

Model-Based Reinforcement Learning

Put the model back — learned — and let the two lineages reunite

Jinkyoo Park · KAIST

A model given (7, 9), a model deleted (8, 10), a model *learned* — and shared

Where we are — both lineages, each missing its model

PREVIOUS STEP	THIS CHAPTER'S QUESTION
Lectures 8 and 10 learned values and policies from experience.	Now learn dynamics and use predicted transitions to help control.

Model error can affect both plans and learned policies. Follow the core model-use loop before differentiating an optimizer.

Re-plan from a measured temperature and explain how each learner uses imagined data.

Learning route — use a learned model, then check what it changes

Bring: Bellman updates, feedback, SAC and the idea of a latent state.

FIRST PASS	WHAT TO DO
Follow the idea	Fit a model → MPC/PETS → Dyna/MBPO → Dreamer
Work without the solution	Re-plan from a measured temperature and explain how each learner uses imagined data.
Return later	KKT-based differentiable MPC, PILCO and GPS/PLATO are after the main route.

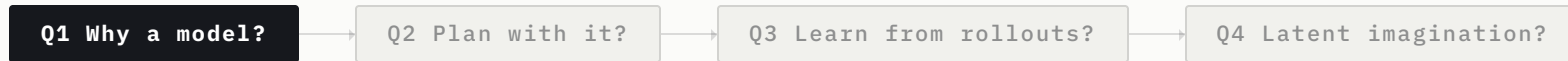
For the temperature thread: **predict** → **calculate** → **reveal and check** → **change one condition**. Complete the core calculation before reading the research extensions.

— ACT 1

The spectrum of model use

Q1. "Model-based" is not the opposite of "model-free". It is the far end of a dial.

Two ways to come by a model



OFTEN WE SIMPLY HAVE ONE

- **Games** — Atari, chess, Go: the rules *are* the transition function.
- **Easily modelled systems** — a car following a bicycle model.
- **Simulated environments** — the simulator is the dynamics.

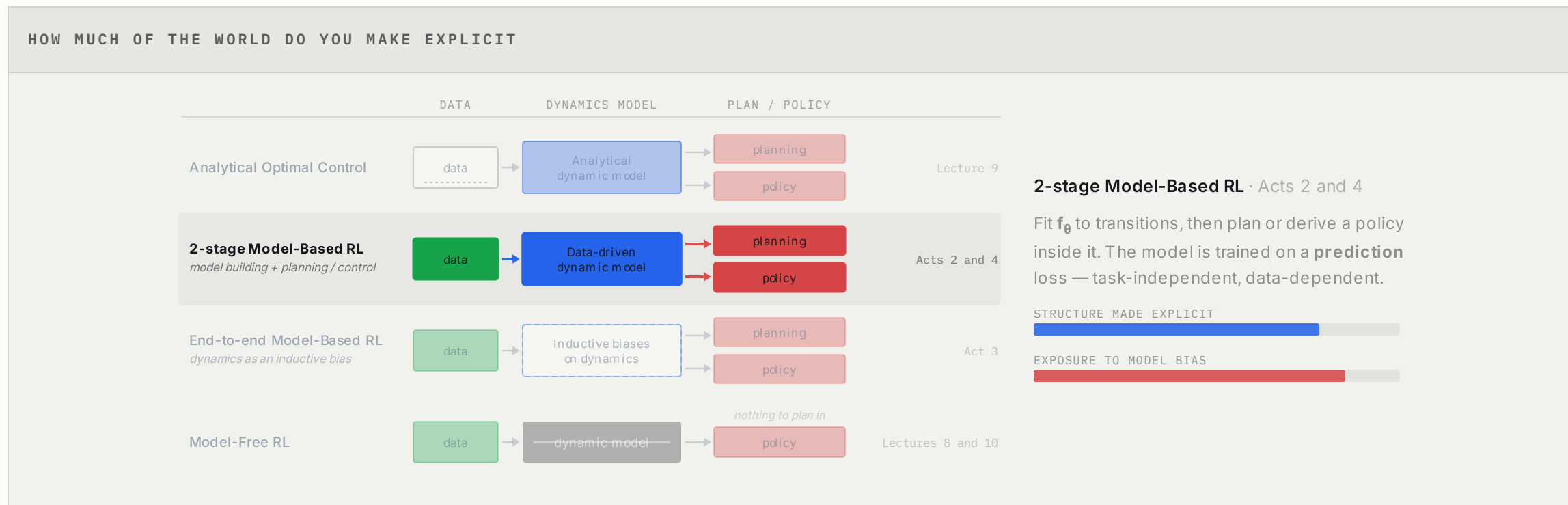
OFTEN WE CAN LEARN ONE

- **System identification** — fit the few unknown parameters of a **known** structure. Excite the system, measure the response, fit.
- **Learning** — fit a **general-purpose** model to observed transitions $\{(s, a, s')\}$, with architecture and regularization supplying inductive assumptions.

Either way, the payoff has two faces, and they are Acts 2 and 4 of this lecture:

Knowing the dynamics helps you **plan a sequence of actions**, and it helps you **derive a policy**.

The ladder of model use



Four ways to get from data to a decision, drawn as the source deck draws them. Walk down and the model is made less and less explicit: an analytical model needing no data at all; a model *fitted* then planned in; dynamics baked into the architecture as an inductive bias; and finally the model deleted, data wired straight to a policy. Note the right-hand column: only the model-free row cannot produce a **plan**, because there is nothing to plan *in*.

The same ladder, as a ledger

	MODEL-FREE RL	MODEL-BASED RL	DIFFERENTIABLE MPC
explicit model?	no	yes	yes
trained object	policy $\pi(\cdot)$	model $f(\cdot)$	model $f(\cdot)$, and cost $c(\cdot)$
loss	task loss	model-prediction loss	task loss
execution	policy	planning	planning

Read left to right and more of the world's structure is made explicit, buying **sample efficiency** and generalisation. Read right to left and there is less to get wrong, dodging **model bias**.

A learned model is reusable but **wrong**; a model-free policy is honest but **hungry**.

The price, named — model errors can change the plan

The naive plan is to fit a model, plan in it, and execute. It works for exceptionally simple cases — those in which the fitted model generalises over the *whole* state space.

WHAT GOES WRONG

A model is fitted where the base policy went. Planning then evaluates it where the *plan* goes, which is somewhere else — and a planner can prefer actions whose predicted outcomes are overly optimistic.

$$p_{\pi_f}(s_t) \neq p_{\pi_0}(s_t)$$

WHERE YOU HAVE MET THIS BEFORE

Recall Lecture 5: optimization can select optimistic model errors. A planner can encounter a similar risk when searching sequences of actions.

And it gets *worse* with capacity: extra capacity can worsen off-distribution overfitting without adequate data or regularization; it can also reduce misspecification.

So the optimum found in a learned model is only **optimum from the model's point of view**. The planning examples show ways to reduce or manage that risk.

Check — what a model buys

Why might a learned model reduce the number of real interactions needed for learning?

- ☒ A It can support planning and synthetic updates that reuse the information in real experience
- ☐ B It guarantees that the final policy is better than every model-free policy
- ☐ C It removes the need for an informative state representation
- ☐ D Synthetic transitions are new measurements of the real environment

Imagined experience can be cheap to generate, but it inherits model error. Sample efficiency is a possible benefit to test under a fixed real-data budget, not a guarantee. Training cost, model accuracy and the task all matter.

— ACT 2

Planning with a learned model

Q2. Fit a model; plan in it. Four versions, each named by the failure of the last.

Version 1—fit, then plan once

Q1 Why a model?

Q2 Plan with it?

Q3 Learn from rollouts?

Q4 Latent imagination?

MODEL BUILDING + OPEN-LOOP PLANNING

1. Run a base policy $\pi_0(a_t | s_t)$ — e.g. random — and collect $\mathcal{D} = \{(s, a, s')_i\}$ (*experiencing*)
2. Learn $f(s, a)$ by minimising $\sum_i \|f(s_i, a_i) - s'_i\|^2$ (*model fitting*)
3. Plan through $f(s, a)$ to choose actions (*model-based planning*)
4. Execute those actions

YES — THIS IS CLASSICAL SYSTEM IDENTIFICATION

Excite, measure, fit a **predefined** model. Particularly effective when physics gives the structure and only a few parameters are free: fit Newton's law once and it describes any rigid body.

Linear sysID estimates A, B in $\dot{x} = Ax + Bu$ — and **once the linear system is identified, all of Lecture 9 applies**: Riccati, K , the lot.

NO — THE DISTRIBUTION MOVES UNDER YOU

The plan is executed open loop. The states it visits are not the states the model was fitted on, and there is no correction channel.

$$p_{\pi_f}(s_t) \neq p_{\pi_0}(s_t)$$

What system identification actually offers

Before reaching for a network, note how much structure the control tradition already knows how to fit — and that every one of these was built to be *planned in*.

METHOD	MODEL	WHAT IT BUYS
Linear sysID	$\dot{x} = Ax + Bu$	Lecture 9 applies verbatim
Model reduction / DMD	$\dot{a} = \tilde{A}a + \tilde{B}u$	a low-dimensional state — fast prediction, fast control
Piecewise linear	local A_k, B_k	iLQR's local model, made explicit
Koopman operator	linear in infinite-dimensional measurements	a nonlinear system, solved by linear theory
Deep / GP models	MLP, RNN, LSTM, Gaussian process	expressive — and the reason distribution mismatch bites

These are alternative structures, not a strict ranking of expressivity or planning difficulty. The advanced differentiable-control section studies models whose structure also helps optimization.

Version 2 — put the loop back

Append what actually happened, refit, repeat. If the model is wrong where the plan goes, then [go there and collect data](#):

ITERATIVE MODEL BUILDING

5. Append $\{(s, a, s')_j\}$ to $\mathcal{D}$, and return to step 2 (*collecting data*)

This is data aggregation: add data from the controller's current visits so the training distribution better covers the states planning needs. It helps — and it still executes a whole plan before looking.

So: what if we make a mistake [in the middle of the plan](#)?

Version 3 — re-plan every step, and MPC is born

At each step, optimise a short horizon of actions, **execute only the first**, observe, re-plan:

$$\min_{u_{0:H-1}} \sum_{t=0}^{H-1} C_t(x_t, u_t) + C_H(x_H), \quad x_{t+1} = f_\theta(x_t, u_t), \quad x_0 = x_{\text{measured}}$$

This is **Lecture 9's optimal control, run on a learned model** instead of a given one. Nothing in the solver changes; only the constraint is now fitted.

REPLANNING HELPS WITH MODEL ERRORS

- the more you re-plan, **the less perfect each individual plan needs to be**;
- you can therefore use **shorter horizons**, where the model is still trustworthy;
- and the inner optimiser can be crude — even random shooting often works.

Plan twice — first from a prediction, then from a measurement

Use $\hat{x}_{t+1} = \hat{x}_t + u_t$, start at $x_0 = 2$, and compare two-step plans with cost $u_0^2 + u_1^2 + x_2^2$.

PLAN	PREDICTED FINAL STATE	TOTAL PREDICTED COST
(0, 0)	2	4
(−1, 0)	1	2
(−1, −1)	0	2

Among these candidates, choose $(-1, 0)$ and execute **only** $u_0 = -1$. Suppose the measured next state is **1.4**, not the predicted 1.

The next optimization starts from **1.4**. For a final one-step cost $u^2 + (1.4 + u)^2$, it chooses $u = -0.7$. MPC corrects its starting state; it does not magically correct every model error.

Temperature thread — re-plan after a model error

shared teaching example · predict before revealing the calculation

The planning model is $x' = x + u$, with cost $(x')^2 + u^2$. Starting at $x = -2$, it chooses $u = 1$ and predicts $x' = -1$. The real next measurement is -1.5 .

Predict: If the model has not yet been refitted, should the next plan start from -1 or from -1.5 ?

Calculate and check. Start from the measurement -1.5 . With the current model, one-step replanning gives $u = -x/2 = \mathbf{0.75}$; the stale open-loop continuation was 0.5 .

Replanning corrects the starting state. Learning the model corrects the predicted response; those are different updates.

Try it — the response model is refitted

work independently · reveal only after writing an answer

At the measured state $x = -1.5$, change the learned dynamics to $x' = x + gu$ with $g = 0.5$. For cost $(x + gu)^2 + u^2$, solve $g(x + gu) + u = 0$ for u .

Check your answer. $u^* = -gx/(1 + g^2) = -0.5(-1.5)/1.25 = \mathbf{0.6}$. Under this updated model the cost is $(-1.2)^2 + 0.6^2 = \mathbf{1.8}$.

A fresh state estimate and a changed dynamics model can both change the action. MBPO and Dreamer also depend on the model used to generate predictions.

What does “plan through the model” actually compute?

original model-based RL PDF pp. 18–30 · the inner search in MPC

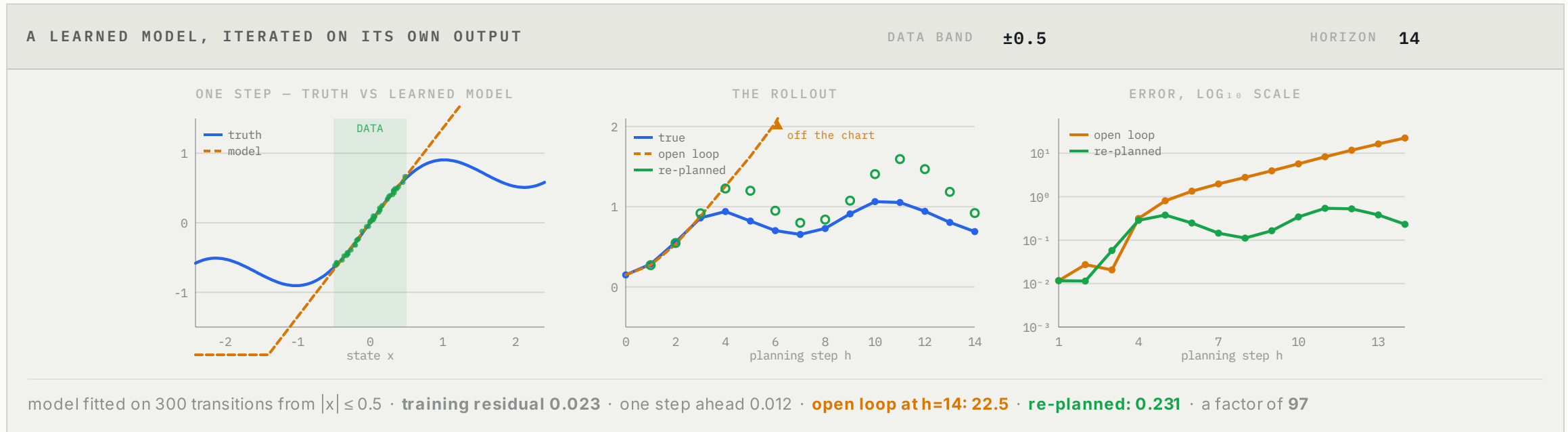
At the measured state s_t , compare candidate action sequences using the learned dynamics and the same objective:

$$\hat{J}(a_{t:t+H-1}) = \sum_{j=0}^{H-1} \gamma^j \hat{r}(\hat{s}_{t+j}, a_{t+j}), \quad \hat{s}_{t+j+1} = \hat{f}(\hat{s}_{t+j}, a_{t+j}).$$

SEARCH OPTION	CALCULATION
Random shooting	sample several action sequences, simulate each, keep the highest predicted return
Cross-entropy method (CEM)	repeatedly refit a sampling distribution to the best candidate sequences
Gradient-based planning	differentiate predicted return through a differentiable dynamics model

MPC executes **only the first action of the chosen sequence**, measures the new state, and searches again. This feedback rule matters even when the inner search is approximate.

What a rollout does to a model's error



A one-dimensional system, a base policy that explored a band around the origin, and a model fitted inside it. Left: the model is **excellent where the data is** and hopeless outside.

Middle: rolled out open loop, the model is composed with *its own output*, so it walks out of the band and never comes back. Right, on a log axis: the open-loop error climbs a decade every few steps, while **re-measuring the state each step holds it flat**.

Why rollout error grows — count the propagation

Let the one-step model error be at most ϵ and let true dynamics amplify state differences by at most L . For the same actions and $e_0 = 0$,

$$e_{t+1} \leq L e_t + \epsilon, \quad e_H \leq \epsilon \sum_{j=0}^{H-1} L^j.$$

For $\epsilon = 0.1$:

AMPLIFICATION	BOUND AFTER 1 STEP	AFTER 3 STEPS	AFTER 5 STEPS
$L = 1$	0.1	0.3	0.5
$L = 2$	0.1	0.7	3.1

Error can grow linearly, geometrically, or remain bounded when $L < 1$. **Exponential growth is not universal.** The bound assumes the error and sensitivity limits hold along the rollout.

Version 3+ — plan with the model's uncertainty

Version 3 still acts on the model's *mean*: it takes actions believed good in expectation, which provides no explicit incentive to seek information. It may still visit unfamiliar states.

Split the error the way Lecture 0 split it, and the way Lecture 2 wrote it:

$$\mathbb{E}\|Y - \hat{f}(X)\|^2 = \underbrace{\mathbb{E}\|Y - f(X)\|^2}_{\sigma_s^2 = \text{systemic noise}} + \underbrace{\mathbb{E}\|f(X) - \hat{f}(X)\|^2}_{\sigma_m^2 = \text{model uncertainty}}$$

This decomposition assumes $f(X) = \mathbb{E}[Y \mid X]$ and zero-mean conditional noise for the evaluation data. The second term includes model error; sparse data can increase it, but no universal equality links error to data density.

Three ways to get σ_m , all of them borrowed

ENSEMBLES

$$p(\theta \mid \mathcal{D}) \approx \frac{1}{N} \sum_i \delta_{\theta_i}(\theta),$$

$$p(s' \mid s, a, \mathcal{D}) = \int p(s' \mid s, a, \theta) p(\theta \mid \mathcal{D}) d\theta$$

$$\approx \frac{1}{N} \sum_i p(s' \mid s, a, \theta_i).$$

Train each θ_i on $\mathcal{D}_i$, sampled with replacement. [Lecture 5's ensemble-alarm](#), in dynamics.

GAUSSIAN PROCESSES

A GP provides a posterior distribution over transitions. Tracking changing dynamics requires an explicit time model, forgetting scheme, or other adaptation; Bayesian updating alone does not establish nonstationarity or robustness. [Lecture 4's posterior](#), used as f .

The third is a **Bayesian neural network** — put a distribution on the weights rather than a point estimate (*Blundell et al., 2015*). All three answer the same question Lecture 2 asked about a coin: **do not carry a number, carry a belief**. **PETS** combines neural-network ensembles with probabilistic outputs — a probabilistic *ensemble*, whose spread far from data is epistemic and whose per-member variance is aleatoric — and learns four continuous-control tasks in **under 100 000 steps, or 100 trials**, where PPO, SAC and DDPG need one to two orders of magnitude more. (*Chua et al., 2018*)

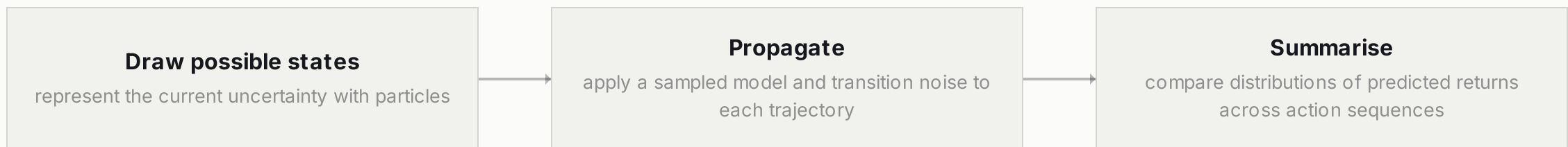
Uncertain inputs require a distribution of rollouts

original model-based RL PDF pp. 38–41 · uncertainty propagates through the model

After one prediction, a future state is uncertain. Feeding only its mean into a nonlinear model generally loses information:

$$\mathbb{E}[f(X)] \neq f(\mathbb{E}[X]).$$

For an illustrative X equally likely to be -1 or 1 and $f(x) = x^2$, the true next-state mean is **1**, while the plug-in mean prediction is $f(0) = \mathbf{0}$.



The source uses Monte Carlo propagation and Gaussian moment approximations. An ensemble represents alternative models; transition noise represents variability within a model. Neither a particle count nor ensemble agreement alone certifies calibration.

Check — why imagined rollouts stay short

Why can long rollouts from a learned model become unreliable?

- ☐ A Longer rollouts are too expensive to compute
- ☒ B Because one-step model error compounds: each imagined step feeds the next, so trajectory errors can accumulate and amplify, especially outside the data
- ☐ C Because the discount factor makes distant steps irrelevant anyway
- ☐ D Because the replay buffer cannot hold long trajectories

A model accurate to within ϵ per step is not accurate to within ϵ over fifty steps — the errors feed forward and the imagined state leaves the region the model was ever fitted on. Short rollouts branched from **real** states keep the model working where it is trustworthy. This shares a concern about model trust with Lecture 5 and Lecture 10, but uses a different mathematical intervention.

— ACT 3

Learn from short simulated experience

Q3. How can a learned model supply useful training transitions?

Route 3 — imagine the data

The simplest reunion: generate transitions with the learned model and use them in a compatible value or policy learner. The update formula can stay the same even though the data quality changes.

DYNA — ONLINE Q-LEARNING THAT PERFORMS MODEL-FREE RL WITH A MODEL

1. From s , pick a by an exploration policy; observe (s, a, s', r)
2. Update the model $\hat{p}(s' \mid s, a)$ and $\hat{r}(s, a)$
3. $Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$ *(from real experience)*
4. Repeat k times: sample (s, a) from the buffer, **simulate** $s' \sim \hat{p}$, $r = \hat{r}$, and apply the same update *(from imagined experience)*

Line 3 is Lecture 8, untouched. Line 4 is a sampled planning backup on an estimated model, extending Lecture 7's planning idea. **The two lineages meet inside a single loop, four lines apart.** (Sutton, 1990)

Only short rollouts are needed — as few as one step — and the algorithm still sees diverse states, because the imagined transitions start from every state in the buffer. MBPO develops this idea with **short policy rollouts branched from real states** and an off-policy learner. MVE instead uses model rollouts in value-target construction. The length and the use of the imagined sequence both matter.

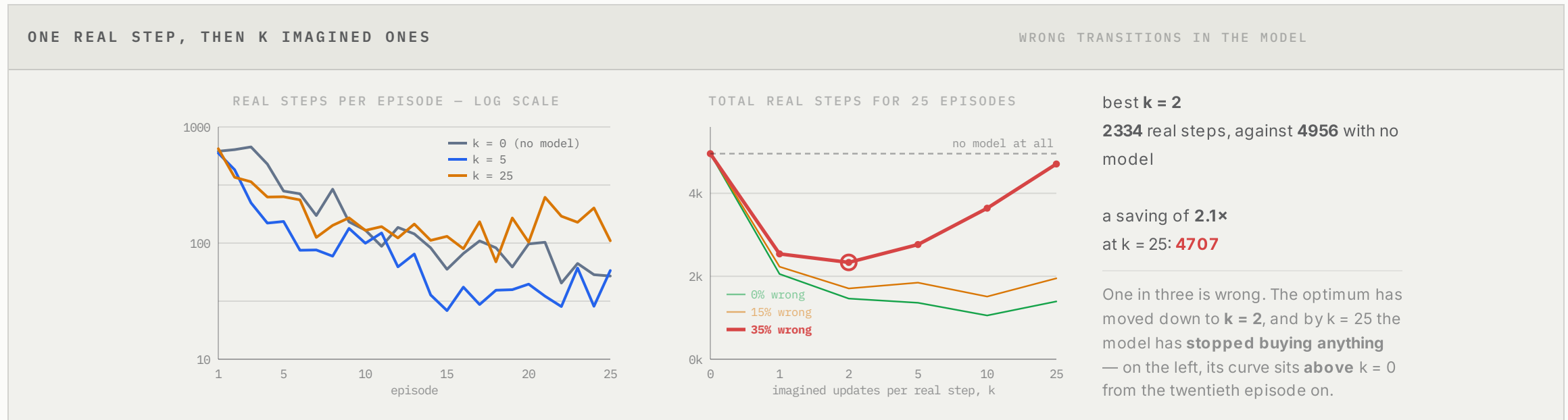
Dyna — the same update from two different sources

Let $Q(s, a) = 2$, $\alpha = 0.2$, $\gamma = 0.9$, and next maximum value 5.

SOURCE	REWARD	TARGET	UPDATED ESTIMATE FROM 2
real transition	1	$1 + 0.9(5) = 5.5$	2.70
correct imagined transition	1	5.5	2.70
optimistic model error	3	$3 + 0.9(5) = 7.5$	3.10

The update cannot tell whether its label came from reality or a model. Extra planning saves real interaction **only insofar as the imagined information is useful**. It can also repeat a mistake many times.

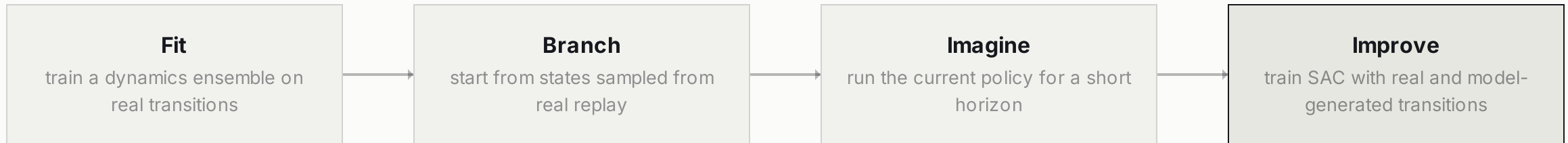
Real steps against imagined steps



Sutton's maze, learned three ways. Raise k — the number of imagined updates per real step — and the number of real environment steps needed collapses. Then corrupt a fraction of the learned model's transitions and raise k again: past a point the imagined updates **stop buying anything at all**, and the heavily-planning agent ends up behind the one with no model. The gain and its price, on one screen.

MBPO — branch short rollouts from real data

Janner et al. · NeurIPS 2019 · Dyna's idea with a continuous-control learner



Keep collecting real experience to update the model. The synthetic buffer expands the learner's training opportunities; it does not add new measurements of the environment.

Count the trade-off: 100 real starting states produce 100 synthetic transitions at horizon 1, or 500 at horizon 5. The second option uses five times as many predictions, with more opportunities for compounding error. These are illustrative budgets, not recommended settings for every task.

The important change is **short, branched imagination**, not simply more simulation. Choose model use in light of its error and policy shift. [When to Trust Your Model](#)

Check — accurate is not the same as useful

Two learned models have the same average one-step prediction error. Why can their policies perform differently?

- ☐ A Equal average prediction error guarantees equal control quality
- ☐ B Only training time can explain the difference
- ☒ C Their errors can occur in different states or quantities, and some errors change the action ranking more than others
- ☐ D A stochastic model always produces a better policy

An aggregate prediction score hides where errors occur. A small error near a decision boundary may matter more than a larger error in an irrelevant observation. Evaluate the resulting decisions as well as prediction fit. Task-aware model training is one possible extension, developed in the appendix.

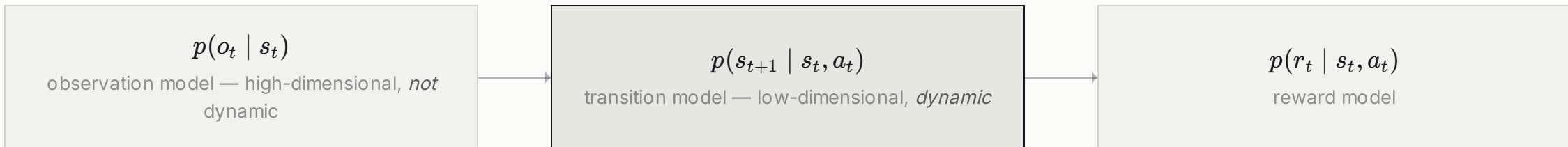
— ACT 4

Learn in latent imagination

Q4. How can a world model train a policy without searching action sequences at every real step?

Version 4 — plan in a latent space

Raw observations may be high-dimensional and partial. A useful latent representation should retain information needed to predict and control; it is not guaranteed to be low-dimensional or Markov.



Learn all three at once, with a deterministic encoder $q_\psi(s_t | o_t) = \delta(s_t = g_\psi(o_t))$:

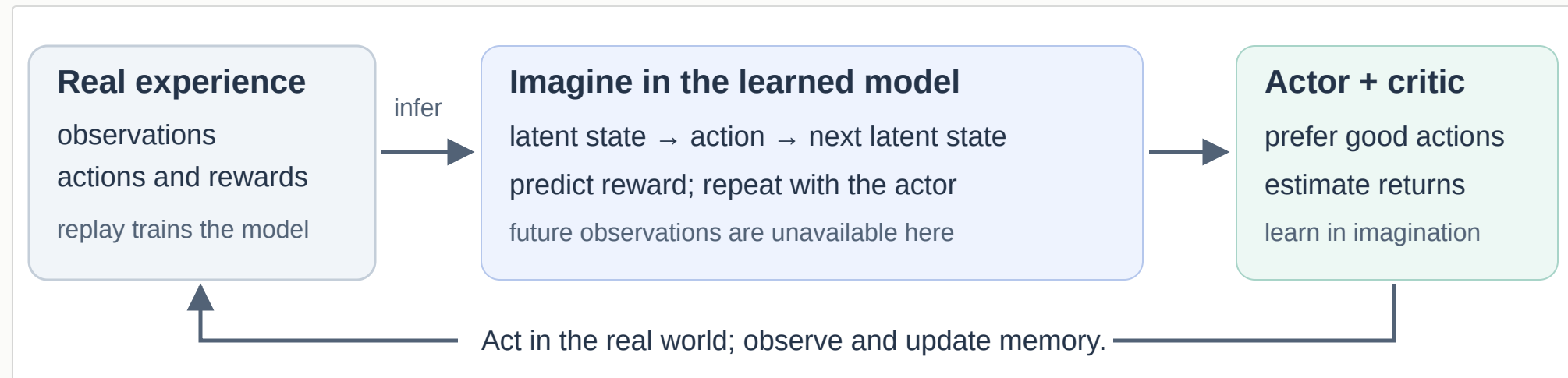
$$\max_{\phi, \psi} \frac{1}{N} \sum_{i, t} \left[\log p_\phi(z_{t+1, i} | z_{t, i}, a_{t, i}) + \log p_\phi(o_{t, i} | z_{t, i}) + \log p_\phi(r_{t+1, i} | z_{t, i}, a_{t, i}) \right], \quad z_{t, i} = g_\psi(o_{t, i}).$$

This is a simplified reconstruction-and-prediction objective, **not the full VAE ELBO**. A partially observed system generally needs history or a belief-state encoder, not one image alone. And the payload of *Embed to Control* is worth naming: it learns a latent space in which the dynamics are locally **linear**, so Lecture 9's LQR applies to a robot controlled from pixels. (Watter et al., 2015; Zhang et al., 2019)

Dreamer — learn feedback inside a latent world model

Hafner et al. · ICLR 2020 · a conceptual redraw of the learning loop

The observation may be an image. Infer a compact model state from the **observation history**, then predict future latent states and rewards under the actor's actions.



The model state summarizes recurrent memory and stochastic latent information. Inferring it from observations differs from predicting it without future observations.

Lecture 3 supplies hidden-state inference, Lecture 6 supplies latent representation learning, and Lecture 10 supplies actor–critic learning. Dreamer joins them inside one feedback loop. [Dream to Control](#)

What imagination teaches — calculate one target

a two-step illustration of bootstrapping, not the full Dreamer loss

From an inferred state, the actor and model predict rewards **2**, then **1**. The critic assigns value **4** to the final imagined state. With $\gamma = 0.9$ and no termination:

$$\hat{G} = 2 + 0.9(1) + 0.9^2(4) = 6.14.$$

COMPONENT	WHAT IT LEARNS FROM THIS COMPUTATION
Critic	Predict future return; 6.14 is one possible bootstrapped training target.
Actor	Prefer actions with higher predicted return. The original continuous-action Dreamer propagates gradients through imagined dynamics.
World model	Fit observed trajectories, rewards and representations from real replay; imagined reward is not a new ground-truth label.

The paper combines different rollout lengths through λ -returns. A convincing imagined return still depends on model and critic accuracy; acting in the real environment supplies the next correction.

Three uses of a model — compare the decision loops

	PETS / MPC	MBPO	DREAMER
Model output	predicted trajectories	short synthetic transitions	latent trajectories and rewards
How actions improve	search candidate sequences	off-policy SAC updates	actor–critic learning in imagination
At deployment	re-plan, execute the first action	act with the learned policy	infer model state, then act with the learned policy
Key risk	plans exploit model error	synthetic data bias value learning	latent predictions bias actor and critic

PETS emphasizes probabilistic ensembles and trajectory sampling. Dreamer does not need to search action sequences at every real step: the actor has already learned from imagined futures.

All three use a learned model, but they spend computation in different places. Compare real interaction cost, training/planning cost and performance under the same evaluation conditions.

From Dreamer to DreamerV3 — read the evidence at the right level

the 2020 paper explains the principle; the 2025 paper tests broader robustness

READ	MAIN QUESTION	EVIDENCE OR MECHANISM TO EXAMINE
Dreamer — ICLR 2020	Can an actor learn from compact imagined futures?	Latent dynamics, predicted rewards and value-based imagination
DreamerV3 — Nature 2025	Can the recipe work across diverse domains with fixed hyperparameters?	Normalization, balancing and transformations; broad empirical evaluation

The 2025 paper reports results across **more than 150 tasks in eight domains** and collecting diamonds in Minecraft without human demonstrations or curricula. This is evidence about the evaluated settings; it does not establish optimality or success on every new control problem.

Learn the original architecture before studying the later implementation details. Versions differ: DreamerV3 uses a score-function actor estimator for both continuous and discrete actions. [Nature paper](#)

The reunion, and the limits

A learned model can support **planning, simulated experience or policy learning in imagination**. These operations connect the course's value and control methods without making them identical.

BENEFIT TO TEST	LIMITATION TO CHECK
Fewer costly real interactions	Fitting the model also consumes data and computation.
More opportunities to improve a policy	Repeated synthetic experience can reinforce the same model error.
Compact representations for image observations	A useful latent state must retain information needed for future decisions.
Faster action selection with a trained actor	Training performance need not transfer to new states or changed dynamics.

PETS plans; MBPO supplies short simulated transitions; Dreamer learns an actor and critic in latent imagination. **Measure actual control performance**, with interaction and computation budgets stated.

Check — what is left to take away

The online model-based methods in this chapter can collect corrective experience. What does Lecture 12 remove during training?

- ☐ A The possibility of fitting any model
- ☐ B The reward values already recorded in the log
- ☐ C Every assumption about state representation
- ☒ D The ability to collect new transitions: training must use a fixed log

A mistaken prediction about an unsupported action cannot be checked by trying it during offline training. This recalls Lecture 5's fixed-data design problem, now with sequential values and policies. Evaluation must distinguish what the log supports from what a model merely predicts.

— CLOSING

Closing

Where we are — Part IV complete

	OR / DYNAMIC PROGRAMMING	CONTROL THEORY
Model-based	MDP & DP (<i>Lec 7 ✓</i>)	Optimal Control (<i>Lec 9 ✓</i>)
Data-driven	Value-Based RL (<i>Lec 8 ✓</i>)	Policy-Based RL (<i>Lec 10 ✓</i>)
Model-Based RL (<i>Lec 11 ✓</i>) — the four, reunited on a learned model		

We can decide statically (Ch 1–6) and dynamically (Ch 7–11); with a model and without; by searching values and by producing policies; and now by **learning the model** so that all of it can collaborate.

What we hand on

MODEL USE LEARNED HERE	HOW IT HELPS CHOOSE AN ACTION
MPC / PETS	Predict candidate trajectories, execute one action, then re-plan from a measurement.
Dyna / MBPO	Generate short synthetic experience to improve a value function and policy.
Dreamer	Learn an actor and critic through imagined latent trajectories.

These online methods use real experience to correct their learned models. Differentiating a planner for task learning or joint design is an **advanced extension** in the appendix.

Lecture 12 asks what changes when training must use a fixed log. We will examine policy restrictions, value learning and the evidence needed to evaluate a new policy.

Model-free RL learned to act by forgetting the
model. Model-based RL *learns it back*.

To plan with it, to control through it, and to let dynamic programming and optimal control — the course's two
parents — finally teach each other.

Questions?

A model can be given, unavailable, or learned. Explain whether the action comes from a fresh plan or a trained actor, and which real observations support its predictions. Next, remove new interaction during training.

— APPENDIX

Backup slides

Complete arguments, kept out of the narrative.

Reading guide — choose how much of the learned model to trust

one main idea to explain, one comparison, one application

ROLE	READ OR REVISIT	QUESTION TO ANSWER
Core	Janner et al., <i>When to Trust Your Model: Model-Based Policy Optimization</i> (NeurIPS 2019)	Why start short model rollouts from real states?
Compare	Hafner et al., <i>Dream to Control: Learning Behaviors by Latent Imagination</i> (ICLR 2020)	How can a latent world model train an actor and critic?
Apply	The original control cases and Hafner et al., <i>Mastering Diverse Control Tasks through World Models</i> (Nature 2025)	What does the later Dreamer result demonstrate beyond the teaching diagram?

Dyna and PETS establish the model-use choices; MBPO and Dreamer are the main comparison. PILCO connects to GPs. Detailed GPS/PLATO and KKT-based MPC derivations remain research extensions.

Backup 1— differentiating the QP / LQR arg min, in full

For a convex QP planning subproblem $\min_z \frac{1}{2} z^\top Q z + q^\top z$ subject to $Az = b$, the optimum is characterised by its KKT system:

$$Qz^* + q + A^\top \nu^* = 0, \quad Az^* - b = 0$$

Rather than differentiate z^* directly, take **differentials** of both conditions:

$$dQ z^* + Q dz + dq + dA^\top \nu^* + A^\top d\nu = 0, \quad dA z^* + A dz - db = 0$$

$$\Rightarrow \begin{bmatrix} Q & A^\top \\ A & 0 \end{bmatrix} \begin{bmatrix} dz \\ d\nu \end{bmatrix} = - \begin{bmatrix} dQ z^* + dq + dA^\top \nu^* \\ dA z^* - db \end{bmatrix}$$

Each parameter perturbation has its own right-hand side; factorization can be reused. For a scalar outer loss, an adjoint solve yields all parameter gradients. Write $[d_z; d_\nu]$ for the solution of the transposed KKT system against $[-\nabla_{z^*} \ell; 0]$. Then the chain rule gives $\nabla_Q \ell = \frac{1}{2}(d_z z^\top + z d_z^\top)$, $\nabla_q \ell = d_z$, $\nabla_A \ell = d_\nu z^\top + \nu d_z^\top$ and $\nabla_b \ell = -d_\nu$.

LQR: the structured linear solve can reuse Riccati factorizations. **Nonlinear MPC:** differentiate a converged local approximation, with a stable active set and nonsingular KKT matrix. If the solver has not converged, differentiating its iterations describes that finite algorithm, not an exact optimum. (*OptNet; Differentiable MPC*)

Backup 2 — guided policy search, the constrained program

Objective. Minimise trajectory cost while forcing the trajectory distribution to agree with the policy:

$$\min_{\theta, p(\tau)} \mathbb{E}_{p(\tau)} \left[\sum_t c(x_t, u_t) \right] \quad \text{s.t.} \quad D_{\text{KL}}(p(x_t)\pi_{\theta}(u_t \mid x_t) \parallel p(x_t, u_t)) = 0 \quad \forall t$$

Lagrangian and alternation. Form $\mathcal{L}_{\text{GPS}}(\theta, p, \lambda) = \mathbb{E}_{p(\tau)}[\ell(\tau)] + \sum_t \lambda_t D_{\text{KL}}(p(x_t)\pi_{\theta}(u_t \mid x_t) \parallel p(x_t, u_t))$ and minimise by turns:

1. **w.r.t.** $p(\tau)$ — trajectory optimisation by iLQG under time-varying linear-Gaussian dynamics; the LQR structure makes the local controller $p(u_t \mid x_t) = \mathcal{N}(K_t(x_t - \hat{x}_t) + k_t + \hat{u}_t, \Sigma_t)$ fall out automatically;
2. **w.r.t.** θ — supervised regression: minimise the weighted sum of KL divergences between π_{θ} and the local controllers;
3. **duals** λ — dual gradient ascent (in practice often *scheduled* rather than updated).

Why the dual step is cheap. With $x^*(\lambda) = \arg \min_x \mathcal{L}(x, \lambda)$ and $g(\lambda) = \mathcal{L}(x^*(\lambda), \lambda)$, the chain rule gives $dg/d\lambda = (d\mathcal{L}/dx^*)(dx^*/d\lambda) + d\mathcal{L}/d\lambda$, and the first term vanishes because $d\mathcal{L}/dx^* = 0$ at the argmin. So $dg/d\lambda = d\mathcal{L}/d\lambda$ evaluated at x^* — no derivative through the inner solve is needed. This envelope calculation differs from the full implicit derivative of an optimizer, developed in the advanced differentiable-control section. **Why it is sample-efficient.** Time-varying linear-Gaussian dynamics are a strong but reasonable *local* assumption for a physical system, so each trajectory is optimised with far fewer samples than general model-free search — the return on injecting control-theoretic structure.

Backup 3 — the four versions of model-based planning, side by side

	DATA	PLANNING	FIXES	STILL BROKEN
V1 Model building + open-loop planning	one batch from π_0	plan once, execute all	nothing	$p_{\pi_f} \neq p_{\pi_0}$
V2 Iterative model building	aggregate π_f 's own visits	plan once, execute all	the <i>training</i> distribution	a mistake mid-plan is never corrected
V3 + MPC	aggregate; refit every N steps	plan, execute the first action, re-plan	mid-plan errors; short horizons suffice	no explicit information-seeking incentive
V3+ + model uncertainty	as V3	plan against an ensemble / GP posterior	over-confidence off-distribution	cost, and the horizon still compounds
V4 + latent space	observations (o, a, o')	plan in the learned latent state	high-dimensional, partial observations	everything above, plus representation error

Propagating uncertainty through a horizon. If the state input is itself uncertain, $x \sim \mathcal{N}(\mu_x, \Sigma_x)$, then $p(f(x) \mid \mu_x, \Sigma_x) = \int p(f(x) \mid x, \mathcal{D}) p(x) dx$, which is intractable and is handled either by Monte-Carlo — sample $x^t \sim p(x)$ and average $p(f(x^t) \mid x^t, \mathcal{D})$ over T particles — or by computing moments analytically where possible and approximating the result as Gaussian, which is PILCO's choice. Rolling a large number of such trajectories and taking their mean and standard deviation gives the fan of predictions that widens with horizon: [compounding uncertainty, drawn](#).

Backup 4 — the sample-efficiency versus model-bias trade, made precise

The gain. A learned model can be queried without environment interaction, so one real transition informs the value or policy at *many* imagined states. Empirically this cuts the required real samples by orders of magnitude — decisive whenever interaction is expensive, slow or dangerous: robots, chemical processes, buildings, patients.

The hazard. Planning trusts f_θ . Where f_θ is wrong the planner optimises a fiction, and errors **compound** over the horizon because each step's error is the next step's input. This is the dynamic cousin of Lecture 5's overestimation trap: an optimiser exploiting a model's blind spots.

Backup 4b — the five mitigations, and where each appeared

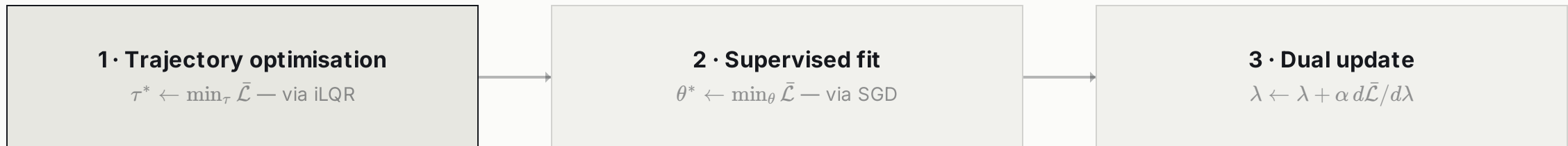
MITIGATION	MECHANISM	WHERE
short planning horizons	fewer compositions of the model with itself	Act 2, V3
re-plan every step	replace predicted inputs with measured ones	Act 2, V3
ensembles and GP posteriors	plan pessimistically where the model disagrees with itself	Act 2, V3+
task loss instead of prediction loss	emphasize errors that affect decisions	advanced differentiable control
structural inductive bias	convexity, linear latents, local linear-Gaussian models	Acts 2 and 4

The through-line of the whole course: **respect the model's uncertainty, or it will be weaponised against you**. Lecture 5 said it about a surrogate over designs; this lecture says it about a surrogate over dynamics; Lecture 12 will say it once more, about a value function, with the escape hatch of fresh data closed.

Guided policy search — solve it by alternation

The constrained program, with an augmented Lagrangian:

$$\bar{\mathcal{L}}(\tau, \theta, \lambda) = c(\tau) + \sum_t \lambda_t^\top (\pi_\theta(x_t) - u_t) + \sum_t \rho_t \|\pi_\theta(x_t) - u_t\|^2$$



Step 1 is Lecture 9. Step 2 is plain supervised learning. Step 3 is dual gradient ascent — and its derivation uses an envelope argument: at the inner optimum $d\mathcal{L}/dx^* = 0$, so **the gradient through the arg min collapses to a single term.**

And the direction of teaching is not one-way: **the optimal-control teacher adapts to the learner**, avoiding actions the student cannot mimic.

What makes it work — and where the trust region reappears

In the stochastic form, the local controller is constrained to stay near the previous one:

$$\min_p \sum_t \mathbb{E}_{p(x_t, u_t)} [c(x_t, u_t)] \quad \text{s.t.} \quad D_{\text{KL}}(p(\tau) \parallel \bar{p}(\tau)) \leq \epsilon, \quad p(u_t \mid x_t) = \pi_\theta(u_t \mid x_t)$$

$$p(u_t \mid x_t) = \mathcal{N}(K_t(x_t - \hat{x}_t) + k_t + \hat{u}_t, \Sigma_t)$$

Look at the second line. The local policy *is* **Lecture 9's** $u = Kx$, with a Gaussian around it — which is why optimising the trajectory hands you the local controller for free, out of the LQR structure.

And the first line is **Lecture 1's trust region, third appearance**: a constraint that keeps each step inside the region where the local model is believable. Lecture 1 measured that region in $\|x - x^{(k)}\|$, Lecture 10 in KL between policies, and here in KL between *trajectory distributions*. Same ratio-and-restrict logic every time.

Because time-varying linear-Gaussian dynamics are a strong and reasonable *local* assumption for a physical system, each trajectory is optimised from very few samples: **5 to 20** where REPS, CEM and RWR need 100 to 800, and nine real manipulation tasks — stacking lego, threading a ring, screwing a bottle cap — converge in about **40 samples each**. Policy search has become supervised learning against an oracle. (*Levine & Abbeel, 2014; Levine et al., 2016*)

The teacher that watches the student — PLATO

The naive version of route 2 collects a dataset from an MPC expert and fits the policy to it. That ignores the one thing imitation always breaks on.

The states visited by the *teacher* are not the states visited by the *learner*, so nothing guarantees long-horizon performance. **PLATO** repairs it by pulling the teacher toward the student at every step:

$$\pi_{\lambda}^t(u \mid x_t, \theta) \leftarrow \min_{\pi} J_t(\pi \mid x_t) + \lambda D_{\text{KL}}(\pi(u \mid x_t) \parallel \pi_{\theta}(u \mid o_t))$$

The only difference from ordinary MPC is that KL term. It encourages the teacher's actions and state distribution to stay closer to the student's, while still reacting competently to surprises the half-trained student could not survive.

Two practical consequences. The MPC teacher may use **full state** at training time while the final policy uses **only the observations** the robot will have at test time — the input-remapping trick. And in flight experiments the crash count stays near zero throughout training, where DAgger's saturates: these are empirical safety results, not a guarantee of zero crashes in other settings. (*Kahn et al., 2017*)

— EXTENSION

Advanced bridge — differentiating a model-based controller

Read after completing the main route.

Treat the planner as a policy class

Every version in Act 2 fits f_θ by minimising $L(x_{t+1}, f_\theta(x_t, u_t))$ — a loss that is **task-independent and data-dependent**. An aggregate prediction objective need not emphasize the errors that most affect action selection. Its weights and the training distribution determine which errors it prioritizes.

But $\text{MPC}(\cdot)$ maps a state to an action, which is what a policy is. So fit its parameters end to end on a task loss — for instance, imitating an expert controller $\hat{\theta}$:

$$\mathcal{L}(\theta) = \mathbb{E}_x \left[\| u_{1:T}(x; \theta) - u_{1:T}(x; \hat{\theta}) \|^2 \right]$$

The trained object is still the model, the execution is still planning — only the loss moves from **prediction** to **task**.

Is the arg min differentiable? Partially, yes

For a convex quadratic planning subproblem — an LQR, or a QP —

$$\tau_{1:T}^* = \arg \min_{\tau_{1:T}} \sum_t \frac{1}{2} \tau_t^\top C_t \tau_t + c_t^\top \tau_t \quad \text{s.t.} \quad x_1 = x_{\text{init}}, \quad x_{t+1} = F_t \tau_t + f_t$$

With a locally unique optimum and a nonsingular KKT system (and a stable active set for inequalities), the solution is differentiable locally. Do not differentiate τ^* ; **differentiate the conditions it satisfies**:

$$\mathcal{L}(\tau, \lambda) \implies \text{KKT} \implies \text{take differentials} \implies \frac{\partial \tau^*}{\partial \theta}$$

The differentials give a *linear system* in $(d\tau, d\lambda)$ whose matrix is the KKT matrix already formed in the forward pass — so the backward pass is one more solve of the same structure. For LQR that solve **is itself an LQR problem**: one Riccati sweep. (*Amos & Kolter, 2017; Amos et al., 2018*)

And the saving is measurable: differentiating the fixed point costs **about 1.5×10^{-2} s over the horizons shown in the reported benchmark, not a horizon-independent complexity guarantee**, while unrolling the iLQR solver and backpropagating through every iteration costs ≈ 3 s at 128 steps — two orders of magnitude, and growing.

Differentiate a solution — first do it with one number

A one-step planner chooses $u^*(\theta)$ by minimizing $\frac{1}{2}(u - \theta)^2 + \frac{1}{2}u^2$.

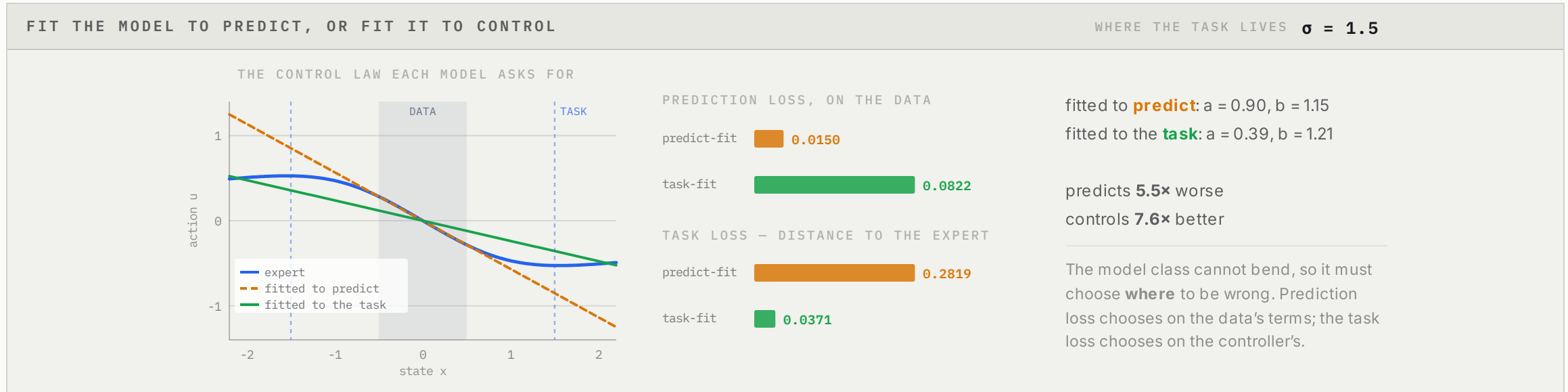
The optimality condition is $2u^* - \theta = 0$. Differentiate that condition:

$$2 \frac{du^*}{d\theta} - 1 = 0 \quad \Rightarrow \quad \frac{du^*}{d\theta} = \frac{1}{2}.$$

If the training goal is an expert action 1, let $\ell = \frac{1}{2}(u^* - 1)^2$. At $\theta = 0$, $u^* = 0$ and $d\ell/d\theta = (-1)(1/2) = -0.5$.

We differentiate **the equation defining the solution**. The KKT matrix method is this same calculation with several variables and constraints.

A model fitted to be wrong in the right places



The same misspecified model class, fitted twice: once to minimise prediction error on the data, once to minimise the *task* loss by differentiating through the one-step planner. Move the task away from where the data was collected and the two fits separate — the task-fitted model becomes a **worse predictor and a better controller**. Both losses are printed; neither is asserted.

Why it matters, and where the evidence is

- **Imitation learning.** Fit θ so that $\pi_\theta(s) \approx \hat{\pi}(s)$, inheriting a controller's structure while learning from demonstrations rather than from a hand-written cost.
- **Structure as inductive bias.** The constraint $x_{t+1} = f_\theta(x_t, u_t)$ is **baked into the architecture**, so far less data is needed than for a generic policy network, and extrapolation improves.
- **Task-aligned models.** Accuracy lands where the task needs it, which is the direct repair of Act 2's "prediction-accurate, control-useless" failure.

MEASURED, ON A DELIBERATELY MISSPECIFIED PENDULUM

The expert's dynamics lie outside the learner's model class. Against a plain system-identification baseline, the task-loss method ends with a **higher system-ID loss (0.0025 vs 0.0020)** and a **lower imitation loss (0.12 vs 0.19)**. A model fitted worse, in the way the task does not care about, controls better.

The honest caveat: for nonlinear problems this needs iLQR to reach a fixed point, and sometimes it does not — then you are back to unrolling the solver. The controller becomes **a differentiable layer**, and like every layer it has conditions of use.

A model built to be planned in — convexity, learned

Every version so far fits a model and hopes the planner copes. Turn the question round: what property should a learned model *have* so that planning inside it is easy?

Lecture 1's answer, imposed on a network. An **input-convex** neural network is convex in its input by construction — every $W^{(z)}$ non-negative, every activation convex and non-decreasing — so the predictive-control problem

$$\min_{u_t, \dots, u_{t+T}} \sum_{\tau} f(x_{\tau-n_w}, \dots, x_{\tau}) \quad \text{s.t.} \quad s_{\tau} = g(x_{\tau-n_w}, \dots, u_{\tau}), \quad \underline{u} \leq u_{\tau} \leq \bar{u}, \quad \underline{s} \leq s_{\tau} \leq \bar{s}$$

is convex in the actions **only when the whole formulation preserves convexity**: appropriate monotone compositions, convex inequality constraints, and affine equalities or a valid reformulation. An ICNN dynamics model alone does not make nonlinear equality constraints convex. (*Amos, Xu & Kolter, 2017; Chen, Shi & Zhang, 2019*)

THE BILL, PAID IN A REAL BUILDING

An input-convex recurrent model fits building dynamics as accurately as an ordinary RNN, and the controller built on it finds actions worth **11.52% more energy saving** — while the ordinary RNN's decisions "vary dramatically". Later work ran the same controller on a real ETH building for a fortnight. **Lecture 1's convexity was never a mathematical convenience; it can make the optimization tractable; predictive accuracy, feasibility, and real-system performance still need separate checks.**

— EXTENSION

Further model uses — direct gradients, teachers and design

Read after completing the main route.

Three routes from a model to a policy

HOW THE MODEL HELPS	REPRESENTATIVE METHOD	WHAT THE POLICY LEARNER RECEIVES
Differentiate predictions	PILCO; the original Dreamer	Gradients of predicted future performance
Supply a teacher	Guided policy search	Actions from an optimized local controller
Supply experience	Dyna → MBPO	Simulated transitions for value/policy updates

Read the first two as connections to control. Then follow Dyna → MBPO and compare Dreamer's latent imagination. Dreamer also trains a critic; these categories describe uses of a model, not mutually exclusive algorithm families.

Ask what the model produces and how that output updates the policy. The detailed GPS/PLATO derivations are retained in the appendix.

Route 1 — backpropagate the model into the policy

Compose policy and model along the horizon and differentiate the whole chain:

$$D_{\theta}a_t = \partial_{\theta}\pi_{\theta}(s_t) + \partial_s\pi_{\theta}(s_t)D_{\theta}s_t, \quad D_{\theta}s_{t+1} = f_s D_{\theta}s_t + f_a D_{\theta}a_t.$$

Starting from $D_{\theta}s_0 = 0$, propagate these sensitivities and sum $\nabla_{\theta}J = \sum_t (r_s D_{\theta}s_t + r_a D_{\theta}a_t)$, with discount factors when needed.

PILCO does this with a *probabilistic* model — a GP on the state difference $\Delta_t = x_t - x_{t-1}$ — and propagates the whole distribution forward by moment matching, so long-horizon planning carries the model's own uncertainty. Moment calculations are analytic for the selected GP and cost forms, but the Gaussian rollout distribution is an approximation. (*Deisenroth & Rasmussen, 2011*)

The result is the data-efficiency headline of the field: real cart-pole swing-up *and* balance from **17.5 seconds of interaction with the physical hardware**; a robotic unicycle in $\mathbb{R}^{12}$ from about 20 trials. Against the methods of the decade before it, roughly **three orders of magnitude less interaction**.

What limits direct policy optimization?

ISSUE	WHY IT MATTERS	WHAT CAN HELP
Model bias	Optimizing predicted reward can exploit dynamics errors.	Real-data collection, uncertainty estimates and limited imagination horizons
Long gradient chains	Repeated Jacobian products can vanish or explode.	Suitable representations, shorter horizons and bootstrapped value estimates
Uncertain predictions	A probabilistic model can still be miscalibrated.	Check predictions where the current policy actually visits

PILCO propagates uncertainty in a GP model. Dreamer later learns a latent state model and combines imagination with a critic. Neither removes the need to learn useful dynamics from real experience.

Direct gradients are one design choice. Score-function policy gradients avoid differentiating the dynamics but introduce their own variance; neither estimator is uniformly better in every problem.

Route 2 — let optimal control teach

Lecture 1 wrote an optimisation. Lecture 9 added the dynamics as a constraint. Add one more constraint and you have this lecture.

	OBJECTIVE	DYNAMICS CONSTRAINT	POLICY CONSTRAINT
Optimisation (<i>Lec 1</i>)	$\min_u c(u)$	—	—
Optimal control (<i>Lec 9</i>)	$\min_{u,x} \sum_t c(x_t, u_t)$	$x_{t+1} = f(x_t, u_t)$	—
+ imitation (<i>Lec 11</i>)	$\min_{u,x,\theta} \sum_t c(x_t, u_t)$	$x_{t+1} = f(x_t, u_t)$	$u_t = \pi_\theta(x_t)$

Three rows, three lectures. The middle row constrains the trajectory to be *physical*; the bottom row constrains it to be **reproducible by a policy**. Everything in guided policy search follows from that one extra line.

The hazard, admitted by the method's own authors

WHAT BREAKS

Imagination rollouts "can suffer from **severe bias when the learned model is inaccurate**". In continuous control it proved "very difficult to train nonlinear neural network models for the dynamics that would actually improve the efficiency of Q-learning when used for imagination rollouts."

The fix that worked was to give up expressiveness: **iteratively refitted time-varying linear dynamics**. (*Gu et al., 2016*)

WHAT IS WORTH KEEPING

The same paper answers Lecture 8's parting wall from the other side. Write Q as a value plus a quadratic advantage,

$$Q(x, u) = V(x) - \frac{1}{2} (u - \mu(x))^{\top} P(x) (u - \mu(x))$$

with $P(x) \succ 0$, the unique maximizing action is $\mu(x)$, analytically.

The continuous arg max is solved by **assuming LQR structure**: Lecture 9 answering Lecture 8 directly.

And design optimisation returns — bilevel

Lecture 1 optimised a wind-farm layout with a trust region, and Lecture 5 designed from a fixed dataset. Neither could touch the *controller*. Differentiable control closes that loop.

Many real problems are **bilevel** — an upper-level *design* p wrapping a lower-level *control* problem:

$$\min_p \sum_t \mathcal{L}(x_{t+1}, u_t^*; p) \quad \text{s.t.} \quad u_{0:T-1}^* = \arg \min_u \sum_t \mathcal{L}(x_{t+1}, u_t; p), \quad x_{t+1} = f(x_t, u_t; p)$$

Turbine layout with the yaw controller that will run on it; furnace geometry with the recipe that will heat it. The advanced differentiable-control section supplies a possible derivative: **differentiate the optimal control through to the design, and optimise both together** rather than designing first and controlling afterwards.

The evidence is already in the room. An input-convex model inside an MPC loop saved **11.52%** of a building's energy and then ran on a real one; the differentiable-MPC literature names **HVAC control and furnace control** as its motivating applications. Lecture 1's `trust-region` widget solved the wind-farm layout by a staircase of convex problems with the controller held fixed. **Here the controller stops being fixed.**