Applied Computational Mathematics Lab

TEDDY LAZEBNIK'S COURSE · LECTURE 9

Reinforcement Learning

Lecture 9 · Planning and models: learning models of the world, planning with them, Dyna, replay as a model, Monte Carlo tree search, and AlphaZero



Teddy Lazebnik · Applied Computational Mathematics Laboratory

BEFORE WE START

Recap, and what this lecture does

- So far: bandits (explore vs exploit); dynamic programming (solve a known MDP); model-free prediction and control (learn from interaction); function approximation (generalise).

DYNAMIC PROGRAMMING

Assume a model

Solve it; no need to interact with the world.

MODEL-FREE RL

No model

Learn values from experience.

MODEL-BASED RL

Learn a model

From experience; then plan values with it.



IN PLAIN WORDS

The best of both worlds: learn a model from experience, then plan with it. Experience no longer teaches the values directly; it teaches a model, and the model teaches the values.

CAREFUL WITH WORDS

Until now, “the model” often meant the policy network. Today a model means a **model of the world**: how states and rewards follow actions.

BEFORE WE START

Today's map

PART 1

Learning a model

Expectation, stochastic and full models; tables and linear models.

PART 2

Planning with a learned model

Dynamic programming, sample-based planning, and the limits of a wrong model.

PART 3

Dyna

Learn and plan at the same time; a maze, and a maze that changes.

PART 4

Replay as a model

When a buffer is enough, and when a learned model is worth it.

PART 5

Planning to act

Forward search, Monte Carlo tree search, and UCT on tic-tac-toe.

PART 6

AlphaGo and AlphaZero

Self-play, a policy-value network, and search.



IN PLAIN WORDS

Reading: Sutton & Barto (2018), chapter 8; Silver et al. (2016, 2017, 2018). By the end you should be able to run Dyna-Q and one round of MCTS by hand.

Applied Computational Mathematics Lab

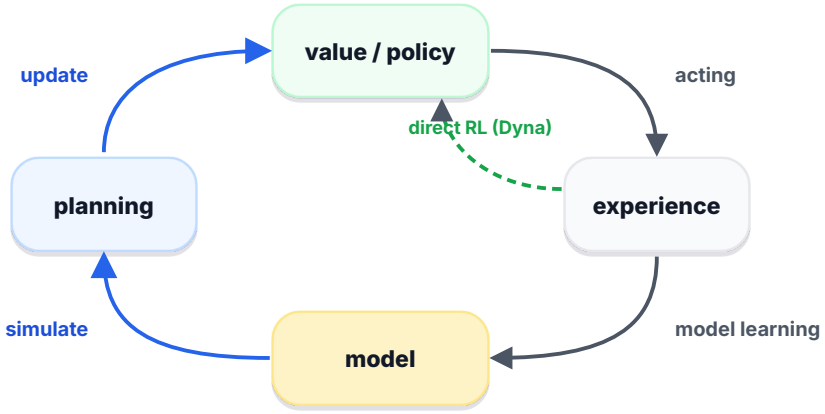
PART 1

Learning a model

Supervised learning of the world's dynamics.



Model-based RL



- Learn a model from experience; plan value functions (and/or policies) with it; act; collect more experience.
- **Why bother?** One clear disadvantage: first learn a model, then a value function from it: **two** sources of approximation error instead of one.
- However:
  - models can be learned efficiently by supervised learning;
  - we can reason about model uncertainty (better exploration?);
  - we can reduce interactions with the real world (data efficiency: faster, cheaper, safer).



IN PLAIN WORDS

A robot run needs a robot, a room, sensors and people; on a bad day the robot breaks. Asking a model costs a few chips and some electricity.

What is a model?

- A model $\mathcal{M}_\eta$ is an approximate representation of an MDP $\langle \mathcal{S}, \mathcal{A}, \hat{p}_\eta \rangle$.
- For now, the states and actions are the same as in the real problem.
- The dynamics $\hat{p}_\eta$ are parametrised by weights η, and approximate the transitions and rewards:

$$R_{t+1}, S_{t+1} \sim \hat{p}_\eta(r, s' \mid S_t, A_t), \qquad \hat{p}_\eta \approx p$$

- Learning it is **supervised learning** over the observed transitions:

$$S_1, A_1 \to R_2, S_2, \qquad \ldots, \qquad S_{T-1}, A_{T-1} \to R_T, S_T$$

- Choose a functional form $f_\eta(s, a) = (r, s')$, pick a loss (e.g. mean squared error), and minimise the empirical loss.
- Notice what we learn: not the policy, the **dynamics**.

KEEP THEM APART

Train two models, one for rewards and one for transitions. The reward depends on what we want; the transitions don't. Learn how a robot's world moves on one task, and you get it for free on the next.

Expectation models

- Minimising squared error gives an **expectation model**: we hope $s' \approx \mathbb{E}[S_{t+1} \mid S_t = s, A_t = a]$.
- Disadvantage: if an action randomly goes left or right past a wall, the expected next state is **inside the wall**.
- With linear values we are mostly fine. With $f_\eta(\phi_t) = \mathbb{E}[\phi_{t+1}]$ and $v_\theta(\phi) = \theta^\top \phi$:

$$\mathbb{E}[v_\theta(\phi_{t+1}) \mid S_t = s] = \mathbb{E}[\theta^\top \phi_{t+1} \mid S_t = s] = \theta^\top \mathbb{E}[\phi_{t+1} \mid S_t = s] = v_\theta\big(\mathbb{E}[\phi_{t+1} \mid S_t = s]\big)$$

- If the model is linear too, $f_\eta(\phi_t) = P\phi_t$, we can even unroll it many steps: $\mathbb{E}[v_\theta(\phi_{t+n})] = v_\theta(\mathbb{E}[\phi_{t+n}])$.



IN PLAIN WORDS

On a grid, an expectation can land half a cell away: a place that doesn't exist.

TRY IT

A world pays −1,000,000 once, +1,000,000 once, and 0 the other 998 times out of 1,000. The expected reward is 0: the model did its job faithfully, and it's still useless for judging risk.

Stochastic and full models

- We may not want to assume everything is linear: then expected states may not be real states, and iterating the model may do weird things.
- **Stochastic (generative) models** add a noise input ω:

$$\hat{R}_{t+1}, \hat{S}_{t+1} = \hat{p}(S_t, A_t, \omega)$$

- They can be chained, even when non-linear, but they add noise.
- **Full models** give the complete transition distribution. They are hard to iterate because of branching:

$$\mathbb{E}[v(S_{t+1}) \mid S_t = s] = \sum_a \pi(a \mid s) \sum_{s'} \hat{p}(s, a, s') \big(\hat{r}(s, a, s') + \gamma v(s')\big)$$

- Every further step multiplies the number of terms by $|\mathcal{A}| \times |\mathcal{S}|$.



IN PLAIN WORDS

If you've met diffusion models, it's the same spirit: start from noise, and shape it into something that fits the condition.

EXAMPLES OF MODELS

We usually split the dynamics into a transition model and a reward model, and pick for each: a table lookup model, a linear expectation model, or a deep network.

Table lookup models

- The model is an explicit MDP. Count visits $N(s,a)$ to each state-action pair:

$$\hat{p}_t(s' \mid s,a) = \frac{1}{N(s,a)} \sum_{k=0}^{t-1} \mathbb{I}(S_k = s, A_k = a, S_{k+1} = s'), \qquad \mathbb{E}_{\hat{p}_t}[R_{t+1} \mid s,a] = \frac{1}{N(s,a)} \sum_{k=0}^{t-1} \mathbb{I}(S_k = s, A_k = a) R_{k+1}$$

- The AB example again (Lecture 5): two states, no discounting, eight episodes.

EPISODES	WHAT HAPPENED
1	A, 0, B, 0, end
6	B, 1, end
1	B, 0, end

THE TABLE MODEL

From A: always to B, reward 0.
From B: the episode ends; reward 1 in 6 of 8 visits, so $\hat{p}(r = 1 \mid B) = 0.75$.
A whole model, learned by counting.



IN PLAIN WORDS

No action choices in this one: things just happen. That's fine: it's still a model of the dynamics.

Linear expectation models

- Assume a feature representation ϕ is given, and parametrise rewards and transitions separately, each linear in the features:

$$\hat{s}'(s,a) = T_a\,\phi(s), \qquad \hat{r}(s,a) = w_a^\top \phi(s)$$

- One square matrix T_a and one vector w_a per action. On each transition (s,a,r,s'), take a gradient step on

$$L(s,a,r,s') = \left\|\phi(s') - T_a\phi(s)\right\|^2 + \left(r - w_a^\top\phi(s)\right)^2$$



IN PLAIN WORDS

The model predicts how the features change, not raw states. Combined with a linear value function, it plans directly in feature space.

TRY IT

$\phi(s) = (1,2)$, $T_a = \begin{pmatrix} 0 & 1 \\ 0.5 & 0 \end{pmatrix}$, $w_a = (1,-1)$.
Predicted $\hat{s}' = (2, 0.5)$ and $\hat{r} = 1 - 2 = -1$.
Observed $\phi(s') = (2,1)$ and $r = 0$: loss $0^2 + 0.5^2 + 1^2 = 1.25$.

Check yourself: Part 1

CHECK YOURSELF

1. Why does model-based RL have two sources of approximation error?

CHECK YOURSELF

2. An action moves you to $x = -1$ or $x = +1$ with equal probability, and there's a wall at $x = 0$. Where does an expectation model put you?

CHECK YOURSELF

3. From state C you saw 4 transitions: twice to D with reward 2, once to E with reward 0, once to D with reward 1. What is the table model for C?

CHECK YOURSELF

4. Why keep separate reward and transition models?





HINT

Answers are at the end of the deck.

Applied Computational Mathematics Lab

PART 2

Planning with a learned model

Spend compute, not real interaction.



What we mean by planning

- Planning means different things to different communities (closer to prediction in robotics, to optimisation in engineering).
- For us: **investing compute to improve values and policies**, without interacting with the environment.
- Dynamic programming is the best example so far, but it needed a perfect specification of the environment. Today the planning algorithms use **learned** models.

OPTION 1

DP on the learned model

Solve $\langle \mathcal{S}, \mathcal{A}, \hat{p}_\eta \rangle$ with value or policy iteration. Needs a full model; sweeps every branch.

OPTION 2

Sample-based planning

Use the model only to generate samples $S, R \sim \hat{p}_\eta(\cdot \mid s, a)$; apply model-free RL (MC, SARSA, Q-learning) to them.



IN PLAIN WORDS

A model is just a simulator you learned yourself: cheap to run, and as honest as the data it came from.

Back to the AB example

REAL EXPERIENCE	SAMPLED FROM THE TABLE MODEL (OUR DRAW)
A, 0, B, 0, end	A, 0, B, 1, end
B, 1, end	B, 1, end
B, 1, end	A, 0, B, 1, end
B, 1, end	B, 1, end
B, 1, end	B, 1, end
B, 1, end	B, 1, end
B, 1, end	B, 1, end
B, 0, end	B, 0, end

- Monte Carlo on the **real** data: $V(A) = 0$, $V(B) = 0.75$.
- Monte Carlo on our **8 sampled** episodes: $V(A) = 1$, $V(B) = 0.875$.
- Under the model itself the answer is $V(A) = V(B) = 0.75$. With 20,000 sampled episodes: $V(A) \approx 0.76$, $V(B) \approx 0.75$.



IN PLAIN WORDS

Even a model that's exactly right about the data gives wrong values if we sample too little from it. More samples fix it: the law of large numbers, again.

When the model is wrong

- Given an imperfect model $\hat{p}_\eta \neq p$, planning may compute a suboptimal policy.
- Performance is limited to the optimal policy of the approximate MDP $\langle \mathcal{S}, \mathcal{A}, \hat{p}_\eta \rangle$: model-based RL is only as good as the estimated model.
- How to live with the inevitable inaccuracies:
  - when the model is wrong, use model-free RL;
  - reason about model uncertainty over η (e.g. Bayesian methods);
  - combine model-based and model-free methods in one algorithm (next: Dyna).



IN PLAIN WORDS

You can test a model like any supervised model, but only on what you've already seen. In RL there are always things you haven't seen yet.

FROM CLASS (AN EXTREME)

A trading model that is wrong every single time: when you should buy, it says sell. Excellent model: do the opposite. Errors you can predict are information.

Real and simulated experience

- Two sources of experience:

	SAMPLED FROM
Real experience	the environment (true MDP): $r, s' \sim p$
Simulated experience	the model (approximate MDP): $r, s' \sim \hat{p}_\eta$

APPROACH	MODEL?	LEARNS VALUES FROM
Model-free RL	none	real experience
Model-based RL (sample-based planning)	learned from real experience	simulated experience
Dyna	learned from real experience	real **and** simulated experience



IN PLAIN WORDS

In Dyna, real and simulated experience are treated the same way: the values don't care where a sample came from.

Check yourself: Part 2

CHECK YOURSELF

5. What does sample-based planning need from the model, and what does DP need?

CHECK YOURSELF

6. In the AB example, why did our 8 sampled episodes give $V(A) = 1$ instead of 0.75?

CHECK YOURSELF

7. A model is perfect on all the data you have. Can planning with it still give a bad policy?



Applied Computational Mathematics Lab

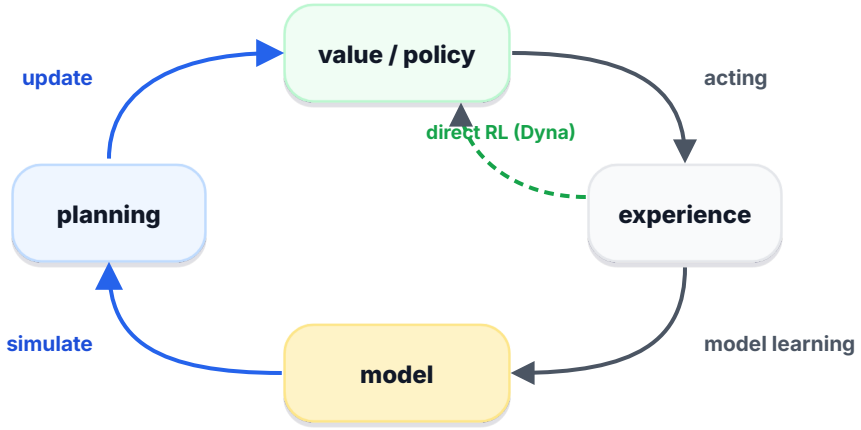
PART 3

Dyna

Learn and plan at the same time.



The Dyna architecture



Dyna adds the green arrow: real experience also updates the values directly.

- Learn a model from real experience.
- Learn **and** plan the value function (and/or policy) from real and simulated experience.
- Conceptually, updates from learning and from planning are not distinguished.



IN PLAIN WORDS

Why have a model at all? Not because it's more accurate: because real samples are expensive. For every real step, take many imagined steps.

THE ARITHMETIC

Half real, half imagined would save nothing: every imagined sample would cost a real one. Nine imagined steps per real step gives ten times the updates.

The Dyna-Q algorithm

DYNA-Q

1. Initialise $Q(s, a)$ and $\text{Model}(s, a)$ for all s, a.
2. Loop: $S \leftarrow$ current state; $A \leftarrow \varepsilon$-greedy(S, Q).
3. Take A; observe R, S'.
4. $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$.
5. $\text{Model}(S, A) \leftarrow R, S'$ (deterministic environment).
6. Repeat n times: pick a previously seen S and an action A taken there; $R, S' \leftarrow \text{Model}(S, A)$; apply the same Q-learning update.

After Sutton & Barto (2018), section 8.2.



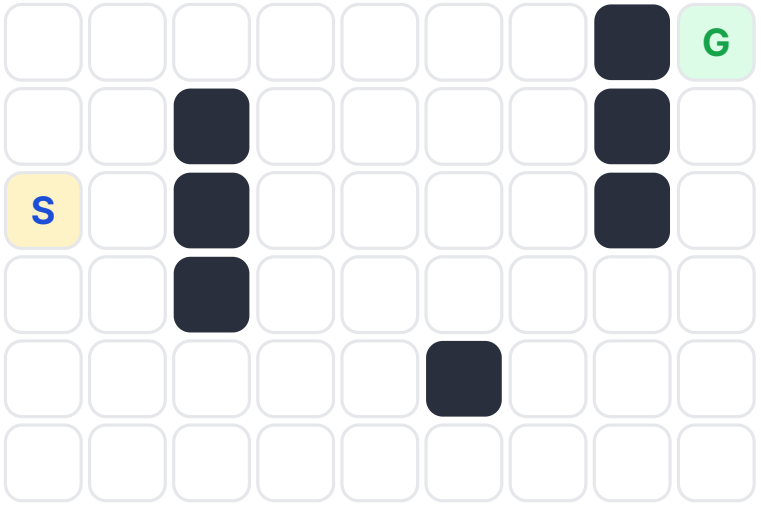
IN PLAIN WORDS

Why only states we've already seen? A model can't invent new states, or worse, it invents them by mistake. In chess, a white pawn can never stand on the first row; a sloppy model might predict exactly that.

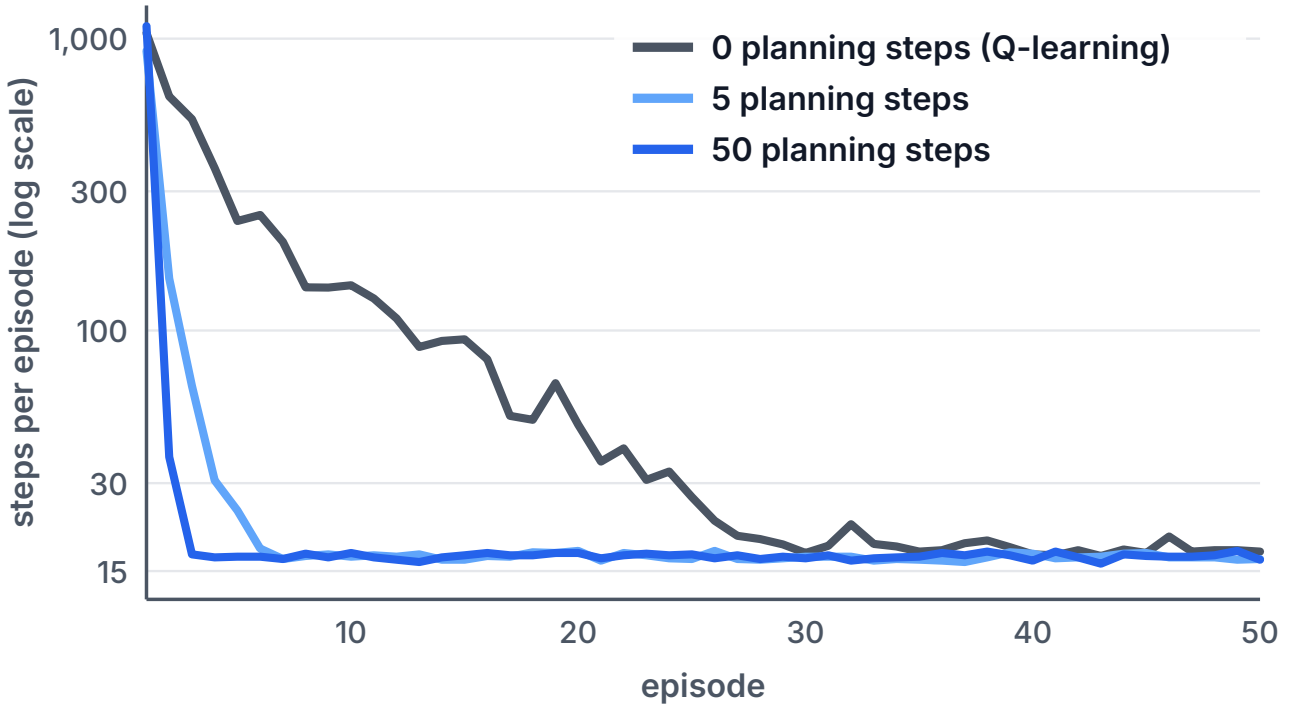
READ IT AS

Steps 3–4 are plain Q-learning on real experience. Steps 5–6 turn the same model into extra, imagined experience.

Dyna-Q on a maze



Our maze (after Sutton & Barto, example 8.1): 6 × 9, reward 1 at G, $\gamma = 0.95$, $\alpha = 0.1$, $\varepsilon = 0.1$. The shortest path takes 14 steps.



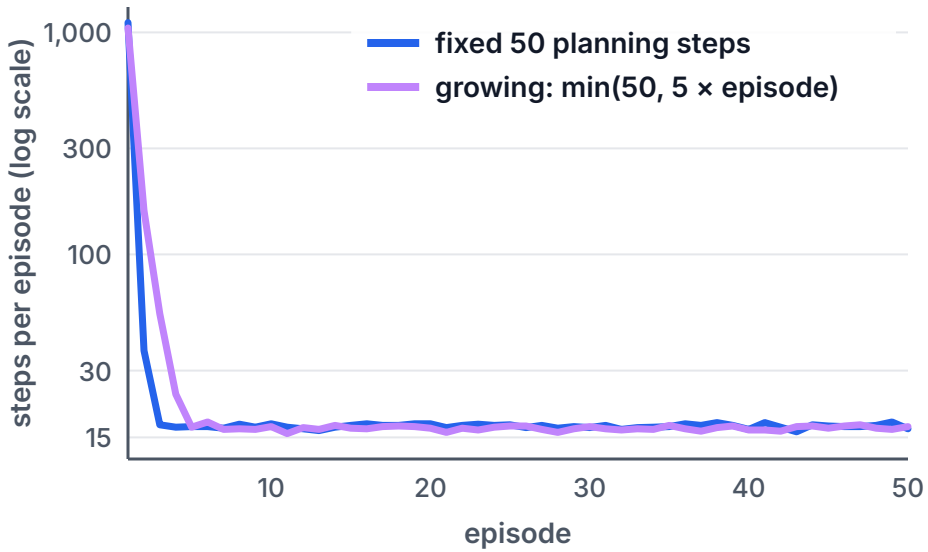
Our runs: 30 runs averaged. Episode 2: about 634, 151 and 37 steps. By episode 10, planning walks the maze in about 17; plain Q-learning still takes about 143.

Why planning helps so much

- When the goal is first reached, only one state learns about it.
- With planning, that news spreads backwards through the whole maze before the next episode starts. Same real experience: the imagined steps do the rest.
- At the start the model is poor (with a learned, noisy model). One option: let the number of planning steps grow as the model improves.

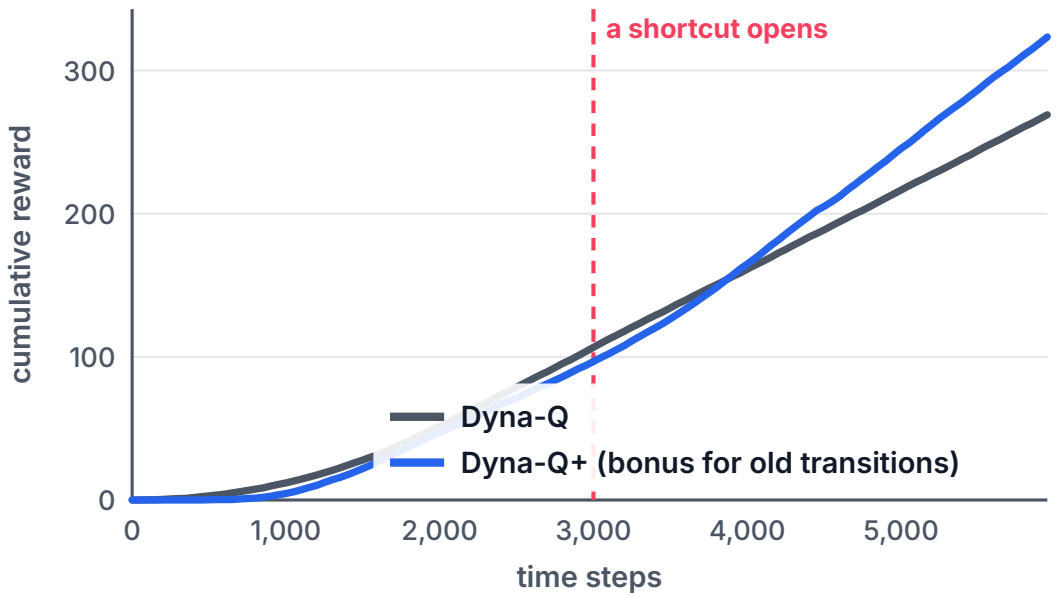
TRY IT

Our schedule: $n = \min(50, 5 \times \text{episode})$. Episode 2 took about 158 steps, and by episode 10 about 16.8. A fixed 50 did better early here, because our model is a table of what happened: never wrong about what it saw.



Our runs, 30 runs averaged.

When the world changes



Our runs: a shortcut maze (after Sutton & Barto, example 8.3), 10 runs. Steps per goal before / after the change: Dyna-Q 18 / 18; Dyna-Q+ 20 / 12.

- A model is also a prior, and priors resist change.
- When the world gets **easier** (a shortcut opens), Dyna-Q's model never learns about it: it keeps exploiting the old path.
- **Dyna-Q+** adds a small bonus $\kappa\sqrt{\tau}$ to transitions not tried for τ steps, so it goes back to check. It found the shortcut; it pays a little exploration cost before the change.
- When the world gets **harder**, the model is corrected quickly: reality disagrees as soon as you hit the new wall.

Why combine learning and planning?

- We can sink in more compute to learn more efficiently from every real step.
- This matters most when real data is:
  - expensive or slow (robotics);
  - unsafe (autonomous driving).



IN PLAIN WORDS

Dyna in one line: real steps teach the model, and imagined steps teach the values.

A REAL PROJECT

An autonomous boat: a day of sea trials needs a ship to carry the small boat out and a crew, and costs tens of thousands of shekels. A physics simulation of the boat is expensive too, but still tens of times cheaper than a day at sea. The agent doesn't need to solve turbulence: give it a goal and enough practice.

TRY IT

A model call costs 10 ms and a real step 100 ms. Up to 9 model calls per real step and you still save time compared with taking 10 real steps; at 11 you are slower.

Check yourself: Part 3

CHECK YOURSELF

8. In Dyna-Q, which updates use real experience and which use the model?

CHECK YOURSELF

9. Why does Dyna-Q plan only from states it has already visited?

CHECK YOURSELF

10. Why did plain Dyna-Q never find the shortcut, while Dyna-Q+ did?



Applied Computational Mathematics Lab

PART 4

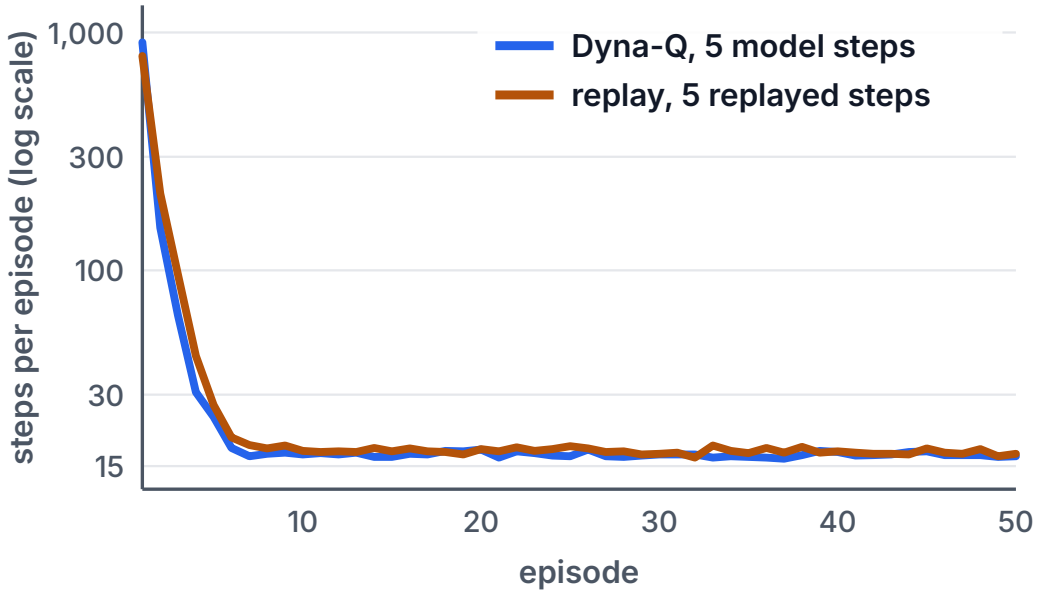
Replay as a model

Is a replay buffer already a model?



Beyond model-based and model-free

- Traditional algorithms didn't store experience, so they fell neatly into two groups: model-free (values and/or policy, no dynamics model) and model-based (learn transitions and rewards, plan from them).
- Today agents store transitions in a **replay buffer** and apply model-free RL to samples from it.
- That is just Dyna, with the buffer as a **non-parametric model**: we sample a whole transition (s, a, r, s') instead of a pair (s, a) and asking the model for r, s'.



Our runs, same maze. Episode 2: about 210 steps with replay, 151 with the model; by episode 10 both are at about 17. Replay draws transitions as often as they happened; our model picks seen pairs evenly.

Parametric model or replay buffer?

- For tabular RL some model-based and model-free algorithms give exactly the same output: a perfect model gives the same answer as a replay buffer for every (s, a) it has seen.
- Could model errors help learning? Unlikely, if we only imagine transitions from state-action pairs we actually tried.
- But a parametric model is more flexible than a buffer:

USE OF A MODEL	THE QUESTION IT ANSWERS
Planning for action selection	what happens if I take this action next?
Counterfactual planning	what if I had taken that other action back then?
Backwards planning	which states could have led to this outcome?
Jumpy planning	what happens over a longer time scale, for long-term credit?



IN PLAIN WORDS

A buffer can only replay what happened. A model can imagine what didn't: powerful, and exactly where a model can fool you.

Compute and memory

	REPLAY BUFFER	PARAMETRIC MODEL
Query cost	very cheap	can be expensive (e.g. a large generative network)
Memory	grows linearly with capacity	fixed, often comparably small
Can answer	what happened	what could happen
Up-front cost	none	must be trained, on real samples

- A bigger buffer often speeds learning up, until the returns diminish and you hold lots of useless memory: spend memory to save compute, or the reverse.



IN PLAIN WORDS

Before building a model, ask one question: how much does a real sample cost me?

FROM CLASS

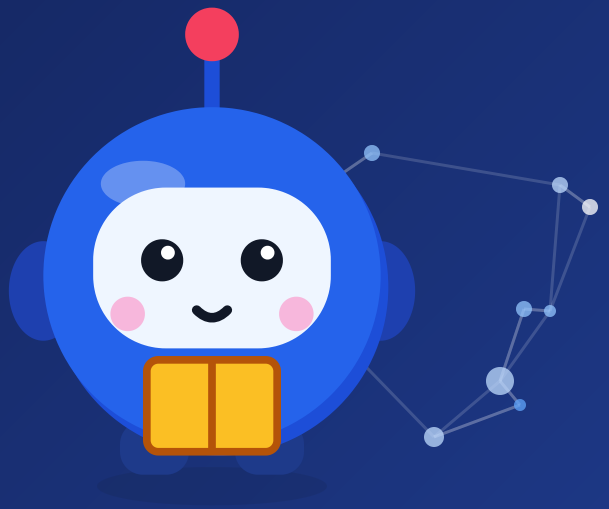
Markov says the present state is enough. In practice, remembering more often helps: in poker, the cards decide your move, but the history tells you who bluffs and who folds early.

Applied Computational Mathematics Lab

PART 5

Planning to act

Search from the state you are in, right now.



Planning for action selection

- So far, planning improved a **global** value function. Now: plan for the near future, to select the **next action**.
- The states you may meet from now are a small slice of the world: a local value function can be much more accurate than the global one.
- Inaccuracies in the model then mostly lead to interesting exploration, rather than bad updates everywhere.



IN PLAIN WORDS

A chess player doesn't evaluate every position in chess: just the ones that can follow from the board in front of them.

ANOTHER WAY TO SEE IT

A tourist with one free afternoon doesn't study the map of the whole city: just the streets around the hotel.

COMPARE

Simple Monte Carlo search computes $Q(s_t, a)$ for the current state only: it is one step of policy improvement, done locally.

Forward search and expectimax

- Forward search selects the best action by lookahead: build a search tree with the current state s_t at the root, and use the model to look ahead.
- No need to solve the whole MDP, just the sub-MDP starting from now.
- With a model, an **expectimax tree** computes optimal $q(s, a)$ for the current state: max over our actions, average over the world's responses.
- Limitation: the tree grows like $(|\mathcal{S}||\mathcal{A}|)^H$ for horizon H.

GAME	COUNT
Tic-tac-toe: ways to fill the board, move by move	362,880 $(9!)$
Go: first moves	361
Go: four moves deep	$361^4 \approx 1.7 \times 10^{10}$
Chess: about 35 moves per turn, six deep	$35^6 \approx 1.8 \times 10^9$

Our arithmetic.



IN PLAIN WORDS

And when the world answers at random, every action branches again into many next states.

Simulation-based search

- Sample-based forward search: simulate episodes of experience from now with the model, and apply model-free RL to them.
- Given a model $\mathcal{M}_\eta$ and a simulation policy π, for each action a, simulate K episodes from the current (real) state s and average:

$$q(s,a) = \frac{1}{K}\sum_{k=1}^{K} G_t^k \approx q_\pi(s,a), \qquad A_t = \operatorname*{argmax}_{a\in\mathcal{A}} q(S_t,a)$$

- Before, we used the model like TD: one step, one prediction. Now we use it like Monte Carlo: a whole imagined episode, to the end.

TRY IT

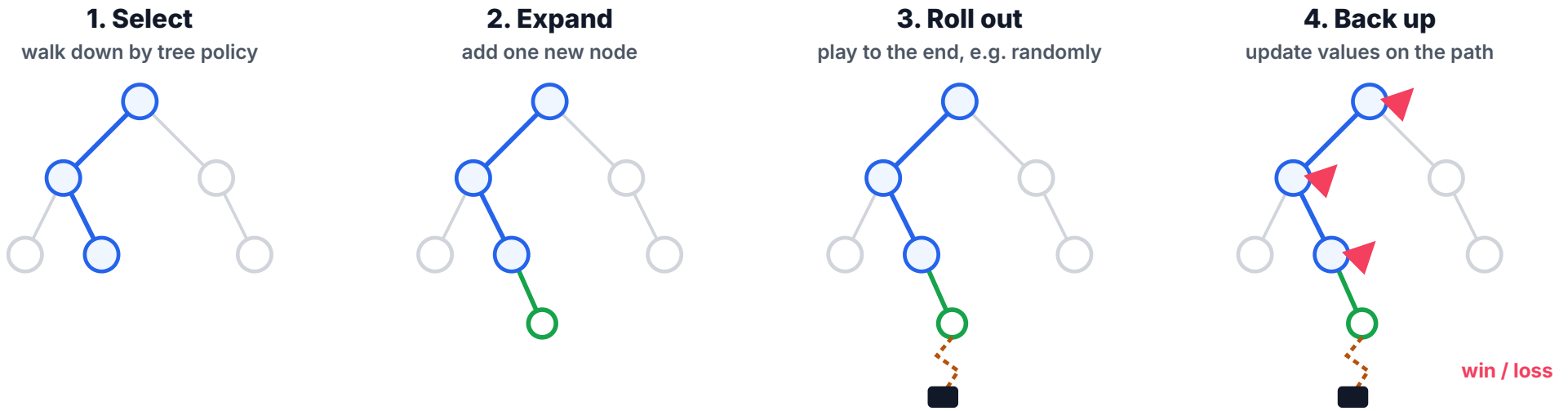
Two moves, $K = 4$ simulated games each, return 1 for a win and 0 otherwise. Move A: 1, 0, 1, 1, so 0.75. Move B: 0, 1, 0, 0, so 0.25. Play A.



IN PLAIN WORDS

The cost: for every action, many imagined episodes. Each is cheap, but they add up.

Monte Carlo tree search



- Build a search tree of visited states and actions, with estimates $q(s, a)$. For each simulated episode: **select** by q until a leaf; **expand** one node; **roll out** to the end with a fixed policy; **back up** the return to every pair on the path.
- Two policies: a **tree policy** that improves during search, and a fixed **rollout policy**, often random.

$$q(s,a) = \frac{1}{N(s,a)} \sum_{k=1}^{K} \sum_{u=t}^{T} \mathbb{I}(S_u^k = s, A_u^k = a)\, G_u^k$$

Upper confidence trees (UCT)

- How should the tree pick actions during a simulation? Always the best so far may miss a better branch; at random wastes the budget.
- UCT treats each node i as a **multi-armed bandit** (Lecture 2) and picks the arm with the highest upper bound:

$$Q(s,a,i) = \frac{1}{N(i,a)} \sum_{k=1}^{N(i,a)} G_k(i,a) + c\sqrt{\frac{\log N(i)}{N(i,a)}}$$

- $N(i,a)$: times arm a was picked at node i; $G_k(i,a)$: the k-th return after it. The simulation policy changes from episode to episode: it learns while it searches.

TRY IT

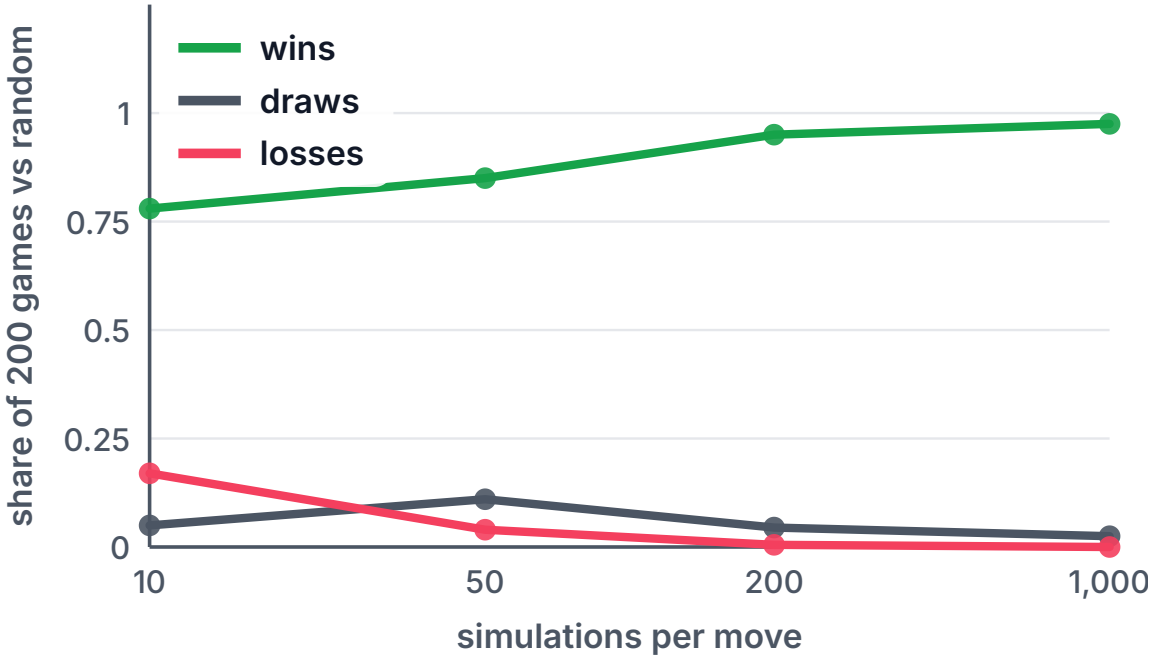
Node visited $N = 100$ times, $c = 1.41$.
Arm A: average 0.6 over 20 tries; arm B: 0.5 over 5.
A: $0.6 + 1.41\sqrt{\ln 100/20} \approx 1.28$.
B: $0.5 + 1.41\sqrt{\ln 100/5} \approx 1.85$.
Try B: it has been tried too rarely to rule out.



IN PLAIN WORDS

Every node is a little room full of slot machines.

UCT plays tic-tac-toe

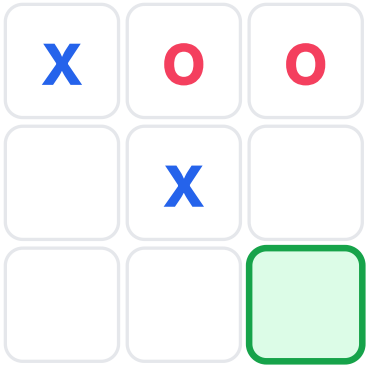


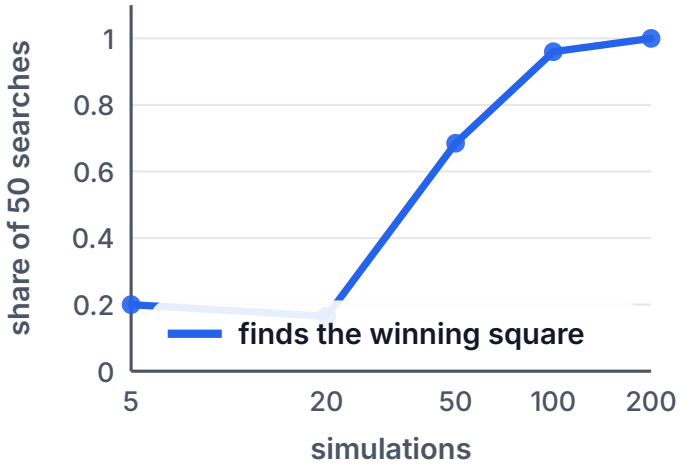
Our runs: UCT with random rollouts, against a random player (it moves first in half the games).

SIMULATIONS	WIN	LOSS
10	78%	17%
50	85%	4%
200	95%	0.5%
1000	97.5%	0%

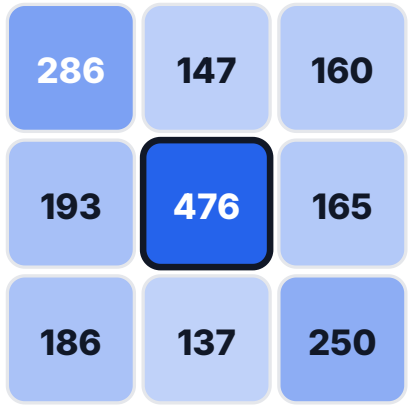
- Flat Monte Carlo with 200 simulations spread evenly won 92.5%; UCT with the same 200 won 95%.
- Two UCT players with 1,000 simulations each: 40 games, 40 draws. Tic-tac-toe is a draw with perfect play.

What the search finds





Our runs: X to move, five empty squares, one wins (green). With 5 simulations: 20%, a pure guess. With 100: 96%; with 200: every time.



Visits per first move from the empty board, 2,000 simulations. Nobody told it about the centre.



IN PLAIN WORDS

Early on, the bonus pushes it to try everything. With enough simulations, the averages win the argument.

Strengths and limits of MCTS

STRENGTHS

why it works

Highly selective, best-first. Evaluates states dynamically (unlike DP). Sampling breaks the curse of dimensionality. Works with black-box models. Anytime, parallelisable.

LIMITS

what to watch

The tree is a table: it doesn't generalise between similar states. A branch it starts ignoring may never be seen. After one decision, most of the tree is thrown away.

- Fix for the first limit: a learned value function inside the tree, so it can generalise (AlphaGo, Part 6).

WHY UCT IS SLIGHTLY STRANGE

UCB was a good idea because exploring cost real reward. Inside the tree nothing is real: a bad simulated move costs only compute. So minimising regret in the simulations isn't quite the goal; the best decision at the root is. (Hay, Russell, Tolpin & Shimony, 2012.)



IN PLAIN WORDS

UCT works very well in practice anyway. Theory and practice are not always on speaking terms.

Check yourself: Part 5

CHECK YOURSELF

11. Name the four steps of one MCTS simulation.

CHECK YOURSELF

12. With $N = 50$ and $c = 1$, arm A has average 0.7 over 10 tries and arm B average 0.4 over 2. Which does UCT pick?

CHECK YOURSELF

13. Why does MCTS usually play the most visited root move rather than the one with the highest average?

CHECK YOURSELF

14. How does a learned value function help MCTS?



Applied Computational Mathematics Lab

PART 6

AlphaGo and AlphaZero

Self-play, a policy-value network, and search.



The game of Go

- More than 2,500 years old; played on 19 × 19 (also 13 × 13 or 9 × 9) boards. Simple rules, complex strategy.
- Black and white place stones alternately; surrounded stones are captured; the player with more territory wins.
- Classic game-tree search, which conquered chess, failed at Go: too many moves (361 to start, around 250 legal moves per turn), and no good way to judge a position.
- In March 2016, DeepMind's **AlphaGo** beat Lee Sedol, one of the world's strongest players, 4–1. It learned from human games, improved by self-play, and searched with MCTS (Silver et al., 2016).



IN PLAIN WORDS

You can't look far ahead in Go, and you can't easily tell who's winning. Search needs both.

THINK ABOUT IT

Does playing Go involve learning to decide in a world whose dynamics and reward are unknown? The rules are known exactly: the hard part is the size of the search.

Self-play

- Let the agent play itself: at every position, compute its best move; for the opponent's move, do the same.
- The only bottleneck is computation: no humans needed.
- Self-play gives a perfectly matched opponent, always: never too weak, never too strong.
- The reward is sparse: win or lose, once, at the end of the game.



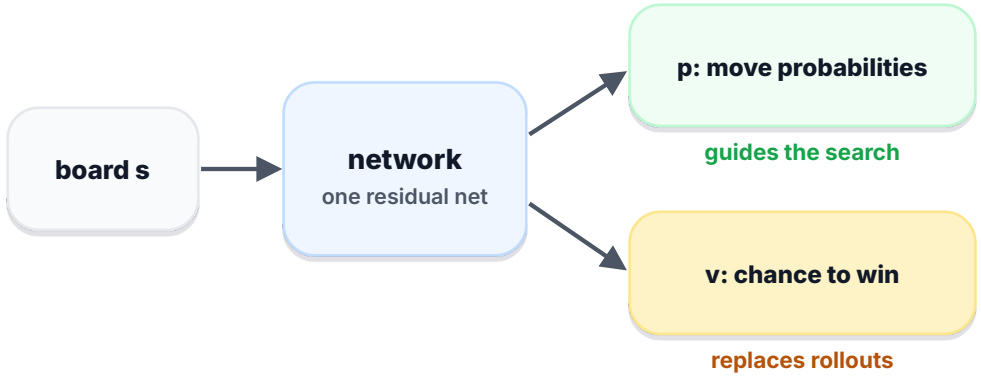
IN PLAIN WORDS

There's no curriculum to design. Your teacher is always exactly one step ahead of you, because your teacher is you.

THE NEXT QUESTION

AlphaGo Zero then asked: what if we drop the human games completely? Just the rules, and self-play.

Search with a policy-value network



$$a = \operatorname*{argmax}_a \left(Q(s,a) + c\, P(s,a) \frac{\sqrt{N(s)}}{1 + N(s,a)} \right)$$

The selection rule of AlphaGo Zero (Silver et al., 2017), a variant of UCT that uses the network's move probabilities P.

- One network, two heads: $(\mathbf{p}, v) = f_\theta(s)$.
- **Select**: walk down using Q plus a bonus scaled by the policy head's prior.
- **Expand** a leaf and **evaluate** it with the value head: no random rollouts.
- **Back up** v along the path (alternating sides: the tree mimics a min-max tree).
- After many simulations, turn the root visit counts into a policy:

$$\pi(a \mid s) \propto N(s,a)^{1/\tau}$$

Training by self-play

- Play a game: at each move run the search, then pick a move from π (proportional to visits early in the game, the most visited move later).
- Train the network on the positions: the policy head learns to predict the search's choice π, the value head to predict the winner z:

$$\ell = (z - v)^2 - \boldsymbol{\pi}^\top \log \mathbf{p} + c\|\theta\|^2$$

- A better network makes a better search; a better search makes better training data; round and round.



IN PLAIN WORDS

It's policy iteration: the search improves the policy, and the network learns to imitate the improvement.

A NUMBER

Each move used 1,600 simulations, about 0.4 seconds of thinking: few for a game this big, because the network makes every simulation count (Silver et al., 2017).

What made it strong?

QUESTION	FINDING (SILVER ET AL., 2017)
Search	The raw network alone: about 3,055 Elo. With tree search: about 5,185 Elo.
Architecture	One residual network for policy and value beat separate networks in their comparisons.
Human data	Starting from zero, it ended stronger than versions that learned from humans first.
Speed	After 3 days (about 4.9 million self-play games) it beat the version that beat Lee Sedol, 100–0.

Numbers as reported in the paper, not our runs.



IN PLAIN WORDS

A policy is good. A policy plus search is better.

ONE MORE STEP

AlphaZero then did the same for chess and shogi: one algorithm, just the rules (Silver et al., 2018). It was given the rules, so its model was perfect; when the model must be learned too, everything about wrong models comes back.

Check yourself: Part 6

CHECK YOURSELF

15. What does each head of the AlphaGo Zero network predict, and what is it trained toward?

CHECK YOURSELF

16. What replaces the random rollouts in AlphaGo Zero's search?

CHECK YOURSELF

17. With $\tau = 1$, the root visits are 60, 30 and 10. What is the search policy?



WRAP-UP

Summary

- **Models** are learned by supervised learning; expectation, stochastic and full models; tables and linear models.
- **Planning** spends compute instead of real interaction; performance is capped by the model.
- **Dyna**: real steps teach the model and the values; imagined steps teach the values. Dyna-Q+ copes with change.
- **Replay** is a non-parametric model: cheap and honest; a parametric model can imagine what never happened.
- **MCTS**: select, expand, roll out, back up; **UCT** puts a bandit in every node.
- **AlphaZero**: a policy-value network inside the search, trained by self-play.

$$R, S' \sim \hat{p}_\eta(\cdot \mid s, a)$$

$$\bar{G} + c\sqrt{\log N(i)/N(i,a)}$$

$$\pi(a \mid s) \propto N(s,a)^{1/\tau}$$

ANSWERS

Answers to check yourself

- **1.** The model is approximate, and the values computed from it are approximate too.
- **2.** At $x = 0$: inside the wall.
- **3.** $\hat{p}(D \mid C) = 3/4$, $\hat{p}(E \mid C) = 1/4$; expected reward $(2 + 2 + 0 + 1)/4 = 1.25$.
- **4.** The reward depends on the task; the transitions can be reused across tasks.
- **5.** Samples only; DP needs the full distribution.
- **6.** Too few samples: both sampled A episodes happened to end with 1.
- **7.** Yes: it can be wrong where you have no data.
- **8.** Real: the direct Q update and the model. Model: the n planning updates.
- **9.** The model knows nothing reliable about unvisited states.
- **10.** Its model said the shortcut was a wall, and it never went back to check; the bonus made Dyna-Q+ re-try old transitions.
- **11.** Select, expand, roll out, back up.
- **12.** A: $0.7 + \sqrt{\ln 50/10} \approx 1.33$; B: $0.4 + \sqrt{\ln 50/2} \approx 1.80$: B.
- **13.** A few lucky playouts can inflate an average; many visits are harder to fool.
- **14.** It evaluates leaves and generalises across similar states, which the table-like tree can’t.
- **15.** Policy head: the search policy π. Value head: the game result z .
- **16.** The value head’s estimate at the leaf.
- **17.** $(0.6, 0.3, 0.1)$.

CREDITS

Sources and credits

- The order of topics follows *Planning and Models* by Matteo Hessel (DeepMind × UCL Reinforcement Learning Lecture Series, 2021) and, for MCTS and AlphaZero, *Lecture 14: Monte Carlo Tree Search* by Emma Brunskill (Stanford CS234, Spring 2024). This deck is not an official DeepMind, UCL or Stanford material.
- The text, figures, examples and simulations were written, drawn and run for ACML. Numbers labelled “our run” come from our own code; examples taken from books or papers are credited where they appear.
- Sutton & Barto (2018), chapter 8: Dyna, the Dyna maze (8.1) and shortcut maze (8.3), re-run by us. Kocsis & Szepesvári (2006), UCT. Silver et al. (2016, 2017, 2018): AlphaGo, AlphaGo Zero, AlphaZero; the Elo and game counts quoted here come from the 2017 paper.
- Companion videos: **ACML Explains**, season 1, episodes 31–37. More from the lab: acml.teddylazebnik.com



Applied Computational Mathematics Lab

END OF LECTURE 9

Next: policy gradients

Lecture 10 forgets values for a while and learns the policy directly: the policy gradient theorem, REINFORCE and actor-critic.

