Applied Computational Mathematics Lab

TEDDY LAZEBNIK'S COURSE · LECTURE 8

Reinforcement Learning

Lecture 8 · Deep reinforcement learning: neural networks as value functions, automatic differentiation, deep Q-learning, making RL friendly to deep learning, and making deep learning fit RL



Teddy Lazebnik · Applied Computational Mathematics Laboratory

BEFORE WE START

Recap: value function approximation

- Tabular RL does not scale to large problems: too many states to store, and too slow to learn each one separately. We must **generalise** across states.
- Estimate values (or policies) approximately:
  - map states s onto a feature representation $\phi(s)$;
  - map features to values with a parametrised function $v_\theta(\phi)$;
  - update θ so that $v_\theta(\phi(s)) \approx v_\pi(s)$.
- From now on the parameters are called θ.



IN PLAIN WORDS

All those lovely promises from tables, the global optimum and all, need every state and action visited infinitely often. In practice, you won't.

FUNCTION CLASS	PRACTICE	THEORY
Table	doesn't scale	strong
Linear	good near the data	reasonable
Neural network	great	very little

BEFORE WE START

Today's map

PART 1

Deep value functions

Networks as approximators, computational graphs, autodiff, deep Q-learning.

PART 2

Deep-learning-aware RL

Correlated data, the deadly triad in practice, target networks, double DQN, prioritised replay, multi-step targets.

PART 3

RL-aware deep learning

Architectures for agents, dueling networks, capacity, and generalisation.

EXAMPLES

Our own runs

A tiny network fitting a curve, a gradient by hand, tic-tac-toe, and a replay buffer of dogs, cats and wolves.

READING

Papers

Mnih et al. (2015); van Hasselt et al. (2016, 2018); Schaul et al. (2016); Wang et al. (2016).

CODE

JAX

NumPy, plus automatic gradients, plus compilation for fast hardware.



IN PLAIN WORDS

By the end you should be able to compute a gradient through a small network by hand, explain each part of DQN, and say why each trick helps.

Applied Computational Mathematics Lab

PART 1

Deep value functions

Let the model learn its own features.



The objective, again

- Goal: find θ minimising the difference between v_π and v_θ, over the state distribution d induced by π and the dynamics p:

$$L(\theta) = \mathbb{E}_{S\sim d}\big[(v_\pi(S) - v_\theta(S))^2\big], \qquad \Delta\theta = -\tfrac{1}{2}\alpha\nabla_\theta L(\theta) = \alpha\,\mathbb{E}_{S\sim d}\big[(v_\pi(S) - v_\theta(S))\nabla_\theta v_\theta(S)\big]$$

- The expectation is hard to evaluate, so use stochastic gradient descent with a sampled return estimate G_t:

$$\Delta\theta = \alpha\big(G_t - v_\theta(S_t)\big)\nabla_\theta v_\theta(S_t), \qquad G_t = \begin{cases} R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots & \text{(Monte Carlo)} \\ R_{t+1} + \gamma v_\theta(S_{t+1}) & \text{(TD)} \end{cases}$$



IN PLAIN WORDS

Nothing new so far. What's new today is v_θ: a deep network instead of a linear map, and therefore a harder $\nabla_\theta v_\theta$.

TRY IT

Linear: $v_\theta(s) = \theta^\top\phi(s)$, so $\nabla_\theta v_\theta(s) = \phi(s)$, read straight off the features. A network has no such shortcut: we need automatic differentiation.

Why deep networks?

- In past lectures the feature representation was **fixed**, and v_θ was a linear map.
- Today: non-linear maps, typically deep neural networks.
- They discover feature representations tailored to the task, and we can reuse years of research on architectures and optimisation from supervised learning.



IN PLAIN WORDS

The question hanging over the course: is our feature vector any good? Good means states with similar values get similar features. But if we knew that, we wouldn't need to approximate. So let the model learn the features.

THREE WAYS TO WRITE DOWN A CHESS POSITION

- **The board**: 8 × 8 squares, a code for each. Wasteful: most squares are empty.
- **The pieces**: a list of pieces, each with its type and square.
- **The moves**: the starting position plus the moves so far, as in chess notation. Cheapest for short games.

Which is best? The one that predicts values best, and you don't know that in advance.



IN PLAIN WORDS

A deep network says: I don't care about your representation; I'll find a good one for you.

From linear to deep

- Draw linear regression as circles and arrows, and you get a perceptron.
- Stack linear layers, and you still get something linear: you must break the linearity with an **activation function**.
- A multilayer perceptron with one hidden layer:

$$v_\theta(S) = W_2 \tanh(W_1 S + b_1) + b_2, \qquad \theta = \{W_1, b_1, W_2, b_2\}$$

- When v_θ was linear, ∇v_θ was trivial. How do we compute it through a deep network?

DEEP, NOT JUST NEURAL

Deep learning means chaining a learning model into itself. Chain decision trees and you just get a bigger tree; chain linear layers with non-linear activations, and each layer can build new features from the last.

TRY IT

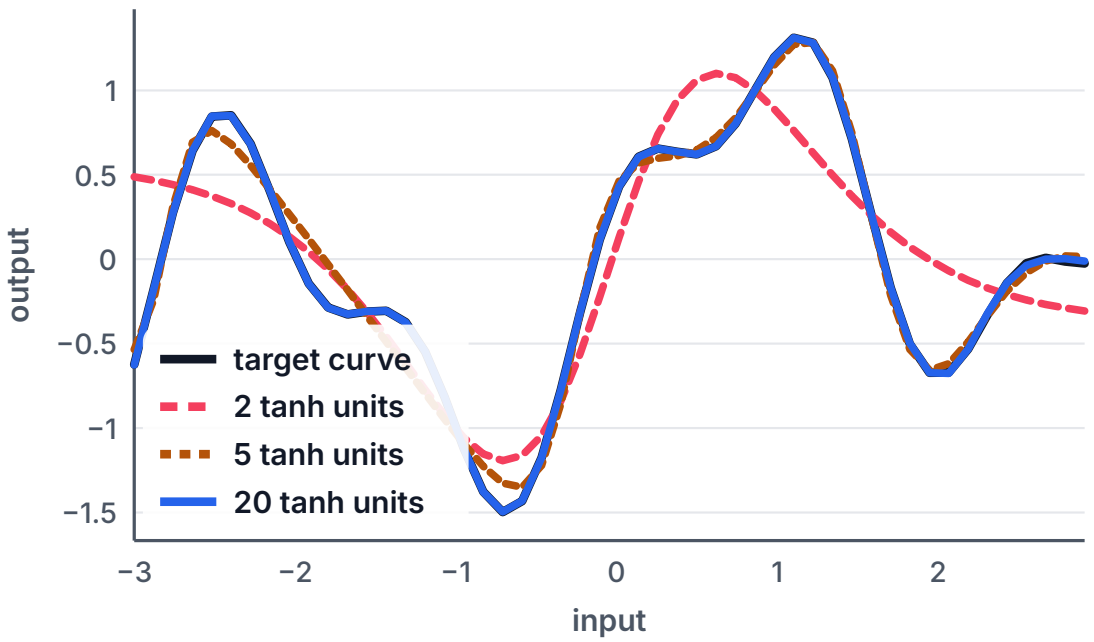
Two linear layers: $W_2(W_1 s) = (W_2 W_1)s$, a single linear map. With $W_1 = 2$, $W_2 = 3$: just $6s$. Insert a tanh in between, and the result can bend.



IN PLAIN WORDS

Everything is a vector: a matrix folds into a vector, an image is a row of pixels. Whatever your state looks like, you can feed it to a network.

A big enough network fits anything



Our fits, one hidden layer, 30,000 training steps each. Mean squared error: 2 units 0.125, 5 units 0.0108, 20 units 2.9e-05.

- A network with one hidden layer and enough units can approximate any continuous function on a bounded range (universal approximation).
- In practice, “enough” can mean very many.



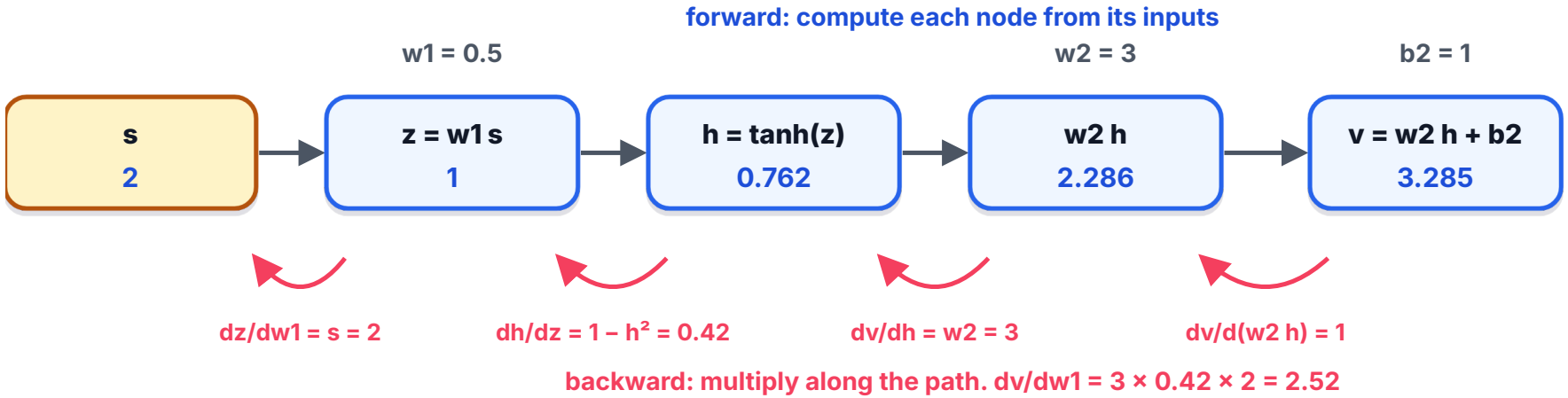
IN PLAIN WORDS

Deep learning has two advantages: expressiveness, and learning its own representation. In RL the second is gold.

THE PRICE

Networks are expensive: hard to train, hungry for data and memory.

Computational graphs and autodiff



- Represent a computation as a directed acyclic graph: inputs flow forward, one operation at a time.
- If we know the gradient of each node with respect to its inputs, one backward sweep gives the gradient of any node with respect to any other.
- Multiply the local gradients along each path; **add** them where paths merge.

TRY IT

Our tiny value network, $v = w_2 \tanh(w_1 s) + b_2$, at $s = 2$: $v = 3.285$. If the target is 4, the error is 0.715, and gradient descent nudges w_1 up in proportion to 2.52, w_2 in proportion to $h = 0.762$.

Automatic differentiation in practice

- Many frameworks compute these gradients for you. This course uses **JAX**:

NUMPY

arrays

The canonical Python library for matrices and matrix operations.

AUTODIFF

jax.grad

Gradients implemented by tracing your function.

ACCELERATORS

jax.jit

Just-in-time compilation for GPUs and TPUs.

- In code, you write the forward pass; the library writes the backward pass.



IN PLAIN WORDS

Writing gradients by hand for a deep network would be miserable. Draw the graph, and let the computer sweep backwards.

WHY IT MATTERS

The backward sweep costs about the same as the forward pass, whatever the number of parameters. That is why training networks with millions of weights is possible at all.

Deep Q-learning

- A network approximates $q_\theta : O_t \mapsto \mathbb{R}^m$ for m actions:
  - map the input to an h-dimensional vector;
  - apply a non-linear transformation, e.g. tanh or ReLU;
  - map the hidden vector to the m action values.
- We could also pass state **and** action as input.
- Update θ with

$$\Delta\theta = \alpha\big(G_t - q_\theta(S_t, A_t)\big)\nabla_\theta q_\theta(S_t, A_t), \qquad G_t = R_{t+1} + \gamma \max_a q_\theta(S_{t+1}, a)$$

- In deep-learning notation, the gradient of a **pseudo-loss**, with a stop-gradient $[\![\cdot]\!]$:

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



IN PLAIN WORDS

We ignore how the bootstrap target depends on θ. So this is not a true loss; it just has the right gradient.

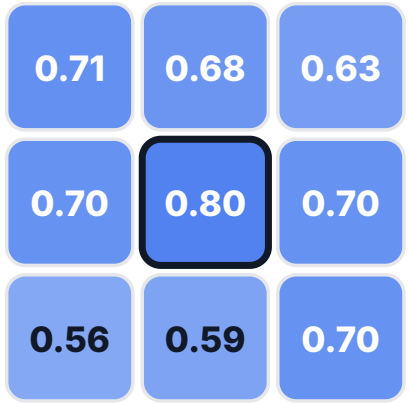
NOTE

Only the output of the action taken gets feedback on this step. The other outputs are left alone.

Example: tic-tac-toe with a table



Our run: tabular Q-learning, crosses against a random opponent, $\gamma = 0.9$, $\varepsilon = 0.1$, reward +1 / 0 / −1 at the end. After 40,000 games the table holds 2,081 positions.



Learned values of the first move: the centre comes out on top.

- Nine squares, nine actions. The policy is implicit: the argmax of Q.
- The table's values are the model's "weights". Swap the table for a network, and nothing else changes.
- Ninety-six percent wins sounds great, but the opponent is random.

What goes in, and what comes out

NETWORK	INPUT	OUTPUT
value	state (a vector)	one number, $v(s)$
action value, action in	state and action	one number, $q(s, a)$
action value, action out	state	one number per action
policy	state	an action, or a distribution over actions

- Why not feed in the whole history? It gets huge fast, and the Markov property says the present state is enough. (When it isn't, use an agent state, Lecture 1.)

A CLOUD OF SERVERS

An example from class: route jobs (small, medium, large) to small or large servers, or wait. The reward mixes completion time, utilisation and idle machines, each with a weight to tune. Is that reward correlated with what the client wants? Stop and check.



IN PLAIN WORDS

The right network depends on your problem, not on fashion.

Check yourself: Part 1

CHECK YOURSELF

1. Why doesn't stacking two linear layers give you anything new?

CHECK YOURSELF

2. In the tiny network, with $s = 1$ instead of 2 (same weights), what are z and $\partial z / \partial w_1$?

CHECK YOURSELF

3. What does the stop-gradient in the Q-learning pseudo-loss do?

CHECK YOURSELF

4. A Q-network with action-out has 4 actions. How many outputs, and how many get a gradient on one update?





HINT

Answers are at the end of the deck.

Applied Computational Mathematics Lab

PART 2

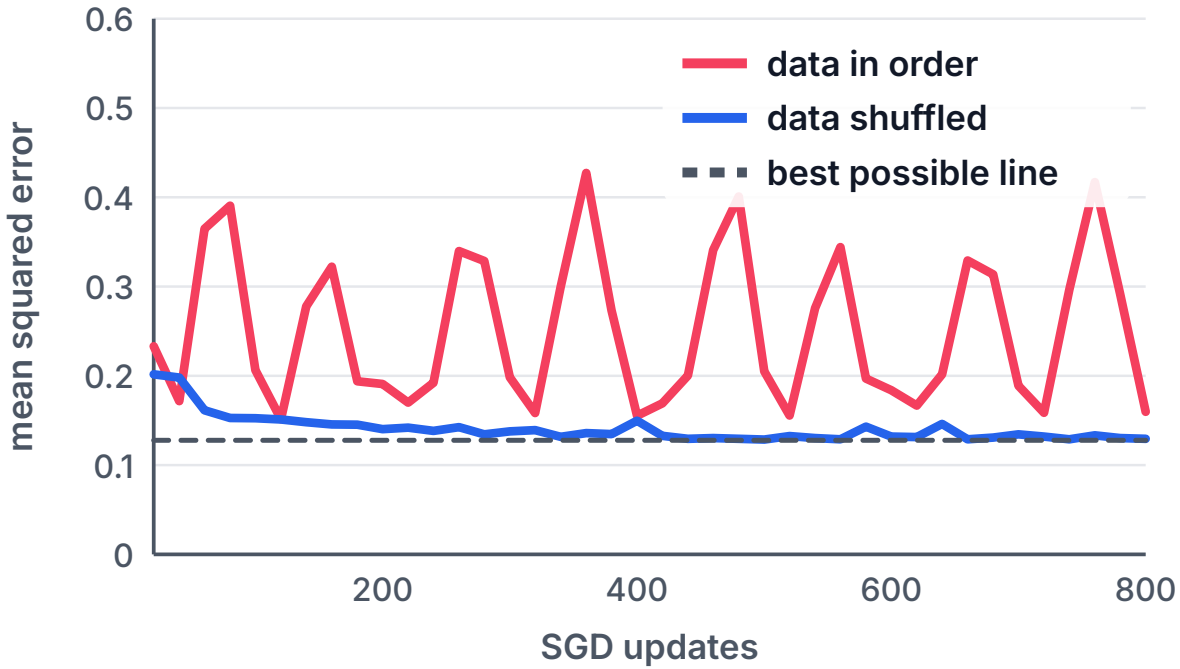
Deep-learning-aware RL

Make the RL problem friendlier to deep networks.



Online deep RL fights the data

- Supervised deep learning assumes gradients are sampled **i.i.d.**, and mini-batches usually work better than single samples.
- Online RL updates on every new step, and consecutive updates are **strongly correlated**.
- Like learning about a whole city from one street.



Our run (also in Lecture 7): a line fitted to a wavy curve by SGD. Shuffled, the last error is within 1% of the best fit; in order, the last pass swings between about 1.2 and 3.3 times it.

Friendlier data distributions

- **Experience replay** and **Dyna-style planning** mix online updates with updates on data sampled from a buffer of past experience, or from a learned model.
- Both reduce the correlation between consecutive updates, and support mini-batch updates instead of plain SGD.



IN PLAIN WORDS

Parallel environments: solved by engineering, not by maths. Totally legit. Eight streams also means eight times the data for every second you wait.

BETTER ALGORITHMS

eligibility traces

Spread each error over past states.

BETTER OPTIMISERS

momentum

Average gradients over time.

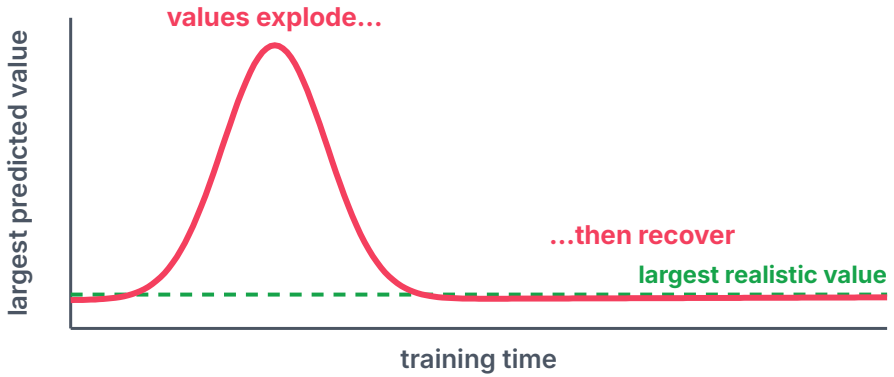
CHANGE THE SETTING

parallel environments

Eight cores, eight independent streams of experience.

The deadly triad, in practice

- With replay (as in DQN) we combine all three corners of the triad:
  - **function approximation**: a neural network fits the action values;
  - **bootstrapping**: the target uses $\max_a q_\theta(s, a)$;
  - **off-policy learning**: the replay buffer holds data from a mixture of past policies.
- Is this a sane thing to do?
- Van Hasselt et al. (2018) measured it: **unbounded divergence was rare**. More common: value estimates that explode early on, then recover. They called it **soft divergence**.



an illustration of "soft divergence", not data

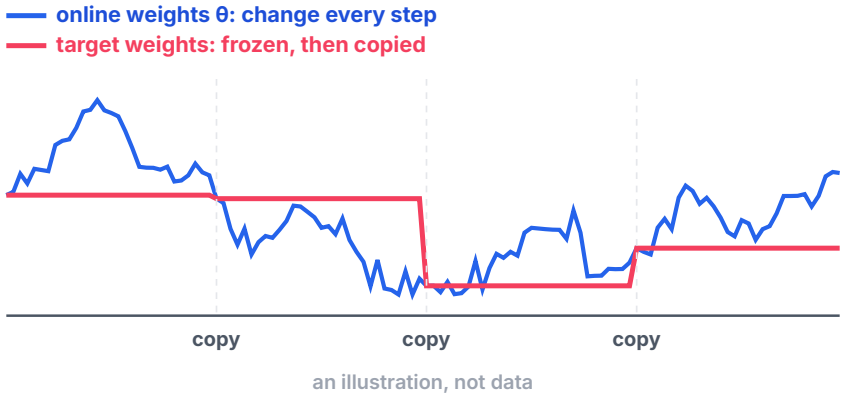


IN PLAIN WORDS

The guarantees die, but the patient usually survives. While the values are wrong, though, the policy built on them is wrong too.

Target networks

- Soft divergence can still leave the values poor for long stretches.
- Fix: a separate **target network** with parameters θ^{-}:
- 1. hold θ^{-} fixed when computing the bootstrap targets $\max_a q_{\theta^{-}}(s, a)$;
- 2. copy $\theta^{-} \leftarrow \theta$ only periodically (every few hundred or thousand updates).
- This breaks the feedback loop at the heart of the deadly triad: between copies, training is ordinary regression on fixed targets.



an illustration, not data



IN PLAIN WORDS

A dog chasing its own tail. Now tie the tail to a post for a while.

TRADE-OFF

Copy too often, and the target moves again.
Copy too rarely, and you learn from stale values.

Double DQN

- Q-learning overestimates (Lecture 6); double Q-learning fixes it by separating selection from evaluation.
- DQN already has two networks. Let the online network **pick** the next action and the target network **score** it:

$$L(\theta) = \frac{1}{2}\left(R_{i+1} + \gamma[\![q_{\theta^-}\left(S_{i+1}, \operatorname*{argmax}_a q_\theta(S_{i+1}, a)\right)]\!] - q_\theta(S_i, A_i)\right)^2$$

- A great combination with target networks: the frozen parameters serve as θ^-, almost for free (van Hasselt, Guez & Silver, 2016).



IN PLAIN WORDS

The online network is fresh, so it picks well. The frozen one is independent enough to score honestly.

TRY IT

Next state: online $q_\theta = (2.0, 2.6, 1.1)$, target $q_{\theta^-} = (2.4, 1.8, 1.5)$, $\gamma = 0.9$, reward 0.

DQN target: $0.9 \times \max q_{\theta^-} = 0.9 \times 2.4 = 2.16$.

Double DQN: q_θ picks action 2; q_{θ^-} scores it: $0.9 \times 1.8 = 1.62$. When the networks disagree on the best action, the double target is less optimistic.

Which targets explode?

- The form of the return estimator matters for divergence. Van Hasselt et al. (2018) trained many agents on Atari games and counted how often soft divergence happened:

BOOTSTRAP TARGET	RUNS WITH SOFT DIVERGENCE
Q-learning (no target network)	61%
Q-learning with a target network	14%
“Inverse” double Q	33%
Double Q (with the target network)	10%

Percentages as reported by van Hasselt et al. (2018), “Deep reinforcement learning and the deadly triad”. Not our runs.



IN PLAIN WORDS

Correlated data: shuffle it, or run in parallel. Moving targets: freeze them. Optimism: split the job in two. None of these is a proof; together they made deep Q-learning work in practice.

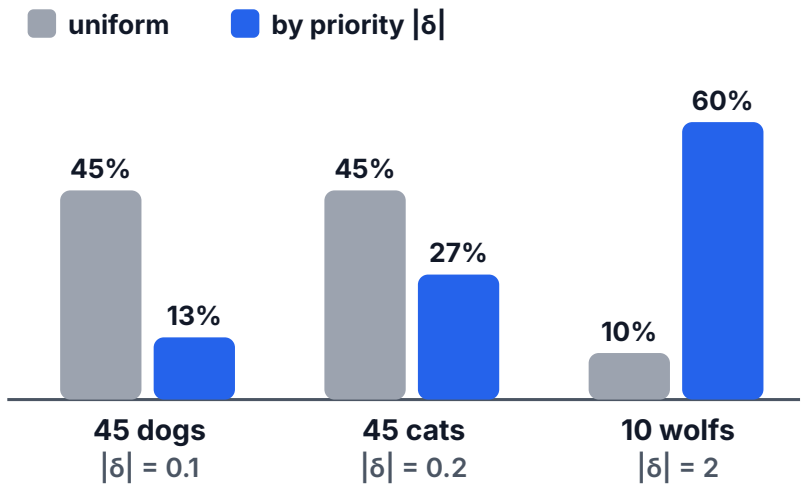
Prioritised replay

- DQN samples from replay uniformly. Idea: prioritise transitions we can learn much from (Schaul et al., 2016).
- Basic version: the priority of sample i is $|\delta_i|$, its TD error the last time it was sampled. Sample according to priority, and keep updating the priorities.
- In practice there are further design choices (how sharply to prioritise, how to correct).



IN PLAIN WORDS

A classifier for dogs, cats and wolves: tons of dog and cat photos, very few wolves, and the model is badly wrong on wolves. Spend your effort on the wolves.



Our computation: a buffer of 45 dogs, 45 cats and 10 wolves. Wolves go from 10% of the samples to 60%.

Prioritising shifts the data

- Prioritising biases the sampled states away from the states the agent actually visits: the updates become **even more off-policy**.
- Van Hasselt et al. (2018) saw more value explosions with stronger prioritisation.
- Importance sampling corrects it, at least partly: samples that are picked often count a little less in the update.

TRY IT

In the dogs, cats and wolves buffer, a wolf is sampled with probability about 0.60 instead of 0.10. Weighting its update by $0.10/0.60 \approx 0.17$ undoes the bias in expectation; in practice the correction is often only partial.



IN PLAIN WORDS

It doesn’t have to be wolves: any rare, surprising transition gets more attention.

Multi-step control

- The n-step Q-learning target, with the double-Q bootstrap:

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \underbrace{q_{\theta^-}\left(S_{t+n}, \operatorname*{argmax}_a q_\theta(S_{t+n}, a)\right)}_{\text{double-Q bootstrap}}, \qquad \Delta\theta = \alpha\left(G_t^{(n)} - q_\theta(S_t, A_t)\right)\nabla_\theta q_\theta(S_t, A_t)$$

- The return is partly on-policy; the bootstrap is off-policy. A well-defined target: “on-policy for n steps, then act greedily”. Less greedy, but still a policy improvement.
- It trades bias and variance, and relies less on bootstrapping, so it also makes divergence less likely.

TRY IT

$n = 3$, rewards 1, 0, 2, $\gamma = 0.9$, and the double-Q bootstrap value 5: $G^{(3)} = 1 + 0 + 0.81 \times 2 + 0.729 \times 5 = 6.265$.

n	RUNS WITH SOFT DIVERGENCE
1	82%
3	43%
10	9%

As reported by van Hasselt et al. (2018). Not our runs.



IN PLAIN WORDS

Less bootstrapping, fewer explosions.

Check yourself: Part 2

CHECK YOURSELF

5. Name two ways to make the data in online deep RL less correlated.

CHECK YOURSELF

6. Which three ingredients of the deadly triad does DQN combine?

CHECK YOURSELF

7. Why does a target network help?

CHECK YOURSELF

8. A buffer has 90 easy samples with $|\delta| = 0.1$ and 10 hard ones with $|\delta| = 1$. What share of samples are hard under proportional prioritisation?



Applied Computational Mathematics Lab

PART 3

RL-aware deep learning

Make deep networks fit reinforcement learning.



Architectures and inductive bias

- Many of deep learning's successes came from encoding the right inductive bias:
  - translation invariance in images: **convolutional** networks;
  - long-term memory: **gating** in LSTMs.
- We shouldn't just copy architectures designed for supervised problems. Which biases help an agent that acts?

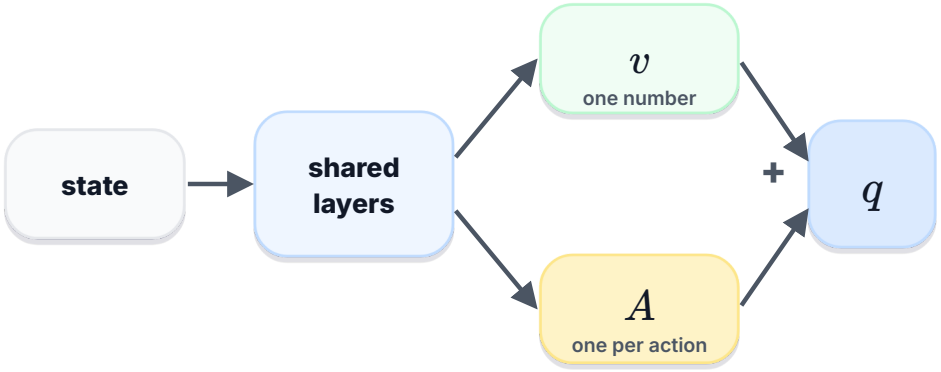
INPUT	A REASONABLE CHOICE
pixels (e.g. Atari)	convolutions
a feature vector	fully connected layers
sequences, partial observability	recurrent networks
sets, relations, long contexts	attention



IN PLAIN WORDS

A cat looks like a cat anywhere in the picture. A feature vector has no such structure, so convolutions don't help there.

Dueling networks



After Wang et al. (2016). Our drawing.

- Decompose $q_\theta(s,a) = v_\xi(s) + A_\chi(s,a)$, with $\theta = \xi \cup \chi$; A is the **advantage** of action a.
- The state value gets an update from every action, so it learns faster.
- In practice the advantages are centred (their mean subtracted), so that v and A are identifiable.

TRY IT

$v(s) = 5$, advantages $(+1, -0.5, -0.5)$: $q = (6, 4.5, 4.5)$. An update that raises v to 5.2 raises all three q values at once; the advantages alone decide which action is best.



IN PLAIN WORDS

On a straight, empty road, every action is about the same: the state value carries it.

Network capacity

- In supervised deep learning: more data + more capacity = better performance. The loss is easier to optimise, with less interference.
- In RL, larger networks typically do better overall too.
- But they are also **more susceptible to the deadly triad** (van Hasselt et al., 2018).



IN PLAIN WORDS

Scale up, but watch your values for explosions. More capacity isn't free: it's one more thing to tune.

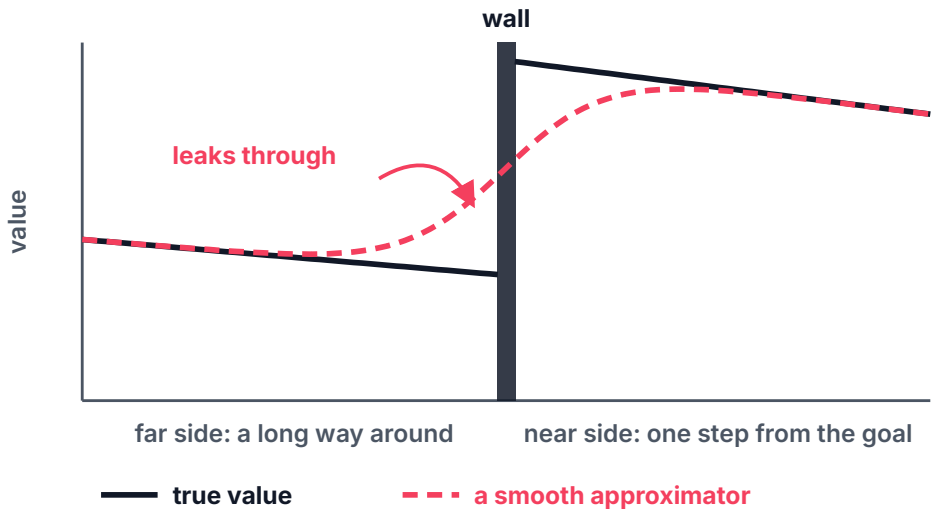
IN PRACTICE

A useful habit: log the largest predicted value during training. If it climbs far above any return the task can give, you are seeing soft divergence.

TRY IT

Rewards in [0, 1] and $\gamma = 0.99$: no return can exceed $1/(1 - 0.99) = 100$. A predicted value of 10,000 is a red flag.

Generalisation can leak



An illustration, not data.

- Picture a maze with a wall. On one side you're one step from the goal; on the other, it's a long way around.
- Nearby states, very different values: a sharp discontinuity in v_π.
- A network smooths over the wall, and the values **leak** through it.
- With TD the leaked values become targets for their neighbours, so the error spreads: **leakage propagation**. Monte Carlo leaks too, but doesn't feed on its own leaks.



IN PLAIN WORDS

Not a bug in one algorithm: it's what generalisation does near a sharp edge.

Learning about many things

- Behind the deadly triad and leakage is the same problem: **inappropriate generalisation**.
- Better representations help. One way to get them: share the state representation across many prediction tasks:
- 1. predict the values of different policies;
- 2. predict future observations;
- 3. predict or control other “cumulants”, signals other than the main reward.



IN PLAIN WORDS

Every extra prediction shapes a better shared representation, almost for free. Even if you only care about one reward, learning about many things helps.

EXAMPLE

A driving agent that also predicts the next camera frame and the distance to the car ahead learns features about roads and cars, which its main value function can reuse.

- More in Lecture 11.

Check yourself: Part 3

CHECK YOURSELF

9. In a dueling network, $v(s) = 2$ and the advantages are $(0.5, -0.5)$. What are the q values? Which action is greedy?

CHECK YOURSELF

10. Why do convolutions help on Atari pixels but not on a feature vector?

CHECK YOURSELF

11. Why is leakage worse for TD than for Monte Carlo?



WRAP-UP

Summary

- Deep networks learn their own features; **autodiff** gives the gradients in one backward sweep.
- **Deep Q-learning** minimises a pseudo-loss with a stop-gradient on the target.
- Online RL data is correlated: **replay**, planning, parallel streams.
- DQN combines the whole deadly triad; in practice it shows **soft divergence**, tamed by **target networks**, **double** targets and **multi-step** returns.
- **Prioritised replay** spends updates where errors are large, at the cost of more off-policy data.
- **Dueling** networks, capacity and generalisation: design networks for RL, and watch for leakage.

$$L(\theta) = \tfrac{1}{2}\big([\![G]\!] - q_\theta(S, A)\big)^2$$

$$q_\theta(s, a) = v_\xi(s) + A_\chi(s, a)$$

$$p(i) \propto |\delta_i|$$

ANSWERS

Answers to check yourself

- **1.** A product of linear maps is linear: $W_2 W_1 s$.
- **2.** $z = 0.5$; $\partial z / \partial w_1 = s = 1$.
- **3.** It treats the target as a constant, so the gradient only flows through $q_\theta(S_t, A_t)$.
- **4.** 4 outputs; only the action taken gets a gradient.
- **5.** Two of: replay, Dyna-style planning, parallel environments (and better optimisers or traces help).
- **6.** Function approximation, bootstrapping, off-policy data.
- **7.** It freezes the targets between copies, breaking the feedback loop.
- **8.** $10 \times 1 / (90 \times 0.1 + 10 \times 1) = 10/19 \approx 53\%$.
- **9.** $q = (2.5, 1.5)$; action 1.
- **10.** Pixels have spatial structure (the same pattern anywhere); a feature vector's positions mean different things.
- **11.** TD bootstraps: leaked values become targets for neighbours, so the error spreads; MC targets real returns.

CREDITS

Sources and credits

- The order of topics follows *Deep Reinforcement Learning* (parts on deep value functions, deep-learning-aware RL and RL-aware deep learning) of the DeepMind × UCL Reinforcement Learning Lecture Series (2021), by Matteo Hessel. 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.
- Mnih et al. (2015), DQN; van Hasselt, Guez & Silver (2016), Double DQN; van Hasselt et al. (2018), “Deep reinforcement learning and the deadly triad” (the soft-divergence percentages quoted here); Schaul et al. (2016), prioritised replay; Wang et al. (2016), dueling networks.
- Companion videos: **ACML Explains**, season 1, episodes 27–30. More from the lab: acml.teddylazebnik.com



Applied Computational Mathematics Lab

END OF LECTURE 8

Next: planning with models

Lecture 9 lets the agent learn a model of the world and plan with it: Dyna, Monte Carlo tree search and AlphaZero.

