Applied Computational Mathematics Lab

TEDDY LAZEBNIK'S COURSE · LECTURE 7

Reinforcement Learning

Lecture 7 · Function approximation: why tables are not enough, gradient descent and features, control with approximation, convergence and the deadly triad, batch methods and a first look at deep RL



Teddy Lazebnik · Applied Computational Mathematics Laboratory

BEFORE WE START

Recap, and what this lecture does

- The policy, value function, model and agent state update are all **functions**, and we want to learn them from experience.
- If there are too many states, we need to **approximate**.
- With neural networks this is called **deep reinforcement learning**. The term is fairly new; the combination is decades old.



IN PLAIN WORDS

So far, every value lived in a table: rows for states, columns for actions. The table has no idea that two neighbouring states are related.

IN ONE LINE

The sentence to remember from this lecture: too many states to visit them all.

THIS LECTURE

Predictions

Value functions, including value-based control.

UPCOMING

More

Off-policy learning, approximate DP, policy gradients, model-based RL.

BEFORE WE START

Today's map

PART 1

Why approximate?

Huge problems, generalisation, function classes, and why RL data is unusual.

PART 2

Gradients and linear features

SGD for values; coarse and tile coding; MC and TD with approximation.

PART 3

Control with approximation

Action values with features; a car stuck in a valley.

PART 4

Convergence and divergence

Fixed points of MC and TD, a two-state blow-up, and the deadly triad.

PART 5

Batch methods and deep RL

Least-squares TD, experience replay, neural Q-learning and DQN.

READING

Sutton & Barto (2018)

Chapters 9 and 10 (and 11 for off-policy divergence).



IN PLAIN WORDS

By the end you should be able to write MC and TD updates with features, explain why TD can diverge, and name the parts of DQN.

Applied Computational Mathematics Lab

PART 1

Why approximate?

Too many states to visit them all.



Large-scale reinforcement learning

PROBLEM	STATES
Backgammon	about 10^{20}
Go	10^{170}
Helicopter	continuous
Robots	the real world

- How can we apply our methods for prediction and control?
- A table is sparse (most actions make no sense in most states), huge, and may not fit in memory.



IN PLAIN WORDS

The observable universe has about 10^{80} atoms. Give each Go position its own atom, and you run out of atoms.

EXAMPLE

A robot's sensors are continuous: infinitely many states. Chop them into bins, and the count explodes anyway. We will only ever see a sliver of the states.

Generalising to states you never saw

- A computer game: you keep finding a bag with 99 gold coins; the reward has averaged 99.
- Now you find a bag with 100 coins. What is the reward?
- You just made two hidden assumptions: more gold is better, and the relation is roughly linear, at least nearby.

TABLE

Rows 99 and 100

Visiting 99 a hundred times says nothing about 100. They are just two rows.

FUNCTION

A neighbourhood

Learn $v(\text{coins}) \approx \text{coins}$, and 100 gets about 100 for free.



IN PLAIN WORDS

Mathematically, it isn't guaranteed. Practically, it's excellent: you generalised from states you saw to one you never saw.

A CAVEAT

Ask about a thousand coins, and a room of economists will say "less than a thousand": diminishing returns. Generalisation needs care far from the data.

Value function approximation

- Estimate the values with a function with parameters $\mathbf{w}$:

$$v_{\mathbf{w}}(s) \approx v_\pi(s) \;\; (\text{or } v_*(s)), \qquad q_{\mathbf{w}}(s,a) \approx q_\pi(s,a) \;\; (\text{or } q_*(s,a))$$

- Update $\mathbf{w}$, for example with MC or TD learning, and generalise to unseen states.
- When the environment state is not fully observable, use the agent state $S_t = u_\omega(S_{t-1}, A_{t-1}, O_t)$; in the simplest case $S_t = O_t$.



IN PLAIN WORDS

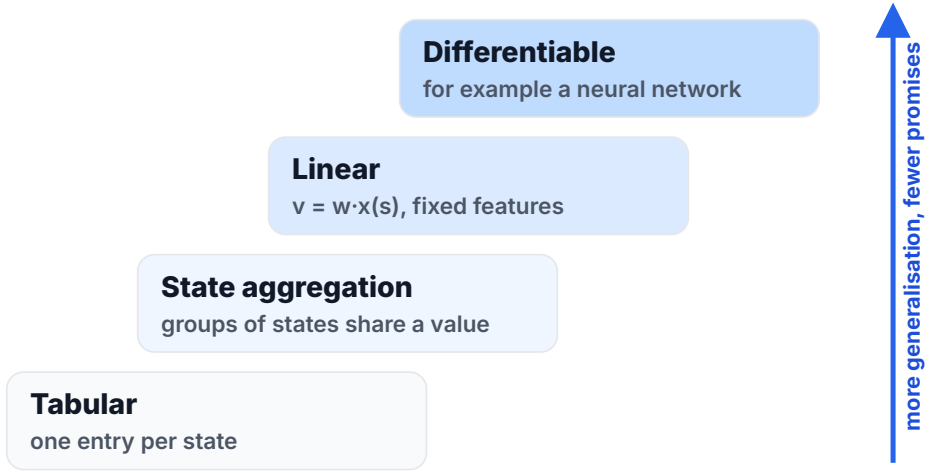
Nothing changes except the representation: Monte Carlo and TD still work. Pick the weights well, and one learned number covers a whole neighbourhood of states.

TRY IT

A table for a million states needs a million numbers. A linear function of 20 features needs 20 weights, and every update moves the values of many states at once.

Classes of function approximation

- **Tabular**: an entry for each state.
- **State aggregation**: partition states (or observations) into a discrete set.
- **Linear**: a fixed feature map $\mathbf{x} : \mathcal{S} \to \mathbb{R}^n$ and $v_{\mathbf{w}}(s) = \mathbf{w}^\top \mathbf{x}(s)$. Tabular and aggregation are special cases.
- **Differentiable**: $v_{\mathbf{w}}(s)$ any differentiable function of $\mathbf{w}$, possibly non-linear; for example a convolutional network on pixels. Another reading: the features are learned, not fixed.

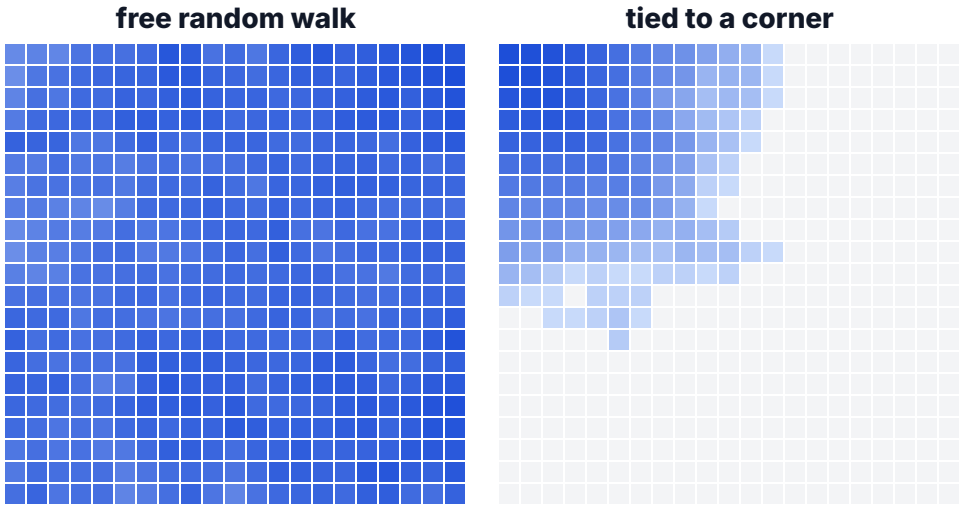


TRY IT

State aggregation in the gold game: 90 to 99 coins become one state. Ten times fewer states, and a bag of 91 and a bag of 99 now look the same: close enough.

Reinforcement learning data is unusual

- Any function approximator could be used, but RL data has special properties:
- Experience is **not i.i.d.**: successive time steps are correlated.
- The agent's policy **affects the data** it receives.
- Regression targets can be **non-stationary**, because of:
  - changing policies (which change the targets and the data);
  - bootstrapping;
  - non-stationary dynamics (for example, other learning agents);
  - a world so large that you're never quite in the same state.



Our run: 20,000 random steps in a 21 × 21 room. Free: 100% of the squares visited. Tied to the top-left corner by a "rubber band": 32%, and never the far quarter.



IN PLAIN WORDS

The bias is in the sampling, not in the room. In supervised learning someone hands you the data; here the learner collects its own.

Which approximator should you choose?

CLASS	THEORY	IN PRACTICE
Tabular	good	does not scale or generalise
Linear	reasonably good	needs good features
Non-linear (e.g. deep nets)	less well understood	scales well; flexible; less reliant on hand-picked features

- It depends on your goals. Deep neural networks often perform quite well, and remain a popular choice.



IN PLAIN WORDS

Every step up the ladder buys power and costs promises. Neural networks: great in practice, no promises at all.

WHY NO PROMISES

A network could in principle be asked to fit a function that is continuous everywhere and differentiable nowhere, such as the Weierstrass function. No general guarantee can cover that.

Check yourself: Part 1

CHECK YOURSELF

1. Why can’t a table generalise from 99 coins to 100?

CHECK YOURSELF

2. Name two reasons RL regression targets are non-stationary.

CHECK YOURSELF

3. Is state aggregation a special case of linear function approximation? What are the features?

CHECK YOURSELF

4. Why did the agent tied to a corner never visit the far quarter of the room?





HINT

Answers are at the end of the deck.

Applied Computational Mathematics Lab

PART 2

Gradients and linear features

Stochastic gradient descent, coarse and tile coding, MC and TD with features.



Gradient descent

- Let $J(\mathbf{w})$ be a differentiable function of the parameters. Its gradient collects the partial derivatives:

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \left(\frac{\partial J(\mathbf{w})}{\partial w_1}, \ldots, \frac{\partial J(\mathbf{w})}{\partial w_n} \right)^\top, \qquad \Delta \mathbf{w} = -\tfrac{1}{2} \alpha \nabla_{\mathbf{w}} J(\mathbf{w})$$

- Goal: minimise J. Method: move $\mathbf{w}$ in the direction of the negative gradient, with step size α.

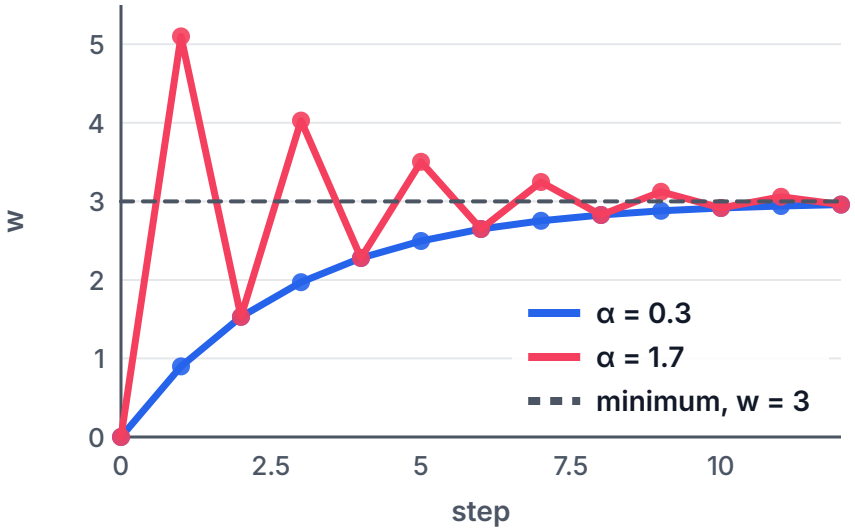


IN PLAIN WORDS

A ball rolling down a hill, a little at a time. Small steps matter: big ones overshoot the valley.

TRY IT

$J(w) = (w-3)^2$, $\alpha = 0.3$, from $w = 0$: $\Delta w = -0.3(0-3) = 0.9$. Then 1.53, 1.97, 2.28, … each step closes 30% of the remaining gap.



Our computation: $J(w) = (w-3)^2$, so $\Delta w = -\alpha(w-3)$, from $w = 0$. The large step overshoots back and forth; above $\alpha = 2$ it would diverge.

Approximating values by SGD

- Goal: find $\mathbf{w}$ that minimises the squared difference between $v_{\mathbf{w}}$ and v_π, over a state distribution d (typically induced by the policy and dynamics):

$$J(\mathbf{w}) = \mathbb{E}_{S\sim d}\left[(v_\pi(S) - v_{\mathbf{w}}(S))^2\right], \qquad \Delta\mathbf{w} = -\tfrac{1}{2}\alpha\nabla_{\mathbf{w}} J(\mathbf{w}) = \alpha\,\mathbb{E}_d\left[(v_\pi(S) - v_{\mathbf{w}}(S))\nabla_{\mathbf{w}} v_{\mathbf{w}}(S)\right]$$

- **Stochastic** gradient descent samples the gradient. The Monte Carlo return G_t is a sample of $v_\pi(S_t)$:

$$\Delta\mathbf{w} = \alpha\big(G_t - v_{\mathbf{w}}(S_t)\big)\nabla_{\mathbf{w}} v_{\mathbf{w}}(S_t)$$

- Short-hand: $\nabla v(S_t)$ for $\nabla_{\mathbf{w}} v_{\mathbf{w}}(S_t)\big|_{\mathbf{w}=\mathbf{w}_t}$.



IN PLAIN WORDS

We don't know the true value, so we sample: step size, times the error, times the gradient. One sample, one small step.

REMEMBER

With Monte Carlo the target is the real return. With TD it is a reward plus our own guess. Remember that: it will bite us in Part 4.

Linear value functions

- Represent a state by a fixed feature vector $\mathbf{x}(s) = (x_1(s), \dots, x_n(s))^\top$: distances of a robot from landmarks, trends in the stock market, piece configurations in chess.
- A linear value function, and its quadratic loss:

$$v_{\mathbf{w}}(s) = \mathbf{w}^\top \mathbf{x}(s) = \sum_{j=1}^{n} x_j(s)\, w_j, \qquad J(\mathbf{w}) = \mathbb{E}_{S\sim d}\left[(v_\pi(S) - \mathbf{w}^\top \mathbf{x}(S))^2\right]$$

- SGD converges on the global optimum. Since $\nabla_{\mathbf{w}} v_{\mathbf{w}}(S_t) = \mathbf{x}_t$: $\Delta\mathbf{w} = \alpha(v_\pi(S_t) - v_{\mathbf{w}}(S_t))\,\mathbf{x}_t$.



IN PLAIN WORDS

Update = step size × prediction error × feature vector. The error says how far the prediction missed; the features say which weights were involved.

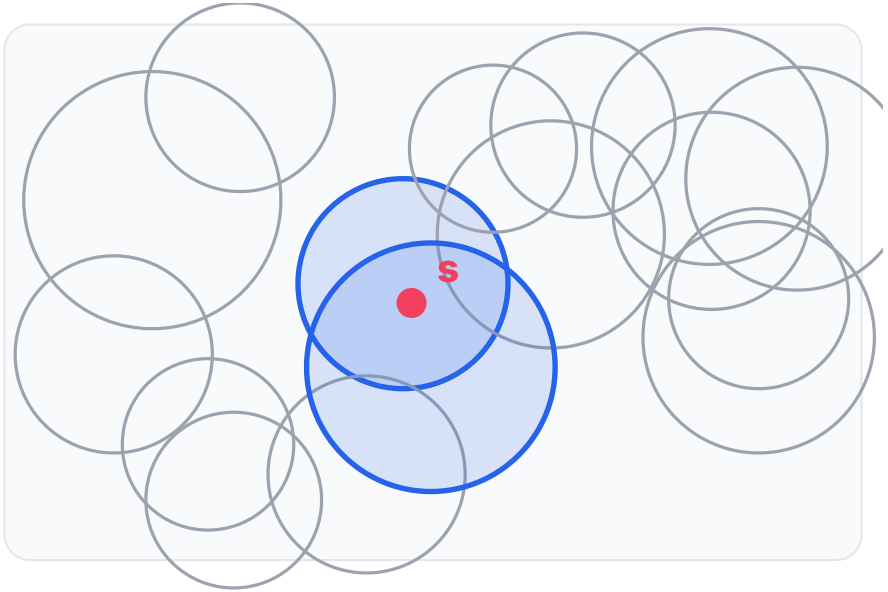
LIFE ISN'T LINEAR

Hooke's law is linear: pull a spring and the force grows in proportion. Up to a point; then the spring deforms. Linear models are a great first approximation, not the last one.

TRY IT

Gold game: one feature, the number of coins. Learn a weight of 1, and a bag of 100 coins is worth 100.

Building features: coarse coding



state s lights up 2 of 16 circles: 2 features are 1

Our drawing, after the coarse-coding idea in Sutton & Barto (2018), section 9.5.

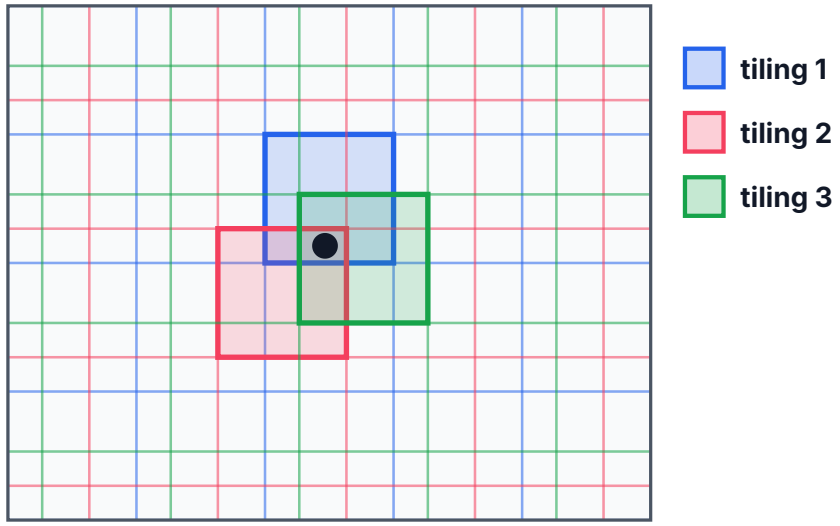
- Scatter circles over the state space. Each circle is one feature: 1 if the state is inside it, 0 if not.
- This gives a large feature vector $\mathbf{x}(s)$; the weights give a value to each feature.
- Learn from one state, and nearby states that share its circles learn too.
- Small circles: narrow, precise generalisation. Big circles: broad and rough. Stretched circles generalise more in one direction.



IN PLAIN WORDS

The circle sizes are a design choice, made before learning starts.

Tile coding



one active tile per tiling: 3 features are 1

Our drawing: three offset tilings over a 2-D state space.

- Tile coding is the tidy version: lay a grid over the space, then another one shifted a little, then another.
- A state falls into exactly one tile per grid: 8 tilings, 8 active features. Always.
- Each grid on its own is coarse; together they pin the state down much more finely.
- The number of active features never changes, so every update costs the same.

TRY IT

Our Mountain Car run (Part 3) uses 8 tilings of 8 × 8 tiles (plus a margin for the offsets): $8 \times 9 \times 9 = 648$ features per action, 8 of them active in any state.

What features can and can’t do

- Coarse features **aggregate** states, so the resulting feature vector (the agent state) is usually **not Markov**.
- That is the common case with function approximation.
- Ask whether a good solution even exists for your features and function class.
- Neural networks tend to be more flexible: they learn their own features.

A WARNING

Fit a line through one point. How many lines are there? Infinitely many; your solver hands you one of them. Technically possible, practically terrible: it runs fine and returns garbage.



IN PLAIN WORDS

Each weight needs data from enough different states. A feature that fires in only one state has nothing to generalise from.

TRY IT

Choose the features, and you choose what the learner can and can’t notice. A robot given only its distance to the door can’t tell left of the door from right of it.

Incremental prediction algorithms

- We can't update toward the true $v_\pi(s)$, so we substitute a target:

$$\begin{aligned}
\text{MC:} \quad & \Delta \mathbf{w}_t = \alpha\big(G_t - v_{\mathbf{w}}(S_t)\big)\nabla_{\mathbf{w}} v_{\mathbf{w}}(S_t) \\
\text{TD:} \quad & \Delta \mathbf{w}_t = \alpha\big(R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1}) - v_{\mathbf{w}}(S_t)\big)\nabla_{\mathbf{w}} v_{\mathbf{w}}(S_t) \\
\text{TD}(\lambda): \quad & \Delta \mathbf{w}_t = \alpha\big(G_t^\lambda - v_{\mathbf{w}}(S_t)\big)\nabla_{\mathbf{w}} v_{\mathbf{w}}(S_t), \quad G_t^\lambda = R_{t+1} + \gamma\big((1-\lambda)v_{\mathbf{w}}(S_{t+1}) + \lambda G_{t+1}^\lambda\big)
\end{aligned}$$

MONTE CARLO

The return is an unbiased sample of $v_\pi(S_t)$: "supervised learning" on the training data $\{(S_0, G_0), \ldots, (S_t, G_t)\}$. Linear MC converges to the global optimum; non-linear MC converges too, perhaps to a local optimum.

TD

The TD target is a biased sample. The training data $\{(S_t, R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1}))\}$ looks like non-stationary regression, but the target depends on our own parameters!

Linear TD, step by step

- For linear TD the update is the TD error times the features:

$$\Delta \mathbf{w}_t = \alpha \underbrace{\big(R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1}) - v_{\mathbf{w}}(S_t)\big)}_{=\delta_t,\ \text{TD error}} \nabla_{\mathbf{w}} v_{\mathbf{w}}(S_t) = \alpha\, \delta_t\, \mathbf{x}_t$$

TRY IT

$\mathbf{x}(s) = (1, 0.5)$, $\mathbf{x}(s') = (0, 1)$, $\mathbf{w} = (2, 2)$: $v(s) = 3$, $v(s') = 2$.
Reward 1, $\gamma = 0.9$, $\alpha = 0.5$.
$\delta = 1 + 1.8 - 3 = -0.2$; $\Delta \mathbf{w} = 0.5 \times (-0.2) \times (1, 0.5) = (-0.1, -0.05)$.
Now $v(s) = 2.875$, and $v(s') = 1.95$: the target moved too.



IN PLAIN WORDS

The update to s also changed s', because they share a weight. That is generalisation, and it is also why TD can chase its own tail.

DIFFERENT FROM SUPERVISED LEARNING

Supervised learning assumes the labels don't care about your predictions. TD's labels do.

- With Monte Carlo the target would be the return G_t: it doesn't move when $\mathbf{w}$ changes.

Check yourself: Part 2

CHECK YOURSELF

5. $J(w) = (w-1)^2$, $w = 5$, $\alpha = 0.25$. What is w after one step of $\Delta w = -\frac{1}{2}\alpha J'(w)$?

CHECK YOURSELF

6. With 10 tilings, how many features are active in any state?

CHECK YOURSELF

7. Linear MC: $\mathbf{x} = (1, 2)$, $\mathbf{w} = (0, 1)$, return 4, $\alpha = 0.1$. New $\mathbf{w}$?

CHECK YOURSELF

8. Why is the TD update not plain supervised learning?



Applied Computational Mathematics Lab

PART 3

Control with approximation

Action values with features, and a car stuck in a valley.



Control with value function approximation

- Generalised policy iteration again:

STEP	WITH APPROXIMATION
Policy evaluation	approximate: $q_{\mathbf{w}} \approx q_\pi$
Policy improvement	for example ε-greedy on $q_{\mathbf{w}}$

- With linear state-action features $q_{\mathbf{w}}(s,a) = \mathbf{x}(s,a)^\top \mathbf{w}$, the SGD update is

$$\Delta \mathbf{w} = \alpha\big(q_\pi(s,a) - q_{\mathbf{w}}(s,a)\big)\nabla_{\mathbf{w}} q_{\mathbf{w}}(s,a) = \alpha\big(q_\pi(s,a) - q_{\mathbf{w}}(s,a)\big)\,\mathbf{x}(s,a)$$

- In practice q_π is replaced by a target, for example SARSA's $R_{t+1} + \gamma q_{\mathbf{w}}(S_{t+1}, A_{t+1})$. SARSA is TD on state-action pairs, so it inherits TD's properties.



IN PLAIN WORDS

With approximation, control reaches the best your approximator can do. A perfect approximator gets you the optimum; anything less, and it doesn't.

FROM CLASS

That's why, from linear functions on, I stop promising you the optimum. But the best your approximator can do is often very, very good.

Action in, or action out?

- **Action in**: $q_{\mathbf{w}}(s, a) = \mathbf{w}^\top \mathbf{x}(s, a)$. One weight vector, features of the pair.
- **Action out**: $q_{\mathbf{w}}(s) = \mathbf{W}\mathbf{x}(s)$, one output per action, $q_{\mathbf{w}}(s, a) = q_{\mathbf{w}}(s)[a] = \mathbf{x}(s)^\top \mathbf{w}_a$. Only the chosen action's row moves:

$$\Delta \mathbf{w}_a = \alpha\big(q_\pi(s, a) - q_{\mathbf{w}}(s, a)\big)\,\mathbf{x}(s), \qquad \Delta \mathbf{w}_b = 0 \;\; \forall b \neq a$$

- Equivalently $\Delta \mathbf{W} = \alpha(q_\pi(s, a) - q_{\mathbf{w}}(s, a))\,\mathbf{i}_a \mathbf{x}(s)^\top$, with $\mathbf{i}_a$ the one-hot vector of a.

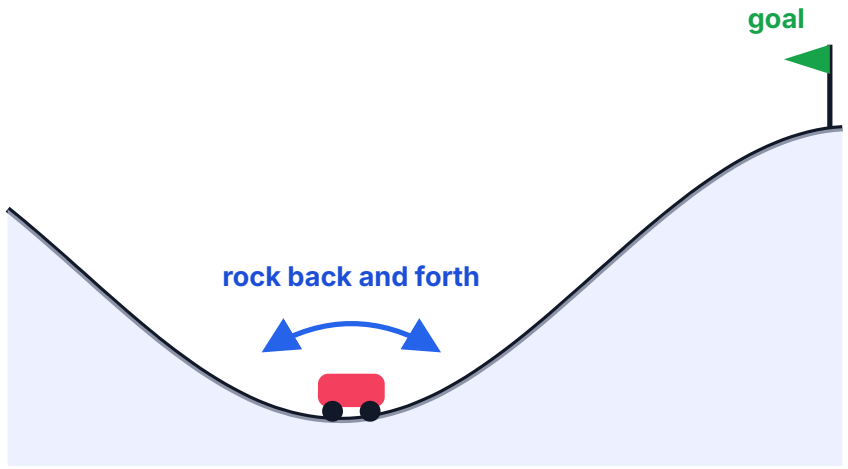
	ACTION IN	ACTION OUT
shares	the weights	the features
continuous actions	easier (later lecture)	hard
small discrete actions	works	common (e.g. DQN)



IN PLAIN WORDS

Which is better in general? Unclear. One reuses the same weights, the other the same features.

Example: Mountain Car



position from −1.2 to 0.5; the engine is too weak to drive straight up

Example 10.1 of Sutton & Barto (2018). Our drawing.

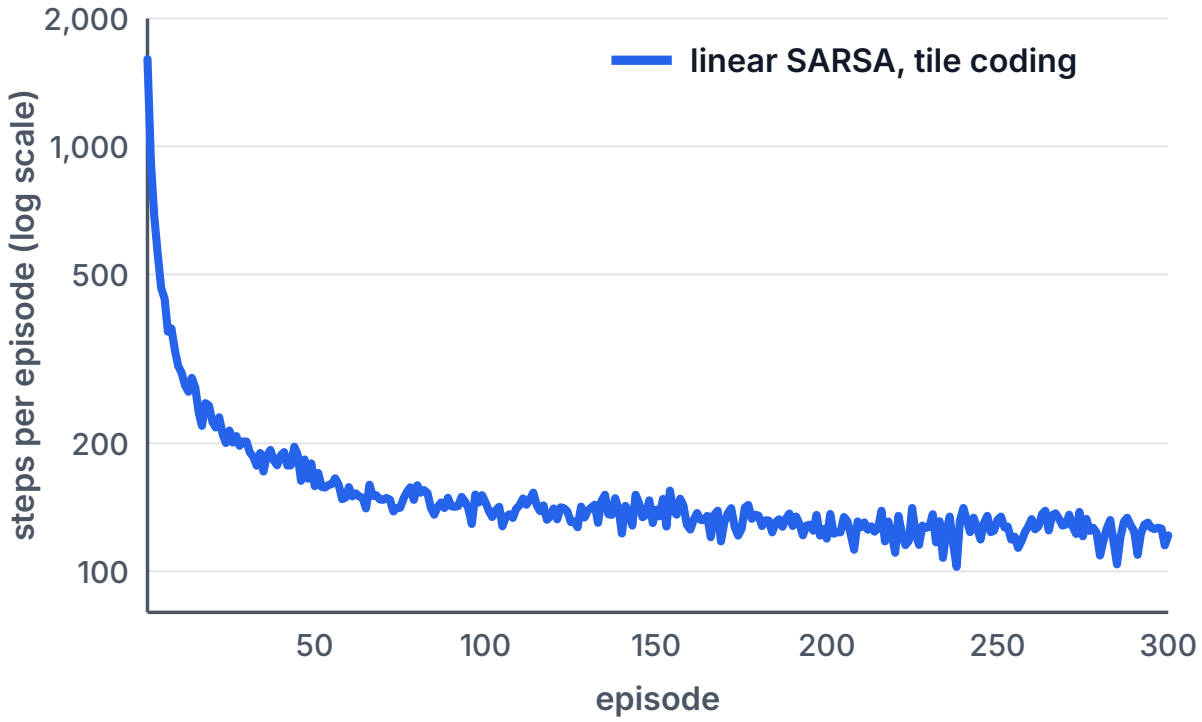
- A weak car, stuck in a valley. It can’t climb the right hill directly: it has to rock back and forth to build up speed.
- State: position and velocity, two continuous numbers. A table can’t hold that.
- Actions: full throttle backward, zero, or forward. Reward −1 per step until the top on the right.
- Our run: linear SARSA with tile coding (8 tilings of 8 × 8), averaged over 10 runs, start near the bottom with zero speed.



IN PLAIN WORDS

All the weights start at 0. Every step costs −1, so 0 looks optimistic: the car explores on its own, without ε.

Mountain Car: learning



Our runs: 10 runs averaged. The first episode takes about 1,606 steps; the last 50 average 125.7.



IN PLAIN WORDS

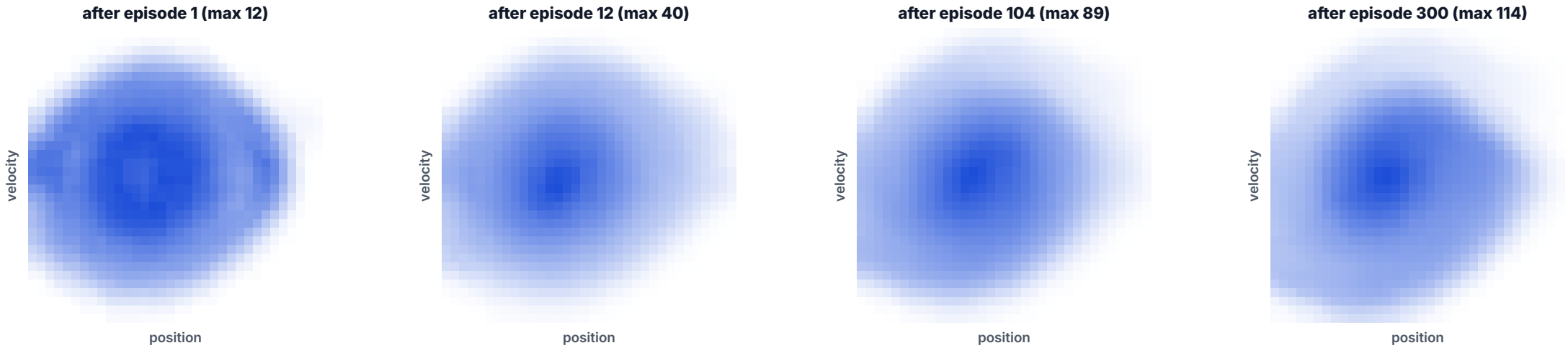
One global line would be terrible here. Many small tiles, each with its own weight, do the job.

DETAIL

Step size $\alpha = 0.5/8$: the usual choice of splitting the step over the 8 active tiles, so that one update moves the value by about half the error.

PART 3 · CONTROL

The learned cost-to-go



Our run (run 1): $-\max_a q_{\mathbf{w}}(s,a)$ over position (left to right, −1.2 to 0.5) and velocity (bottom to top). Darker = more steps still to go; each map is scaled to its own maximum.

- After one long first episode, values have already moved over much of the space: every tile spreads its update to its whole neighbourhood.
- By the end, the cost peaks around the bottom of the valley at low speed, and drops wherever the car already has enough swing to escape.



IN PLAIN WORDS

The dark centre is the car standing still at the bottom: the most steps to go. Speed in either direction, or height on either slope, lowers the cost.

Check yourself: Part 3

CHECK YOURSELF

9. With action-out features and 3 actions, which weights change after an update for action 2?

CHECK YOURSELF

10. Why does Mountain Car explore without ε when the weights start at 0?

CHECK YOURSELF

11. Why can’t a single straight line (one weight per input) represent Mountain Car’s cost-to-go well?



Applied Computational Mathematics Lab

PART 4

Convergence and divergence

When incremental algorithms converge, and when they blow up.



Convergence questions

- When do incremental prediction algorithms converge?
  - when using bootstrapping (TD)?
  - when using (for example linear) function approximation?
  - when using off-policy learning?
- Ideally, algorithms that converge in all cases. Otherwise, we want to understand when they do, and when they don't.



IN PLAIN WORDS

The only place in your academic life where anyone ever promised you a global optimum: tables. Linear sits in the middle; neural networks promise nothing.

A COMFORT

Any smooth function looks linear up close. So linear methods give decent answers, especially when the states are sampled densely.

Where Monte Carlo converges

- With linear functions and suitably decaying step sizes, MC converges to

$$\mathbf{w}_{\mathrm{MC}} = \operatorname*{argmin}_{\mathbf{w}} \mathbb{E}_\pi\left[(G_t - v_{\mathbf{w}}(S_t))^2\right] = \mathbb{E}_\pi[\mathbf{x}_t \mathbf{x}_t^\top]^{-1}\, \mathbb{E}_\pi[G_t \mathbf{x}_t]$$

- Verifying the fixed point (the gradient is zero):

$$\mathbb{E}\left[(G_t - \mathbf{x}_t^\top \mathbf{w}_{\mathrm{MC}})\, \mathbf{x}_t\right] = 0 \implies \mathbb{E}[\mathbf{x}_t \mathbf{x}_t^\top]\, \mathbf{w}_{\mathrm{MC}} = \mathbb{E}[G_t \mathbf{x}_t]$$

- The agent state does not have to be Markov: the fixed point depends only on the observed data and the features.



IN PLAIN WORDS

Monte Carlo sees whole episodes: it has the global picture. That makes it the safe default, when you can afford to wait.

LOOK FAMILIAR?

This is ordinary least squares: regress the returns on the features.

Where TD converges

- With linear functions, in continuing problems with fixed $\gamma < 1$ and decaying α_t, TD converges to $\mathbf{w}_{\mathrm{TD}}$. Setting the expected update to zero (with α_t not correlated with $R_{t+1}, \mathbf{x}_t, \mathbf{x}_{t+1}$):

$$\mathbb{E}[\Delta \mathbf{w}_{\mathrm{TD}}] = 0 = \mathbb{E}\left[\alpha_t (R_{t+1} + \gamma \mathbf{x}_{t+1}^\top \mathbf{w}_{\mathrm{TD}} - \mathbf{x}_t^\top \mathbf{w}_{\mathrm{TD}})\, \mathbf{x}_t\right]$$
$$\Longrightarrow \quad \mathbf{w}_{\mathrm{TD}} = \mathbb{E}\left[\mathbf{x}_t (\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^\top\right]^{-1} \mathbb{E}[R_{t+1} \mathbf{x}_t]$$

THEOREM

This differs from the MC solution. With the value error $\overline{\mathrm{VE}}(\mathbf{w}) = \sum_s d_\pi(s)(v_\pi(s) - v_{\mathbf{w}}(s))^2$, which MC minimises:

$$\overline{\mathrm{VE}}(\mathbf{w}_{\mathrm{TD}}) \leq \frac{1}{1-\gamma}\, \overline{\mathrm{VE}}(\mathbf{w}_{\mathrm{MC}}) = \frac{1}{1-\gamma} \min_{\mathbf{w}} \overline{\mathrm{VE}}(\mathbf{w})$$



IN PLAIN WORDS

Typically the asymptotic MC solution is preferred: the smallest prediction error. But TD often converges faster, especially with intermediate λ or n.

TRY IT

With $\gamma = 0.9$ the bound allows TD's value error to be up to 10 times the best possible. It is a worst case; in practice TD is usually much closer.

TD is not a gradient

- The TD update is not a true gradient update:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha\big(r + \gamma v_{\mathbf{w}}(s') - v_{\mathbf{w}}(s)\big)\nabla v_{\mathbf{w}}(s)$$

- That's okay: it is a **stochastic approximation** update, a broader class than SGD.
- SGD always converges, with bounded noise, decaying step sizes, stationarity, … TD does not always converge.



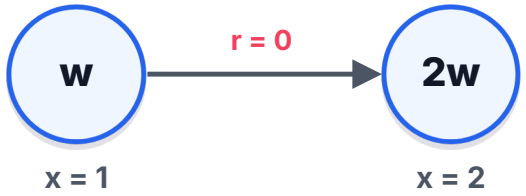
IN PLAIN WORDS

Remember the ball rolling downhill? For TD, the hill moves every time the ball does.

FROM CLASS

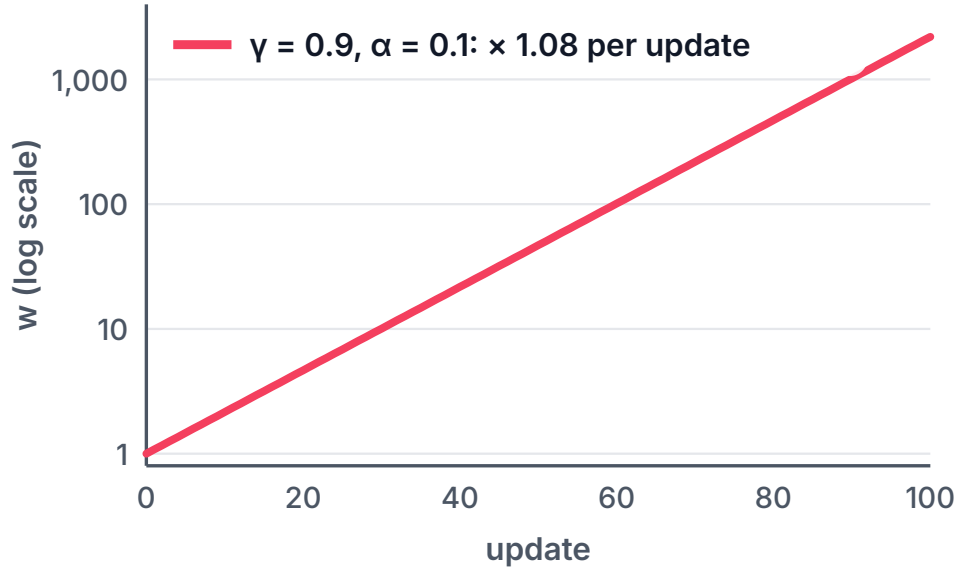
A knight galloping up a hill in thick fog: at the top, is there a higher hill or a valley? TD is that knight; it sees one step ahead. Monte Carlo already rode the whole way.

A two-state blow-up



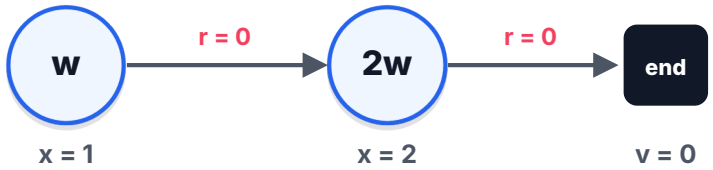
$$
\begin{aligned}
w_{t+1} &= w_t + \alpha_t\big(r + \gamma v(s') - v(s)\big)x(s) \\
&= w_t + \alpha_t(0 + 2\gamma w_t - w_t) = \big(1 + \alpha_t(2\gamma - 1)\big)w_t
\end{aligned}
$$

- One weight w: the values are w and $2w$. The reward is 0, so the true values are 0.
- Use TD only on this one transition. If $w_t > 0$ and $\gamma > \frac{1}{2}$, then $w_{t+1} > w_t$: $w \to \infty$.



Our computation: after 100 updates $w \approx 2,200$. Nothing in the data is wrong: the rewards are all 0.

What saves it

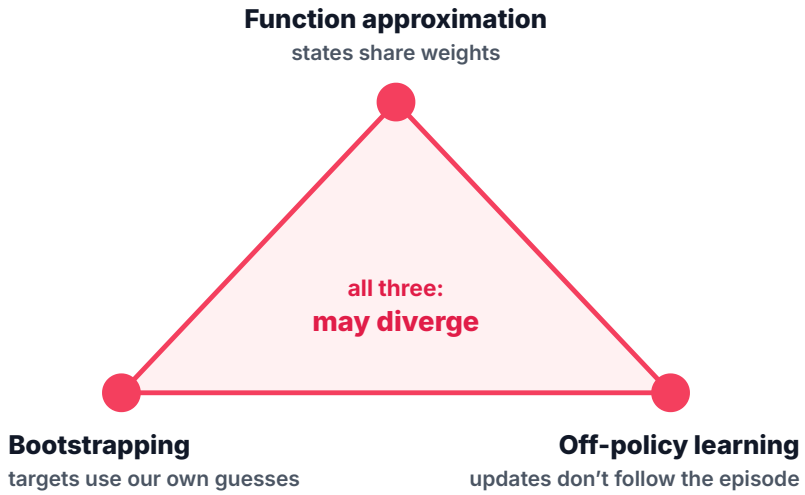


A third state, the end (value 0), after the second state.

FIX	EFFECT ON w	RESULT
Update both states on-policy, along the episode	$\times (1 + \alpha(2\gamma - 5)) = 0.68$ per episode	converges to 0
Tabular features (one weight per state)	plain regression	no divergence
λ-return on the first state	$\times (1 + \alpha(2\gamma(1 - \lambda) - 1))$	converges if $\lambda > 1 - \frac{1}{2\gamma} = 0.444$

- On-policy: the second state's update is $\alpha(0 + \gamma \cdot 0 - 2w) \times x(s') = -4\alpha w$, which pulls w down hard. At $\gamma = 0.9$, $\alpha = 0.1$: about 0.7 per episode; almost nothing is left after 20 episodes.
- λ-return at $\gamma = 0.9$: the factor $2\gamma(1 - \lambda) - 1$ is 0.8 at $\lambda = 0$, 0.35 at 0.25, -0.1 at 0.5: safe from about $4/9$.

The deadly triad



- Algorithms that combine **bootstrapping**, **off-policy learning** and **function approximation** may diverge.
- This is sometimes called the deadly triad. Any two of them are fine.
- In the blow-up: states shared a weight, the target used our own guess, and we updated one state over and over instead of following the episode.



IN PLAIN WORDS

A fire needs heat, fuel and oxygen. Take one away, and nothing burns.

WATCH OUT

Off-policy shows up even when you don't plan it: replay old data, and you're off-policy.

Residual Bellman updates

- TD ignores how $v_{\mathbf{w}}(S_{t+1})$ depends on $\mathbf{w}$: $\Delta \mathbf{w}_t = \alpha \delta_t \nabla v_{\mathbf{w}}(S_t)$.
- Alternative: a true gradient of the **Bellman residual**:

$$\text{loss: } \mathbb{E}[\delta_t^2], \qquad \Delta \mathbf{w}_t = \alpha\, \delta_t\, \nabla_{\mathbf{w}} \big(v_{\mathbf{w}}(S_t) - \gamma v_{\mathbf{w}}(S_{t+1})\big)$$

- This tends to work worse in practice: Bellman residuals **smooth**, whereas TD methods **predict**; smoothed values may lead to suboptimal decisions.
- Minimising the Bellman **error** $\mathbb{E}[\delta_t]^2$ instead needs a second, independent sample S'_{t+1} of the next state, so it can't be used online.



IN PLAIN WORDS

Being a true gradient is not the same as being a good predictor.

WHY A SECOND SAMPLE IS HARD

Two independent next states from the same state means resetting the world to that state and trying again. A simulator can; a real robot usually can't.

PART 4 · CONVERGENCE

When does it converge?

	ALGORITHM	TABLE	LINEAR	NON-LINEAR
On-policy	MC	yes	yes	yes
	TD	yes	yes	**no**
Off-policy	MC	yes	yes	yes
	TD	yes	**no**	**no**

“No” means not guaranteed: it may still work, but it can diverge.

- Control: tabular algorithms such as Q-learning extend to function approximation (for example DQN), but the theory of control with approximation is not fully developed.
- Often we prefer **tracking** to convergence: keep adapting the policy as the world changes.



IN PLAIN WORDS

If you remember one table from this lecture, let it be this one.

FROM CLASS

Monte Carlo needs episodes. If your task never ends, define one anyway. In a traffic-light project I worked on, an episode was one day at one junction: the traffic resets overnight. Imperfect, but usable.

Check yourself: Part 4

CHECK YOURSELF

12. In the two-state example with $\gamma = 0.6$ and $\alpha = 0.5$, by what factor does w change per update? Does it diverge?

CHECK YOURSELF

13. Name the three ingredients of the deadly triad. Which one does experience replay add?

CHECK YOURSELF

14. Which converges with linear features off-policy: MC or TD?

CHECK YOURSELF

15. Why does minimising $\mathbb{E}[\delta]^2$ need two samples of the next state?



Applied Computational Mathematics Lab

PART 5

Batch methods and deep RL

Squeeze more out of every sample. Then DQN.



Least-squares TD

- Gradient descent is simple and appealing, but not sample efficient. Batch methods find the best-fitting value function for a given set of past experience.
- Recall TD's fixed point, $\mathbb{E}[(R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1}) - v_{\mathbf{w}}(S_t))\mathbf{x}_t] = 0$. Solve its empirical version directly:

$$\frac{1}{t}\sum_{i=0}^{t}\big(R_{i+1} + \gamma v_{\mathbf{w}}(S_{i+1}) - v_{\mathbf{w}}(S_i)\big)\mathbf{x}_i = 0$$

$$\Longrightarrow \quad \mathbf{w}_{\mathrm{LSTD}} = \Big(\sum_{i=0}^{t}\mathbf{x}_i(\mathbf{x}_i - \gamma\mathbf{x}_{i+1})^\top\Big)^{-1}\sum_{i=0}^{t}R_{i+1}\mathbf{x}_i$$

- This is **least-squares TD** (LSTD): no step size, no passes over the data; one linear solve.



IN PLAIN WORDS

Squeeze everything out of the data you already have.

TRY IT

One feature; two transitions, $\gamma = 0.9$: $x = 1 \to x = 2$ with reward 0, then $x = 2 \to$ end with reward 1. $A = 1(1 - 1.8) + 2(2 - 0) = 3.2$, $b = 0 + 1 \times 2 = 2$. $w = b/A = 0.625$: values 0.625 and 1.25. The true values are 0.9 and 1: one feature can't fit both.

LSTD in practice

$$\mathbf{w}_t = \underbrace{\left(\sum_{i=0}^{t} \mathbf{x}_i(\mathbf{x}_i - \gamma\mathbf{x}_{i+1})^\top\right)^{-1}}_{=\mathbf{A}_t^{-1}} \underbrace{\sum_{i=0}^{t} R_{i+1}\mathbf{x}_i}_{=\mathbf{b}_t}$$

- Update $\mathbf{A}_t$ and $\mathbf{b}_t$ incrementally online: $\mathbf{A}_{t+1} = \mathbf{A}_t + \mathbf{x}_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^\top$, $\mathbf{b}_{t+1} = \mathbf{b}_t + R_{t+1}\mathbf{x}_t$, then invert: $O(n^3)$.
- Faster, $O(n^2)$: update $\mathbf{A}^{-1}$ directly with the Sherman–Morrison formula:

$$\mathbf{A}_{t+1}^{-1} = \mathbf{A}_t^{-1} - \frac{\mathbf{A}_t^{-1}\mathbf{x}_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^\top\mathbf{A}_t^{-1}}{1 + (\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^\top\mathbf{A}_t^{-1}\mathbf{x}_t}$$

METHOD	COMPUTE PER STEP
TD	$O(n)$
LSTD, Sherman–Morrison	$O(n^2)$
LSTD, naive	$O(n^3)$

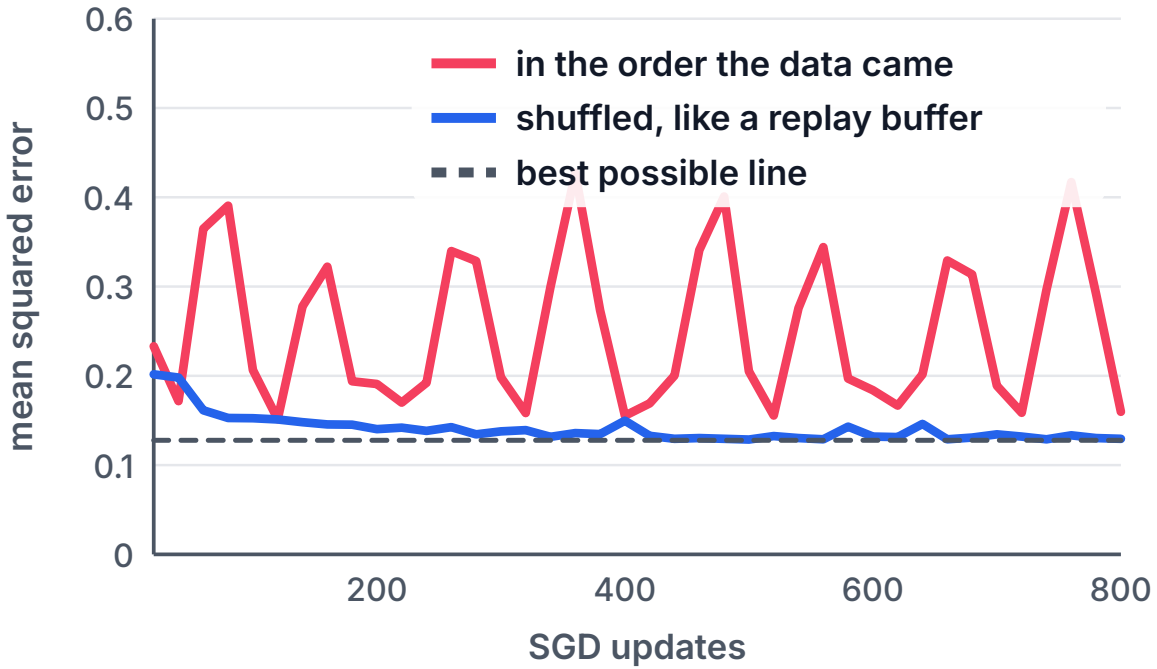
- In the limit, LSTD and TD converge to the same fixed point.
- Extensions: LSTD(λ) for multi-step returns; LSTDQ for action values; least-squares policy iteration (LSPI).

Experience replay

EXPERIENCE REPLAY

1. Store experience $\mathcal{D} = \{S_0, A_0, R_1, S_1, \ldots, S_t\}$.
2. Sample transition(s), for example $(S_n, A_n, R_{n+1}, S_{n+1})$ with $n \leq t$.
3. Apply an SGD update: $\Delta \mathbf{w} = \alpha(R_{n+1} + \gamma v_{\mathbf{w}}(S_{n+1}) - v_{\mathbf{w}}(S_n))\nabla_{\mathbf{w}} v_{\mathbf{w}}(S_n)$.
4. Repeat; old data is re-used.

- This is also a form of batch learning. Beware: the data is off-policy if the policy changes.



Our run: fitting a line to a curve by SGD, 200 points, 4 passes. Fed in order (as along a trajectory), the line swings back and forth; shuffled, it settles near the best fit.

Deep reinforcement learning

TRANSFERS EASILY

to deep networks

TD and MC; double learning (e.g. double Q-learning); experience replay; ...

DOESN'T TRANSFER EASILY

to deep networks

UCB (counts per state); least-squares TD and MC (matrix inverses).



IN PLAIN WORDS

Neural networks: the cool kid on the block. They learn their own features, which is exactly what linear methods lacked.

- A deep network is a differentiable function approximator: $v_{\mathbf{w}}$ or $q_{\mathbf{w}}$, trained with the gradient updates above, where the gradient comes from backpropagation.
- More in Lecture 8.

Example: neural Q-learning

- Online neural Q-learning may include:
  - a network $O_t \mapsto q_{\mathbf{w}}$ (action-out);
  - an exploration policy, $\pi_t = \varepsilon\text{-greedy}(q_t)$, then $A_t \sim \pi_t$;
  - a Q-learning weight update;
  - an optimiser (SGD, RMSProp, Adam).

$$\Delta \mathbf{w} \propto \big(R_{t+1} + \gamma \max_a q_{\mathbf{w}}(S_{t+1}, a) - q_{\mathbf{w}}(S_t, A_t)\big) \nabla_{\mathbf{w}} q_{\mathbf{w}}(S_t, A_t)$$

- Often implemented as a "loss" with a **stop-gradient** $[\![\cdot]\!]$ on the target:

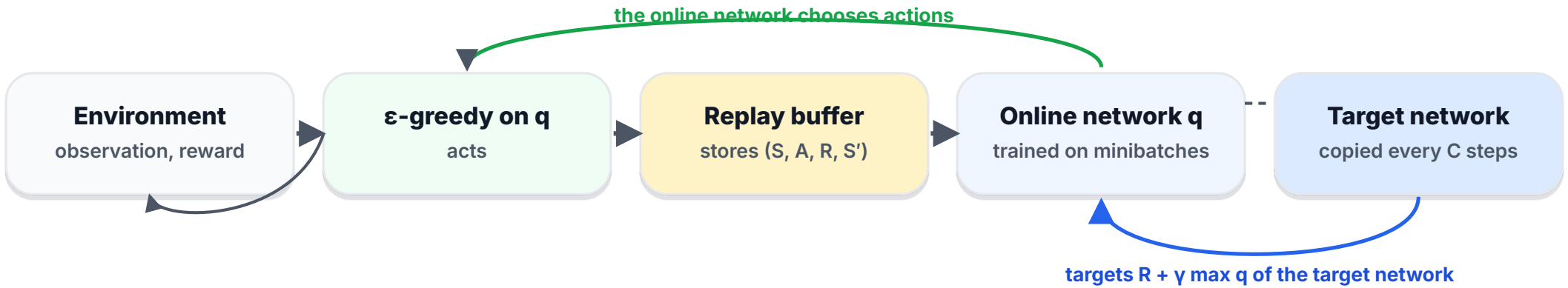
$$L(\mathbf{w}) = \tfrac{1}{2}\big([\![R_{t+1} + \gamma \max_a q_{\mathbf{w}}(S_{t+1}, a)]\!] - q_{\mathbf{w}}(S_t, A_t)\big)^2$$



IN PLAIN WORDS

$L(\mathbf{w})$ is not a real loss: it just happens to have the right (semi-)gradient. Without the stop-gradient you'd get the residual update of Part 4.

Example: DQN



- DQN (Mnih et al., 2013, 2015): a network $O_t \mapsto q_{\mathbf{w}}$, ε-greedy exploration, a **replay buffer** of transitions $(S_i, A_i, R_{i+1}, S_{i+1})$, and **target network** parameters $\mathbf{w}^-$, copied from $\mathbf{w}$ occasionally (e.g. every 10,000 steps).

$$\Delta \mathbf{w} = \left(R_{i+1} + \gamma \max_a q_{\mathbf{w}^-}(S_{i+1}, a) - q_{\mathbf{w}}(S_i, A_i)\right) \nabla_{\mathbf{w}} q_{\mathbf{w}}(S_i, A_i)$$



IN PLAIN WORDS

Replay and target networks make RL look more like supervised learning. Neither is strictly necessary, but they helped for DQN: “deep-learning-aware RL”.

WRAP-UP

Summary

- Too many states to visit them all: approximate, and **generalise**. Tables, aggregation, linear features, neural networks.
- RL data is correlated, policy-dependent and non-stationary.
- Update = step × error × gradient (features, for linear). **MC** targets returns; **TD** targets its own guesses.
- **Linear MC** finds the least-squares fit; **linear TD** a nearby fixed point, within $1/(1-\gamma)$ of the best value error.
- **Deadly triad**: approximation + bootstrapping + off-policy may diverge.
- **LSTD** and **replay** squeeze more out of data; **DQN** combines replay and a target network.

$$\Delta \mathbf{w} = \alpha \left(\text{target} - v_{\mathbf{w}}(S_t)\right) \nabla_{\mathbf{w}} v_{\mathbf{w}}(S_t)$$

$$\mathbf{w}_{\text{TD}} = \mathbb{E}[\mathbf{x}_t(\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^\top]^{-1} \mathbb{E}[R_{t+1}\mathbf{x}_t]$$

ANSWERS

Answers to check yourself

- **1.** Its rows are unrelated: nothing ties the entry for 100 to the entry for 99.
- **2.** Any two of: the policy changes; bootstrapping; changing dynamics; a world too large to revisit.
- **3.** Yes: one-hot features, one per group.
- **4.** The band pulls it back; its own behaviour shapes its data.
- **5.** $\Delta w = -0.125 \times 2(5 - 1) = -1$, so $w = 4$.
- **6.** 10, one per tiling.
- **7.** $v = 2$, error 2: $\mathbf{w} = (0, 1) + 0.2(1, 2) = (0.2, 1.4)$.
- **8.** The target depends on the weights being learned.
- **9.** Only row 2, $\mathbf{w}_2$.
- **10.** Every step costs −1, so initial values of 0 are optimistic: untried actions look better.
- **11.** The cost-to-go peaks in the middle and falls off in every direction; a single plane can't bend like that.
- **12.** $1 + 0.5(1.2 - 1) = 1.1$: yes, it diverges.
- **13.** Approximation, bootstrapping, off-policy; replay adds off-policy data.
- **14.** MC.
- **15.** $\mathbb{E}[\delta]^2$ is a squared expectation; using one sample twice gives $\mathbb{E}[\delta^2]$ instead, which includes the variance.

CREDITS

Sources and credits

- The order of topics follows *Lecture 7: Function Approximation in Reinforcement Learning* of the DeepMind × UCL Reinforcement Learning Lecture Series (2021), by Hado van Hasselt. This deck is not an official DeepMind or UCL 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), chapters 9–11: coarse and tile coding, Mountain Car (10.1, re-run by us), the deadly triad. Tsitsiklis & Van Roy (1997) for the TD fixed-point bound; Bradtke & Barto (1996) for LSTD; Mnih et al. (2013, 2015) for DQN.
- Companion videos: **ACML Explains**, season 1, episodes 24–26. More from the lab: acml.teddylazebnik.com



Applied Computational Mathematics Lab

END OF LECTURE 7

Next: deep reinforcement learning

Lecture 8 looks inside deep RL: neural networks as function approximators, DQN in detail, and the tricks that make it work.

