we don't output one gaussian with the mean average and variance. We instead average over the distributions, not the means

How to train?

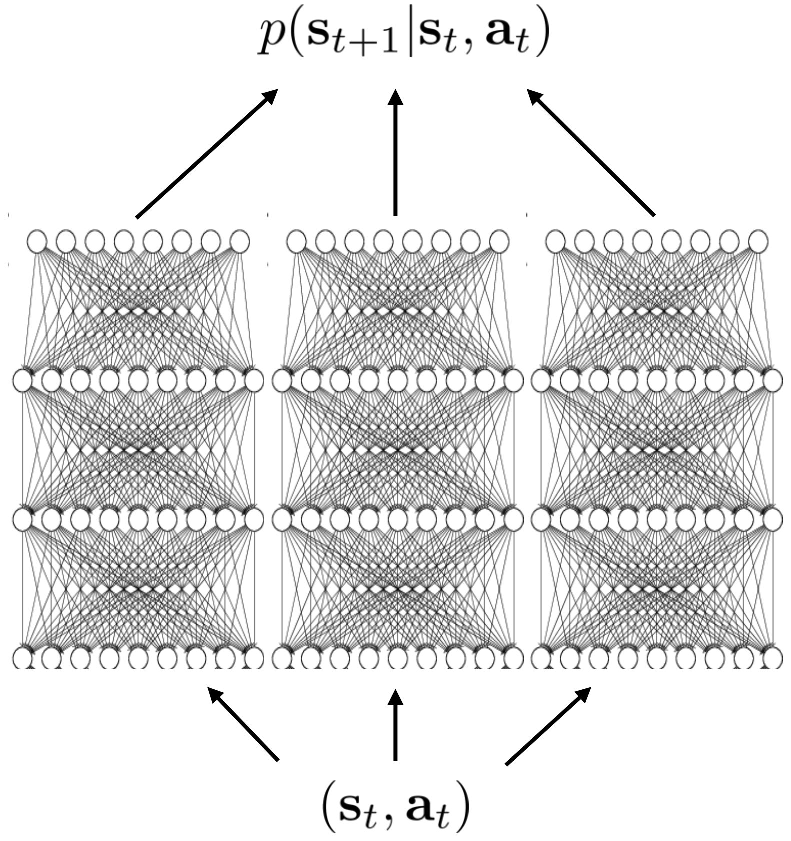
this is the definitions of creating bootstrapped datasets

Main idea: need to generate "independent" datasets to get "independent" models

so each model is trained with a slightly different dataset and this is enough to give a parameter posterior

θ_i is trained on $\mathcal{D}_i$, sampled *with replacement* from $\mathcal{D}$

Bootstrap ensembles in deep learning


$p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$
$(\mathbf{s}_t, \mathbf{a}_t)$

This basically works

Very crude approximation, because the number of models is usually small (< 10)

Resampling with replacement is usually unnecessary, because SGD and random initialization usually makes the models sufficiently independent

While bootstrapping (resampling with replacement) is important for theoretical results, in practice it's actually unnecessary. It turns out that random weight initialization + SGD training will make the models sufficiently independent.

Planning with Uncertainty, Examples

How to plan with uncertainty

Let's say we've trained our uncertainty-aware model, perhaps by using a bootstrap ensemble, and now we'd like to use it in our model-based RL v1.5 algorithm to make decisions.

Before: $J(\mathbf{a}_1, \ldots, \mathbf{a}_H) = \sum_{t=1}^{H} r(\mathbf{s}_t, \mathbf{a}_t)$, where $\mathbf{s}_{t+1} = f(\mathbf{s}_t, \mathbf{a}_t)$

whether you're using random shooting, CEM, this is essentially the problem you're solving

Now: $J(\mathbf{a}_1, \ldots, \mathbf{a}_H) = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{H} r(\mathbf{s}_{t,i}, \mathbf{a}_t)$, where $\mathbf{s}_{t+1,i} = f_i(\mathbf{s}_{t,i}, \mathbf{a}_t)$

now we want to maximize the average reward across all N models

distribution over deterministic models

In general, for candidate action sequence $\mathbf{a}_1, \ldots, \mathbf{a}_H$:

Step 1: sample $\theta \sim p(\theta|\mathcal{D})$ choose one of N models randomly

Step 2: at each time step t, sample $\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t, \theta)$

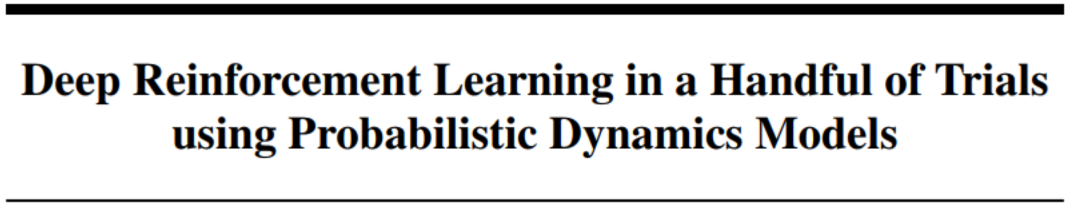
Step 3: calculate $R = \sum_t r(\mathbf{s}_t, \mathbf{a}_t)$

Step 4: repeat steps 1 to 3 and accumulate the average reward

Other options: moment matching, more complex posterior estimation with BNNs, etc.

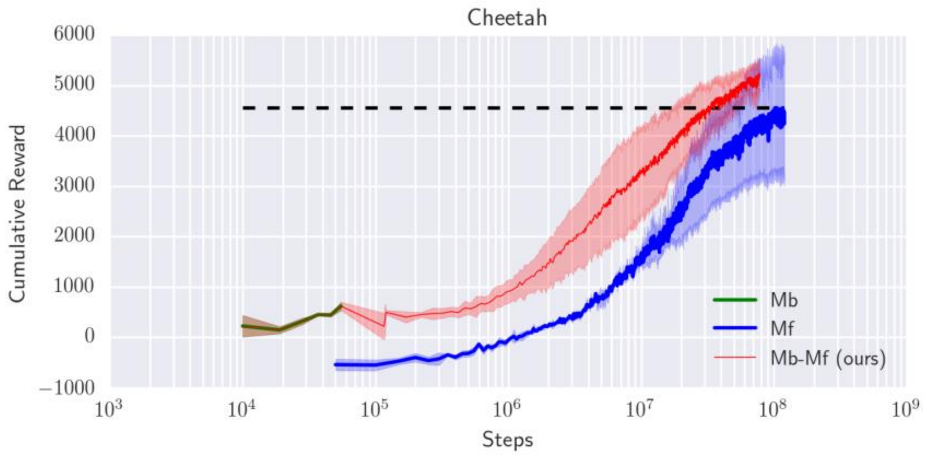
Example: model-based RL with ensembles

so essentially, we can do way better than we did before. In fact, we can do better than the model-free version we did before in only 10 minutes of training

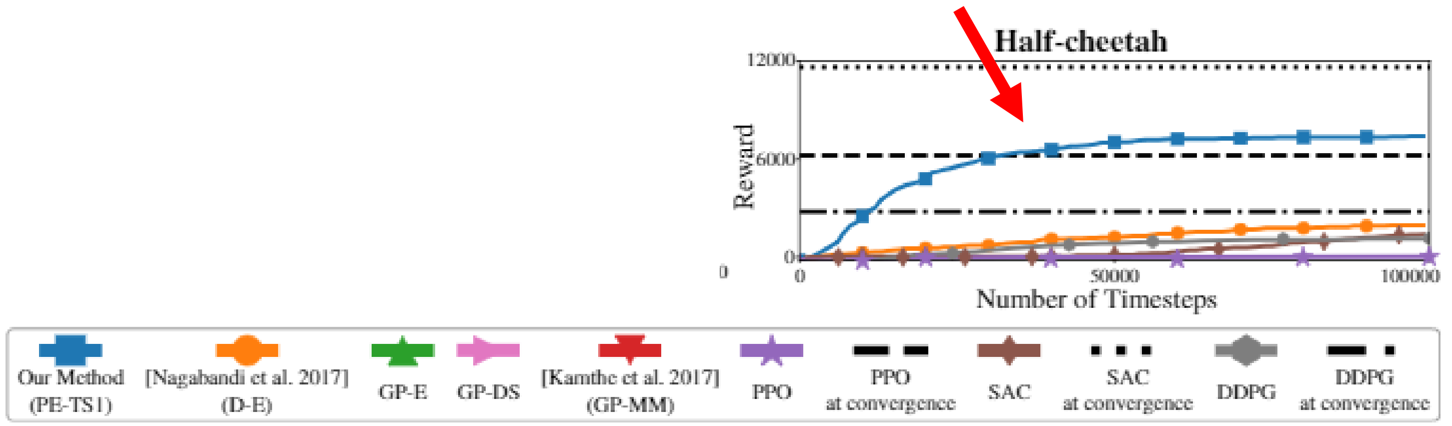


Deep Reinforcement Learning in a Handful of Trials using Probabilistic Dynamics Models

exceeds performance of model-free after 40k steps (about 10 minutes of real time)

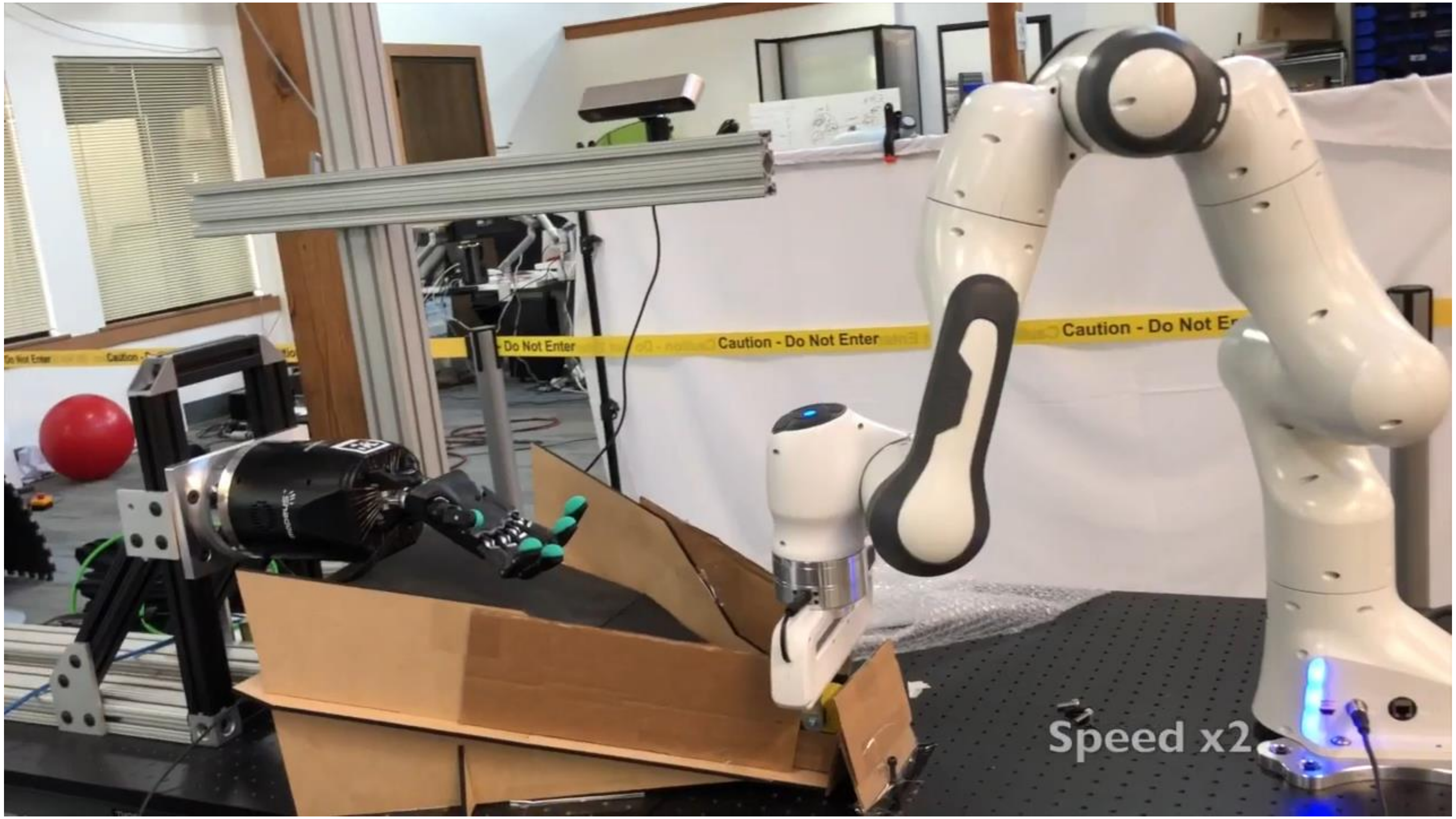


before



after

More recent example: PDDM



Deep Dynamics Models for Learning Dexterous Manipulation. Nagabandi et al. 2019

Further readings

- Deisenroth et al. PILCO: A Model-Based and Data-Efficient Approach to Policy Search.

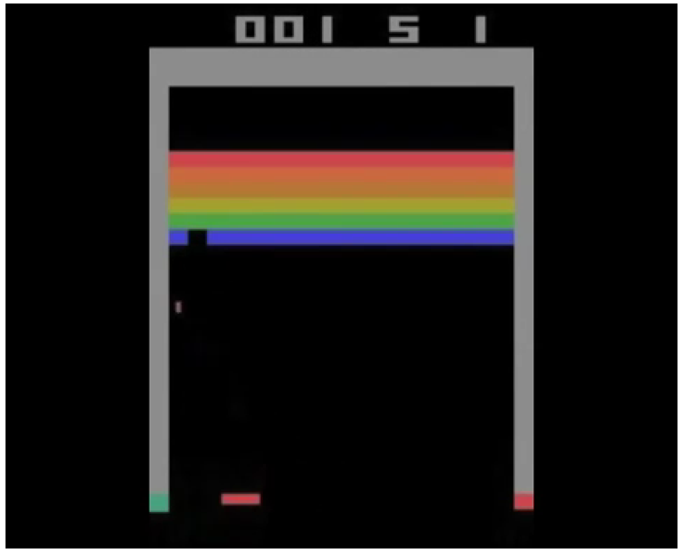
Recent papers:

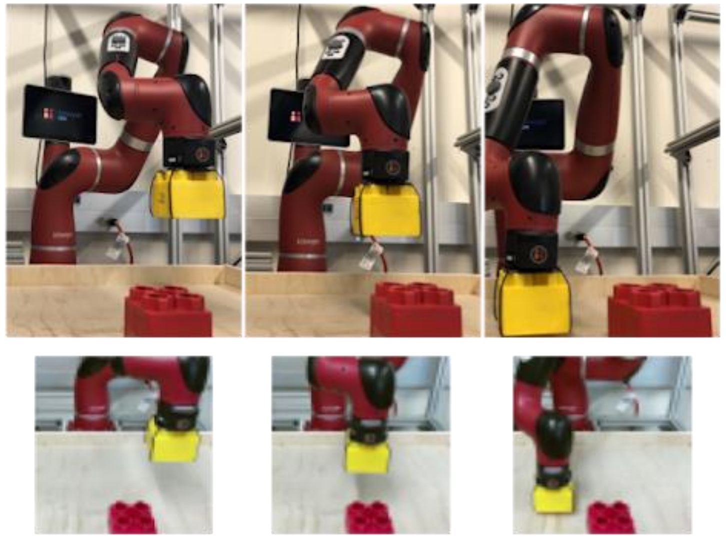
- Nagabandi et al. Neural Network Dynamics for Model-Based Deep Reinforcement Learning with Model-Free Fine-Tuning. model-based RL v1.5
- Chua et al. Deep Reinforcement Learning in a Handful of Trials using Probabilistic Dynamics Models. introduces ensembles
- Feinberg et al. Model-Based Value Expansion for Efficient Model-Free Reinforcement Learning.
- Buckman et al. Sample-Efficient Reinforcement Learning with Stochastic Ensemble Value Expansion.

Model-Based RL with Images

What about complex observations?

if you observe one frame in an Atari game, you won't know how fast a ball is moving and in which direction





$$f(\mathbf{s}_t, \mathbf{a}_t) = \mathbf{s}_{t+1}$$

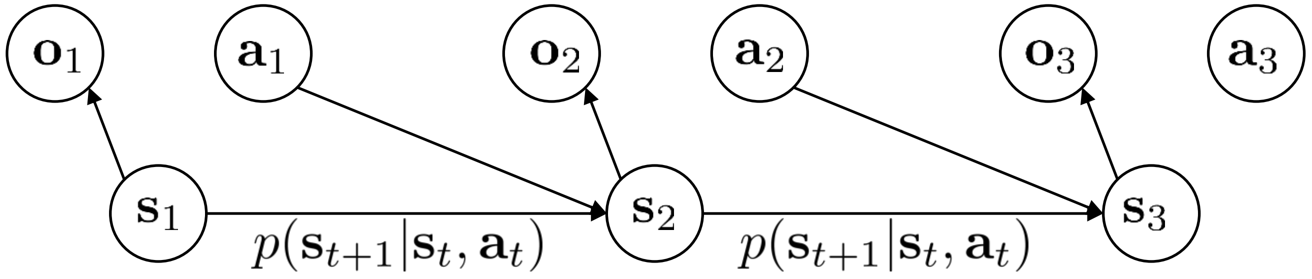
What is hard about this?

- High dimensionality
- Redundancy neighboring pixels are very similar
- Partial observability

high-dimensional but not dynamic

low-dimension but dynamic

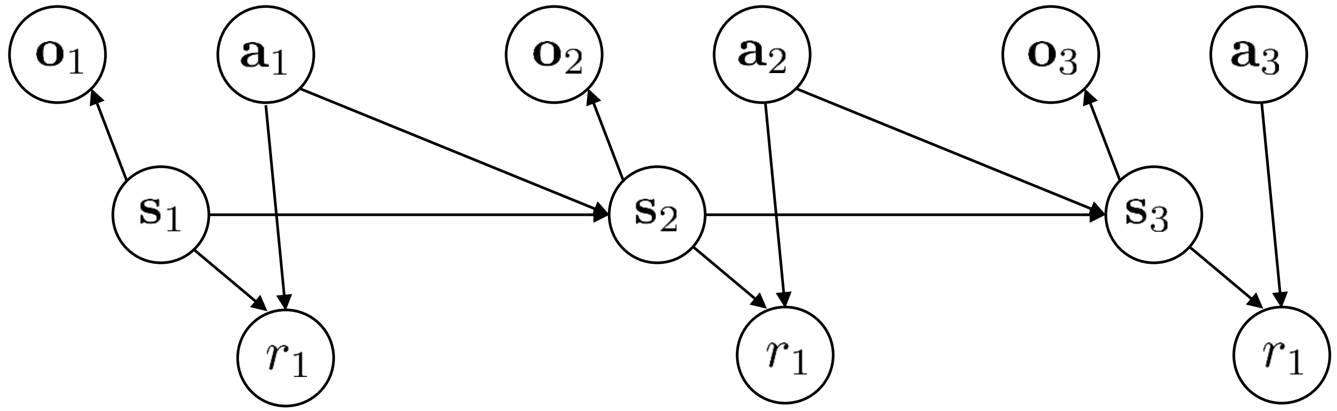
separately learn $p(\mathbf{o}_t|\mathbf{s}_t)$ and $p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$?



$$\begin{tikzcd}
\mathbf{o}_1 & \mathbf{a}_1 \arrow[dr] & \mathbf{o}_2 & \mathbf{a}_2 \arrow[dr] & \mathbf{o}_3 & \mathbf{a}_3 \\
\mathbf{s}_1 \arrow[u] \arrow[rr, "p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)"'] & & \mathbf{s}_2 \arrow[u] \arrow[rr, "p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)"'] & & \mathbf{s}_3 \arrow[u] &
\end{tikzcd}$$

we don't know the state, only the observation state space AKA latent space

State space (latent space) models



how our states map to images

$p(\mathbf{o}_t\|\mathbf{s}_t)$	observation model
$p(\mathbf{s}_{t+1}\|\mathbf{s}_t, \mathbf{a}_t)$	dynamics model
$p(r_t\|\mathbf{s}_t, \mathbf{a}_t)$	reward model

our reward depends on the state and since we don't know what our state is, we don't know how the reward depends on it

How to train?

standard (fully observed) model: $\max_\phi \frac{1}{N}\sum_{i=1}^N\sum_{t=1}^T \log p_\phi(\mathbf{s}_{t+1,i}|\mathbf{s}_{t,i}, \mathbf{a}_{t,i})$

typically we'd use MLE, where for every transition, we'd find the model parameters that maximize the likelihood of s_t+1 given s_t and a_t

we need an algo that can compute the posterior distribution of over states given our images, and then estimate the expected log-likelihood using states sampled from that approximate posterior

latent space model: $\max_\phi \frac{1}{N}\sum_{i=1}^N\sum_{t=1}^T E\left[\log p_\phi(\mathbf{s}_{t+1,i}|\mathbf{s}_{t,i}, \mathbf{a}_{t,i}) + \log p_\phi(\mathbf{o}_{t,i}|\mathbf{s}_{t,i})\right]$

we have to use an expected log-likelihood because we don't know what the states are! The expectation is taken over the distribution of our unknown states in the training trajectories

expectation w.r.t. $(\mathbf{s}_t, \mathbf{s}_{t+1}) \sim p(\mathbf{s}_t, \mathbf{s}_{t+1}|\mathbf{o}_{1:T}, \mathbf{a}_{1:T})$

so how do we actually do this?

Model-based RL with latent space models

$$\max_{\phi} \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} E\left[\log p_\phi(\mathbf{s}_{t+1,i}|\mathbf{s}_{t,i}, \mathbf{a}_{t,i}) + \log p_\phi(\mathbf{o}_{t,i}|\mathbf{s}_{t,i})\right]$$

expectation w.r.t. $(\mathbf{s}_t, \mathbf{s}_{t+1}) \sim p(\mathbf{s}_t, \mathbf{s}_{t+1}|\mathbf{o}_{1:T}, \mathbf{a}_{1:T})$

learn *approximate* posterior $q_\psi(\mathbf{s}_t|\mathbf{o}_{1:t}, \mathbf{a}_{1:t})$ "encoder"

this will be another NN that gives us a distribution over s-T given the observations and actions we've seen thus far, from timesteps 1 to t

many other choices for approximate posterior:

$q_\psi(\mathbf{s}_t, \mathbf{s}_{t+1}|\mathbf{o}_{1:T}, \mathbf{a}_{1:T})$ full smoothing posterior

+ most accurate

- most complicated

$\boxed{q_\psi(\mathbf{s}_t|\mathbf{o}_t)}$ just tries to guess the current state based on the observation single-step encoder

+ simplest

- least accurate

we'll talk about this one for now

in general, you want a better approximate posterior for environments that are more partially observed. So if you think the state can pretty much be guessed by the observation, a single-step encoder might be okay

We will discuss variational inference in more detail next week!

Model-based RL with latent space models

$$\max_{\phi} \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} E\left[\log p_\phi(\mathbf{s}_{t+1,i}|\mathbf{s}_{t,i}, \mathbf{a}_{t,i}) + \log p_\phi(\mathbf{o}_{t,i}|\mathbf{s}_{t,i})\right]$$

if we were to really do this right, then for every timestep, we would sample s_t from q(s_t | o_t) and s_t+1 from q(s_t+1, o_t+1). we would then use those samples to maximize the value inside the expectation.

expectation w.r.t. $\mathbf{s}_t \sim q_\psi(\mathbf{s}_t|\mathbf{o}_t), \mathbf{s}_{t+1} \sim q_\psi(\mathbf{s}_{t+1}|\mathbf{o}_{t+1})$

$q_\psi(\mathbf{s}_t|\mathbf{o}_t)$

simple special case: $q(\mathbf{s}_t|\mathbf{o}_t)$ is *deterministic*

if you believe your problem is almost fully observed, you can instead use a deterministic encoder. So instead of outputting a distribution of s_t given o_t, you would just output a single s_t for our current o_t.

stochastic case requires variational inference (next week)

$q_\psi(\mathbf{s}_t|\mathbf{o}_t) = \delta(\mathbf{s}_t = g_\psi(\mathbf{o}_t)) \Rightarrow \mathbf{s}_t = g_\psi(\mathbf{o}_t)$ deterministic encoder

the deterministic case can be thought of as a delta function that's centered at some deterministic encoding g(o_t). That means that s_t = g(o_t). Using a deterministic encoder, we can substitute this in everywhere we see s_t in our objective function and we can remove the expectation

$$\max_{\phi,\psi} \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \log p_\phi(g_\psi(\mathbf{o}_{t+1,i})|g_\psi(\mathbf{o}_{t,i}), \mathbf{a}_{t,i}) + \log p_\phi(\mathbf{o}_{t,i}|g_\psi(\mathbf{o}_{t,i}))$$

if the dynamics is stochastic you want to use the reparameterization trick to make this possible to solve with gradient descent

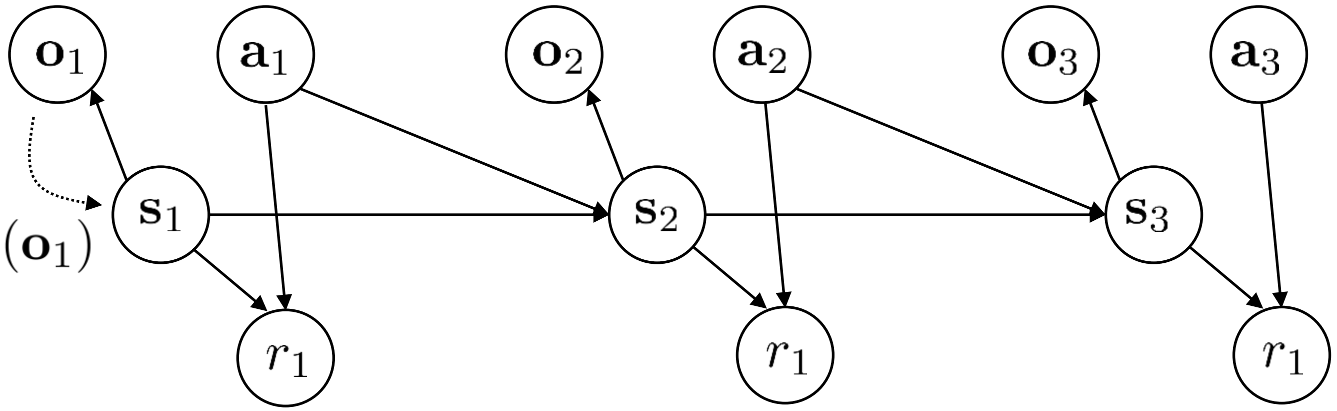
Everything is differentiable, can train with backprop

Summary so far: If you want to learn stochastic state-space models, you need to use an expected log-likelihood instead of a standard log-likelihood, where the expectation is taken w.r.t. an encoder, which represents the posterior. There are many ways to approximate the posterior, but the simplest is to use an encoder from observations to states, and make it a deterministic encoder in which case the expectation goes away. You can then subsequently substitute the encoded observation in place of states in your dynamics and observation model objectives. If we had a reward model, we'd add that in too.

Model-based RL with latent space models

we can think of g(o) as an edge that maps obs to states

$\mathbf{s}_1 = g_\psi(\mathbf{o}_1)$

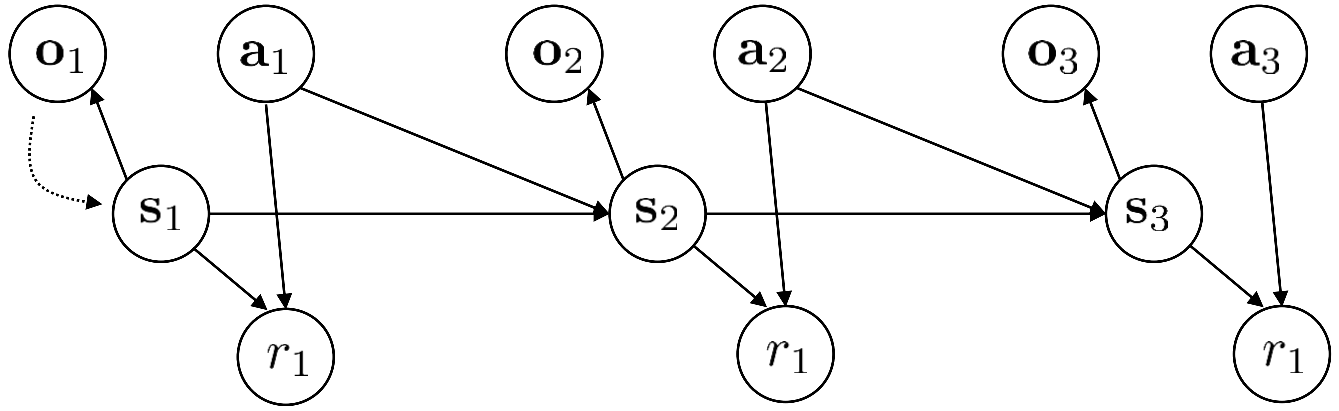


$$\max_{\phi,\psi} \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T} \log p_\phi(g_\psi(\mathbf{o}_{t+1,i})|g_\psi(\mathbf{o}_{t,i}),\mathbf{a}_{t,i}) + \log p_\phi(\mathbf{o}_{t,i}|g_\psi(\mathbf{o}_{t,i})) + \log p_\phi(r_{t,i}|g_\psi(\mathbf{o}_{t,i}))$$

latent space dynamics image reconstruction reward model

Many practical methods use a stochastic encoder to model uncertainty

Model-based RL with latent space models



you can substitute this into the model-based RL v1.5

model-based reinforcement learning with latent state:

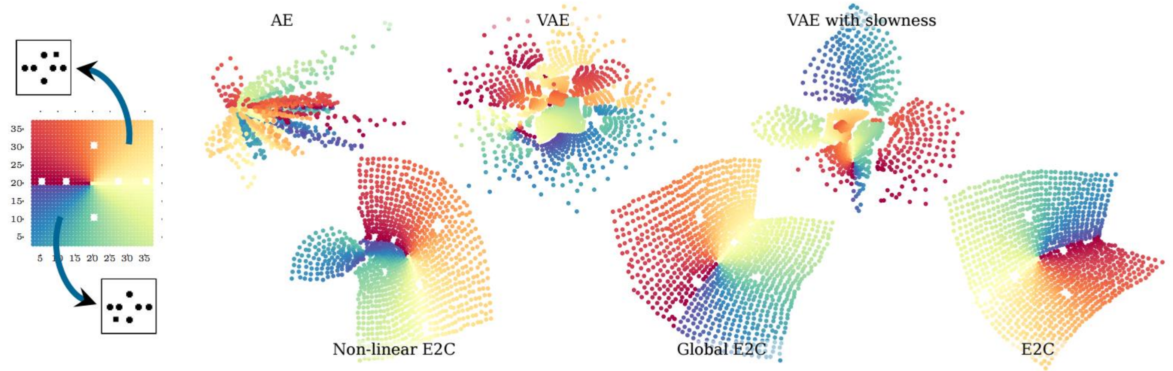
1. run base policy $\pi_0(\mathbf{a}_t|\mathbf{o}_t)$ (e.g., random policy) to collect $\mathcal{D} = \{(\mathbf{o}, \mathbf{a}, \mathbf{o}')_i\}$
2. learn $p_\phi(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$, $p_\phi(r_t|\mathbf{s}_t)$, $p(\mathbf{o}_t|\mathbf{s}_t)$, $g_\psi(\mathbf{o}_t)$ ← learn dynamics, reward model, observation model, and encoder
3. plan through the model to choose actions
4. execute the first planned action, observe resulting $\mathbf{o}'$ (MPC)
5. append $(\mathbf{o}, \mathbf{a}, \mathbf{o}')$ to dataset $\mathcal{D}$

every N steps

Embed to Control: A Locally Linear Latent Dynamics Model for Control from Raw Images

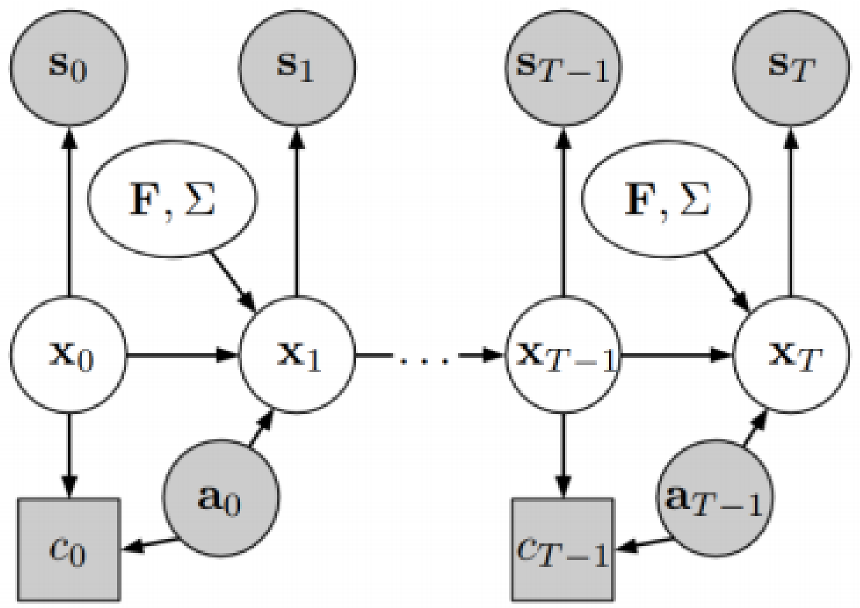
Manuel Watter* **Jost Tobias Springenberg*** **Joschka Boedecker**
University of Freiburg, Germany
{watterm, springj, jboedeck}@cs.uni-freiburg.de

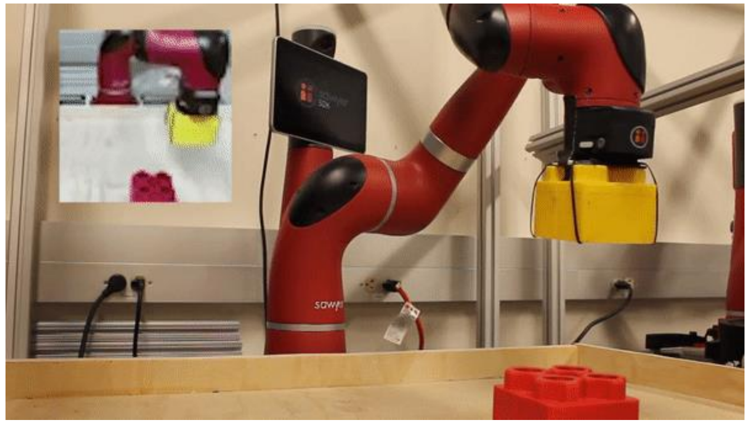
Martin Riedmiller
Google DeepMind
London, UK
riedmiller@google.com



Swing-up with the E2C algorithm

SOLAR: Deep Structured Latent Representations for Model-Based Reinforcement Learning





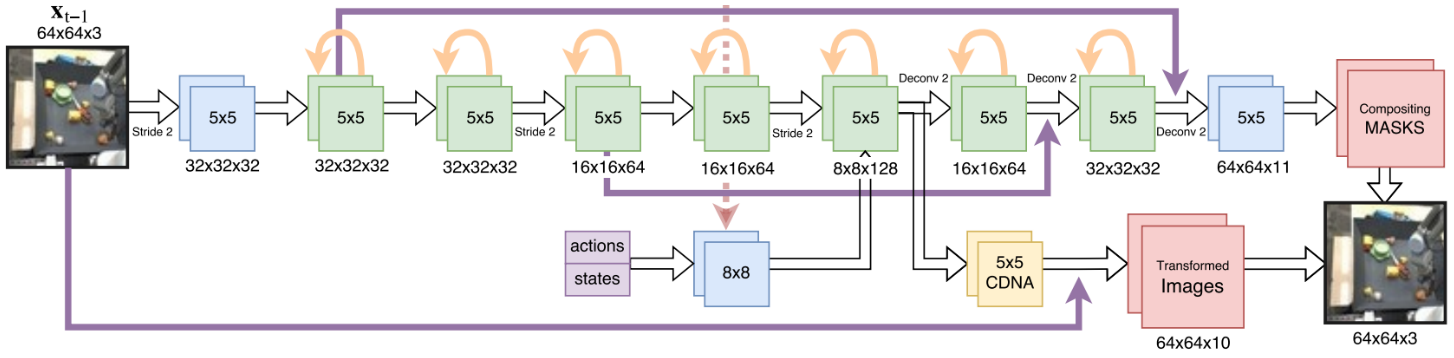
Learn directly in observation space

if we have partial-observability, we probably need to use a recurrent model. so we make o_t+1 depend on old observations o_t. But as long as we do this, we can do a pretty good job of modeling dynamics directly from images

~~**Key idea:** learn embedding $g(\mathbf{o}_t) = \mathbf{s}_t$~~

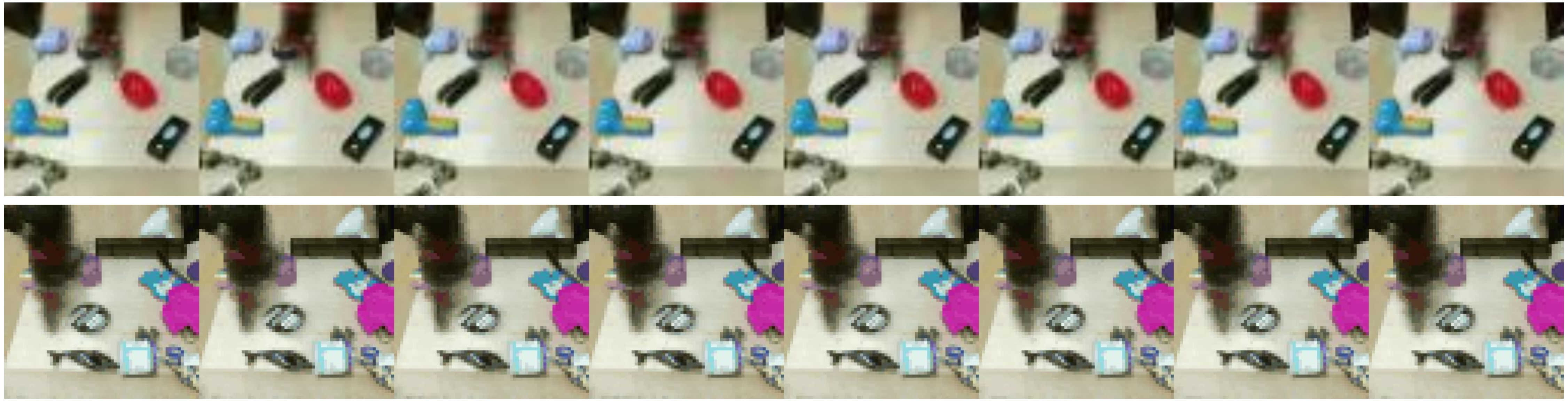
directly learn $p(\mathbf{o}_{t+1}|\mathbf{o}_t, \mathbf{a}_t)$

these methods work well in complex settings where learning a compact latent space is very difficult. So if you had dozens of objects in a scene, it's not obvious on how to construct a compact state space for them. So using the image-space directly can perform well

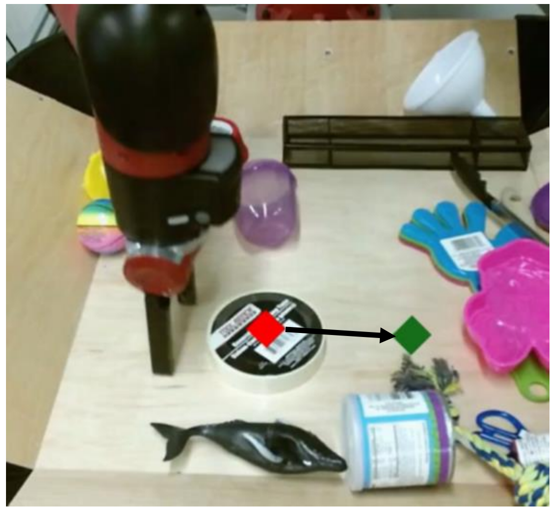


Finn, L. **Deep Visual Foresight for Planning Robot Motion.** ICRA 2017.

Ebert, Finn, Lee, L. **Self-Supervised Visual Planning with Temporal Skip Connections.** CoRL 2017.

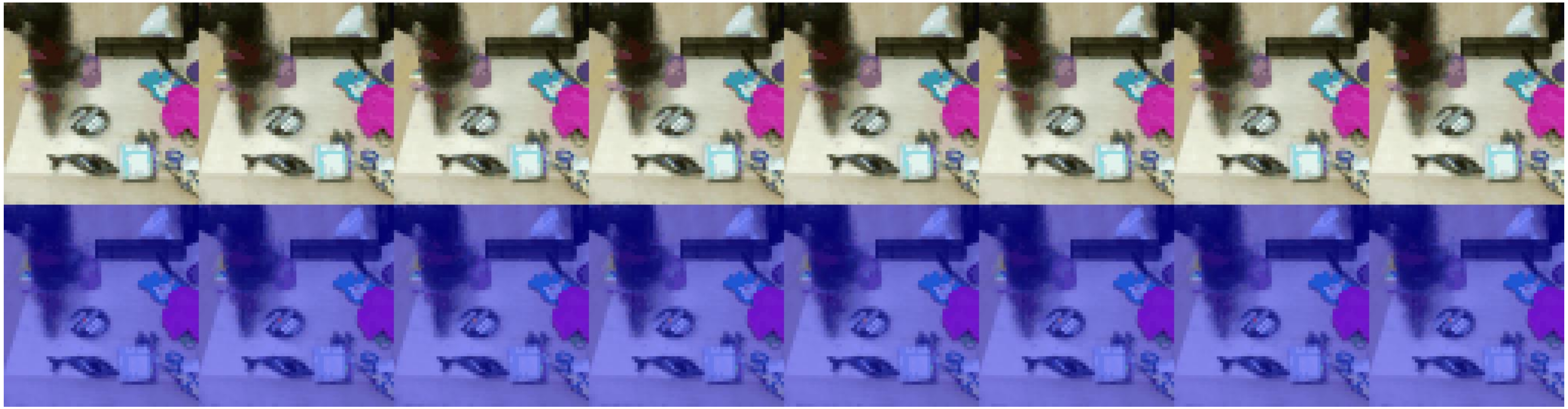


Use predictions to complete tasks



Designated Pixel ◆

Goal Pixel ◆



Task execution