

Value-Based Reinforcement Learning

Learning to act when the model is gone

Jinkyoo Park · KAIST

From the Bellman equation you can *solve* to the one you can only *sample*

Where we are — the OR lineage, made data-driven

PREVIOUS STEP

THIS CHAPTER'S QUESTION

Lecture 7 could compute backups from a known transition model.

Now estimate values using observed transitions.

We keep states, actions, rewards and policies. Sampled targets replace model expectations; prediction error and exploration now affect learning.

Update one action value from a measured temperature transition.

Learning route — change the target, then change the representation

Bring: The Bellman backup and policy/value distinction from Lecture 7.

FIRST PASS	WHAT TO DO
Follow the idea	MC/TD evaluation → control → behavior versus target → DQN/Double DQN
Work without the solution	Update one action value from a measured temperature transition.
Return later	Proof details and the wider deep-RL catalogue are references.

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

— ACT 1

Evaluation without a model

Q1. Fix a policy. With no P and no R , how good is it?

The arena — from MDP to experience

Q1 Evaluate?

Q2 Improve?

Q3 Explore?

Q4 Scale?

We no longer query a model. We *act*, and a stream comes back:

$s_1, a_1, r_2, s_2, a_2, r_3, s_3, \dots, s_T$ (one episode)

- the agent chooses $a_t = \pi(s_t)$ — *decision*,
- the environment returns r_{t+1}, s_{t+1} — *the only window onto P, R* ,
- from this stream we must recover Q^* and act on it — *learning*.

Start with the humblest sub-task: **prediction**. Fix a policy π ; estimate its value V^π . No control yet — just: how good is this behavior? Everything harder is built on getting this right.

The quantity, and the two ways to estimate it

The definition has not changed since Lecture 7 — only our access to it:

$$V^\pi(s) = \mathbb{E}_\pi[G_t \mid s_t = s] = \underbrace{\mathbb{E}_\pi\left[\sum_{k \geq 0} \gamma^k r_{t+k+1} \mid s_t = s\right]}_{\text{the full return}} = \underbrace{\mathbb{E}_\pi[r_{t+1} + \gamma V^\pi(s_{t+1}) \mid s_t = s]}_{\text{one step, then bootstrap}}$$

The same value, written two ways. Each writing is a recipe for estimating it from samples:

- **Monte Carlo** reads the *left* form: wait for the episode to end, average the realized return G_t .
- **Temporal Difference** reads the *right* form: take one step, then **trust your own current estimate** of the rest.

Sample the whole thing, or sample one step and bootstrap — **the deepest fork in all of value-based RL.**

Monte Carlo prediction — average what actually happened

Visit s_t , watch the episode finish, collect the realized return $G_t = r_{t+1} + \gamma r_{t+2} + \dots + \gamma^{T-t-1} r_T$. Then nudge:

$$V(s_t) \leftarrow V(s_t) + \alpha \left[\underbrace{G_t}_{\text{target}} - V(s_t) \right]$$

- **Unbiased.** $\mathbb{E}[G_t] = V^\pi(s_t)$ exactly — the target is the real thing, not a guess.
- **No model, no bootstrap.** Each state's estimate stands alone; we never lean on neighbors.
- **The price:** must wait for the terminal state (no online updates), and the return of a long episode is a **high-variance** single draw — one unlucky tail corrupts the estimate.

MC also *wastes* the structure of the MDP: it learns each state separately and never reuses the fact that s leads to s' . That wasted structure is exactly what TD reclaims.

One episode can visit a state twice — define the Monte Carlo sample

Make the averaging convention explicit

Consider an illustrative episode $A \rightarrow B \rightarrow A \rightarrow \text{terminal}$ with rewards 1, 2, 3 and $\gamma = 1$. The two returns observed after visiting A are **6** and **3**.

ESTIMATOR OF $V^\pi(A)$	CONTRIBUTIONS FROM THIS EPISODE
First-visit Monte Carlo	use 6, from the first visit to A
Every-visit Monte Carlo	use 6 and 3; their average for this episode is 4.5

Across episodes, keep the count of included visits and update the running average. Neither estimate from this single episode is guaranteed to equal the true value.

A Monte Carlo target is a **sampled return after a specified visit**. State values are averages over repeated experience, not a rule that every visit has the same realised return.

Temporal Difference — learn a guess from a guess

Don't wait. After a *single* transition (s_t, r_{t+1}, s_{t+1}) , update:

$$V(s_t) \leftarrow V(s_t) + \alpha \left[\underbrace{r_{t+1} + \gamma V(s_{t+1})}_{\text{TD target}} - V(s_t) \right]$$

The target contains $V(s_{t+1})$ — **our own current estimate**. It need not be better on every individual transition. This is **bootstrapping**.

- **Online, incremental, model-free.** One step is enough — ideal for long or non-terminating tasks.
- **Often lower variance, possible bias.** The target depends on one reward plus a learned value, not a whole noisy trajectory — but it inherits whatever error $V(s_{t+1})$ currently carries.
- TD can learn faster in many tasks; neither method dominates for every reward process, initialization, and step size.

One transition, two targets — calculate the update

Let $V(s) = 2$, $V(s') = 3$, $\alpha = 0.2$, and $\gamma = 0.9$. We observe reward 1; the next transition pays 4 and ends the episode.

METHOD	TARGET	UPDATED VALUE OF THE FIRST STATE
TD, available after step 1	$1 + 0.9(3) = 3.7$	$2 + 0.2(3.7 - 2) = 2.34$
MC, available after termination	$1 + 0.9(4) = 4.6$	$2 + 0.2(4.6 - 2) = 2.52$

Both use **old estimate + step size × prediction error**. TD uses an estimate of the future; MC waits for its realized return.

If the first transition had terminated, both targets would be **1**: terminal continuation value is zero.

Both estimators, the same episodes



Five states, a random walk from **C**, reward 1 only on exiting right — so $V^\pi(s) = i/6$. MC and TD see the *identical* stream and differ only in the target. Watch TD's low-variance target settle while MC's realized returns keep jittering; then raise α and watch MC degrade faster.

The triangle — MC, DP, and TD

Three ways to back up a value; TD is the one that takes from both parents.

	SAMPLES (NO MODEL)	BOOTSTRAPS (USES OWN ESTIMATE)
Dynamic Programming (<i>Lec 7</i>)	no — needs full P	yes
Monte Carlo	yes	no — waits for the true return
Temporal Difference	yes	yes

TD = the model-free-ness of MC + **the bootstrapping of DP**.

That single inheritance is why TD, not MC, becomes the engine of control. We now turn the prediction crank into a *decision-making* one.

Check — MC against TD

Monte Carlo waits for the episode to end and averages the actual return; TD updates after one step using its own estimate of what follows. What is the trade?

- ☒ A MC uses an unbiased full-return target; TD can introduce bootstrap bias, often reduces variance, and can update immediately
- ☐ B MC has lower variance and higher bias; TD the reverse
- ☐ C Both are unbiased; TD is simply faster to compute
- ☐ D TD is unbiased only in deterministic environments, where it equals MC

MC's target is a real sampled return — correct in expectation, but carrying the noise of every step that followed. TD's target $r + \gamma V(s')$ leans on a current estimate, importing whatever error that estimate has while discarding all the downstream noise. Bootstrapping is the trade at the heart of the lecture, and in Act 4 it turns out to be one of the three ingredients that can make learning diverge.

— ACT 2

Improvement without a model

Q2. We can score a policy. How do we make it better, still without P ?

Why V is not enough — the hidden P



Lecture 7's improvement step, written out:

$$\pi'(s) = \arg \max_a \sum_{s'} P(s' \mid s, a) [R(s, a, s') + \gamma V(s')]$$

To turn a *value* into an *action*, you must look one step ahead — and looking ahead needs the model.

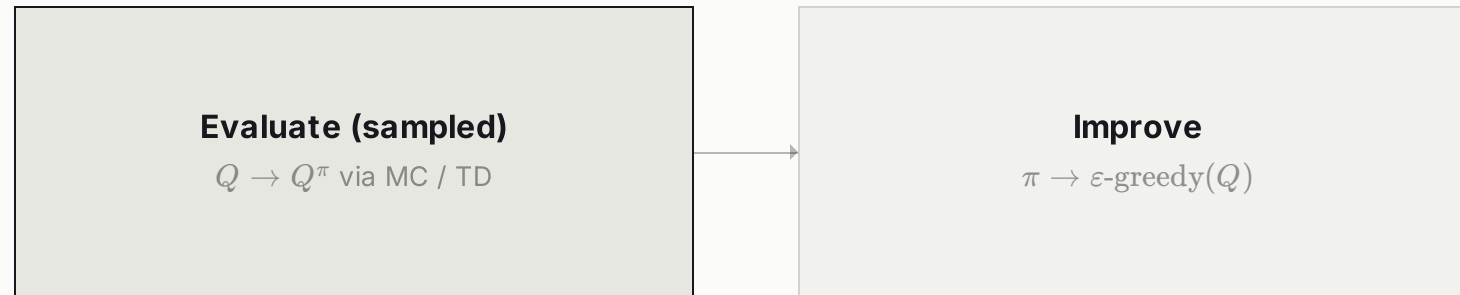
The fix is almost embarrassingly simple. Store the action-value Q instead of V :

$$\pi'(s) = \arg \max_a Q(s, a)$$

No sum, no P . **Q already has the look-ahead baked in.** This is *the* reason value-based RL learns Q , not V : Q is the model-free currency of decisions.

Generalized Policy Iteration, now from samples

The Lecture 7 dance is unchanged — evaluate, improve, repeat — but each half is now sampled.



One subtlety the model-based version never faced: to estimate $Q^\pi(s, a)$ from data, *every* (s, a) must actually be tried. A purely greedy policy never tries the actions it currently dislikes — and so never learns they were good.

Improvement now requires exploration. **That is the new tax.**

The tax, paid — ε -greedy, and the bandit underneath

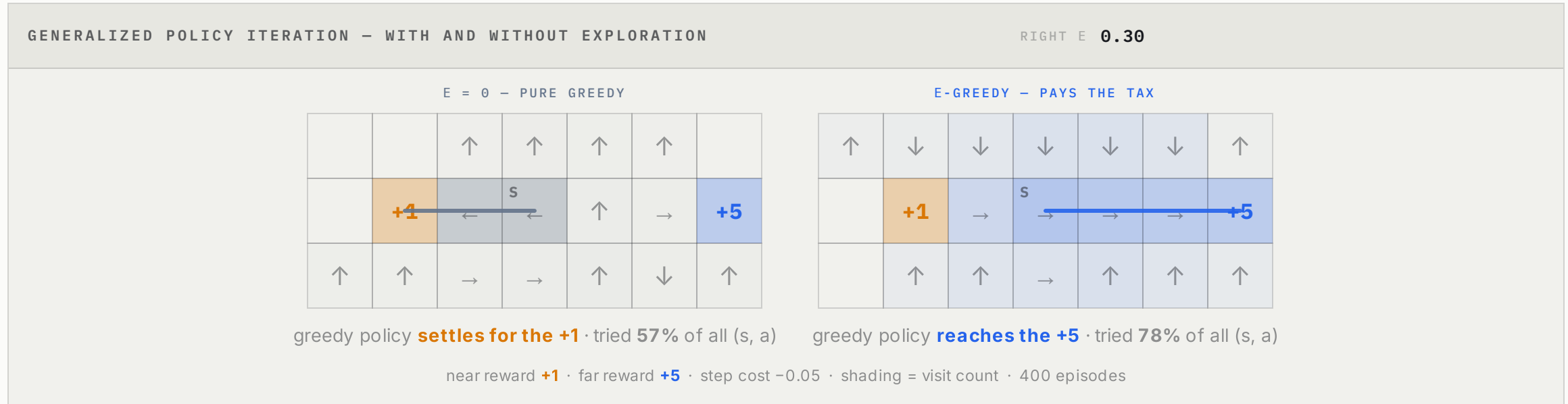
The simplest way to keep trying everything:

$$a_t = \begin{cases} \arg \max_a Q(s_t, a) & \text{with prob. } 1 - \varepsilon \quad (\text{exploit}) \\ \text{a random action} & \text{with prob. } \varepsilon \quad (\text{explore}) \end{cases}$$

This is the **exploration–exploitation trade-off** of the n -armed bandit, now living inside every state. The bandit was RL with the dynamics deleted; here the dynamics are back, but the same dilemma — *spend on what you know, or pay to learn more* — governs every step.

- Decay $\varepsilon \rightarrow 0$ (e.g. $\varepsilon = 1/t$) and, with every (s, a) visited infinitely often, the greedy policy in the limit is optimal.
- Hold ε fixed and you keep a permanently-cautious, permanently-curious agent — sometimes exactly what a real system needs.

The tax, seen — what a greedy agent never learns



Two Q-learning agents, identical but for ϵ . The greedy one locks onto the near +1 it stumbled into first and stops looking — read its coverage figure: the actions it dislikes are never tried, so their values are never corrected. The exploring one pays a little return per episode and finds the far +5.

Monte Carlo control — evaluate an episode, then improve the policy

Algorithm between prediction and TD control

1. Generate a complete episode using the current exploratory policy.
2. Work backwards to compute $G_t = r_{t+1} + \gamma G_{t+1}$.
3. Average the selected returns for each visited (s_t, a_t) into $Q(s_t, a_t)$.
4. Make the policy ε -greedy with respect to the updated Q values.

For four actions and $\varepsilon = 0.2$, a unique greedy action gets probability **0.85** and each other action gets **0.05**.

Exploring starts assumes we can start episodes in every relevant state-action pair. When that is impractical, use an exploratory policy to collect coverage. A fixed positive ε keeps the executed policy stochastic.

Check — the price of exploring

Assume tabular Q-learning satisfies its convergence conditions. With $\varepsilon = 0.1$ fixed, what happens to its behaviour?

- ☐ A It converges to the optimal policy, because Q converges to Q^*
- ☒ B Q converges to Q^* , but the agent keeps taking a random action one time in ten
- ☐ C Neither converges — a fixed ε prevents Q from converging
- ☐ D It converges to the optimal policy only if the environment is deterministic

Under adequate state–action coverage and diminishing step sizes, tabular Q-learning can converge to Q^* while its behavior remains exploratory. With two actions and one unique best action, $\varepsilon = 0.1$ chooses the best with probability $0.9 + 0.1/2 = 0.95$, not 0.9. Exploration can choose the best action too.

— ACT 3

Whose value are we learning?

Q3. The data comes from an exploring policy. Which policy's value does the update actually estimate?

Two control rules, one tiny difference



Put TD evaluation of Q together with ε -greedy improvement. Two natural updates appear, differing in *one term*:

SARSA — ON-POLICY

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$$

a' is the action **actually taken** next.

Q-LEARNING — OFF-POLICY

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

max over **all** next actions, taken or not.

S, A, R, S', A' — SARSA literally uses the next action it sampled. Q-learning replaces that sampled a' with the **best possible** a' . For these two algorithms, that substitution reveals the distinction: the target policy need not be the policy that generated the data.

Same data, different policy in the target

Let $Q(s, a) = 2$, reward $r = 1$, $\gamma = 0.9$, and $\alpha = 0.2$. At the next state, the two action values are **5 and 1**. Exploration actually selects the action worth 1.

UPDATE	CONTINUATION USED	TARGET	NEW ESTIMATE
SARSA	actual next action: 1	$1 + 0.9(1) = 1.9$	$2 + 0.2(1.9 - 2) = 1.98$
Q-learning	greedy next action: 5	$1 + 0.9(5) = 5.5$	$2 + 0.2(5.5 - 2) = 2.70$

The reward and transition are identical. The difference is **what behavior we assume after arriving**. This is what the cliff experiment will make visible.

The 2×2 that organizes everything

Cross the two forks of the whole lecture — *bootstrap?* and *whose policy?*

	NON-BOOTSTRAP (MC)	BOOTSTRAP (TD)
On-policy	MC control	SARSA
Off-policy	off-policy MC	Q-learning

On-policy: the policy that *generates* the data is the policy we *evaluate*. SARSA learns the value of the very (exploring, ϵ -greedy) behavior it is running.

Off-policy: the behavior policy and the target policy differ. Q-learning *behaves* ϵ -greedily but *learns* the value of the pure-greedy optimal policy — it studies one policy while living as another.

Why Q-learning is "off-policy," precisely

Look at what each update assumes about the next step.

- SARSA's target uses $Q(s', a')$ where $a' \sim \pi_{\text{behavior}}$ — the **real** next action, exploration and all. It is honest about the policy it runs.
- Q-learning's target uses $\max_{a'} Q(s', a')$ — it *pretends* the next action will be greedy, regardless of the a' actually taken. The behavioral a' is collected, then **ignored**.

THE CLEAN EXTREME

act with $\varepsilon = 1$, i.e. fully random

SARSA learns the value of *acting randomly*. Q-learning learns the value of the *optimal* policy — **while acting randomly**.

This convergence statement is tabular: finite stationary MDP, bounded rewards, $\gamma < 1$, every state–action pair visited infinitely often, and Robbins–Monro step sizes **per pair**. Random actions alone do not guarantee visits to unreachable states.

Temperature thread — learn from one measured transition

shared teaching example · predict before revealing the calculation

The model probabilities from Lecture 7 are now unavailable to the learner. Observe a heating transition from $x = -2$ to $x' = -1$, with reward -2 . Let $Q(-2, \text{heat}) = -4$, $\max_a Q(-1, a) = -1$, $\gamma = 0.9$ and step size 0.5 .

Predict: Should this observation raise or lower the old estimate -4 ?

Calculate and check. The target is $-2 + 0.9(-1) = -2.9$. The TD error is $-2.9 - (-4) = 1.1$, and the updated estimate is $-4 + 0.5(1.1) = -3.45$.

The update uses one measured outcome. Lecture 7 instead averaged outcomes using a known transition model.

Try it — the observed heating step fails

work independently · reveal only after writing an answer

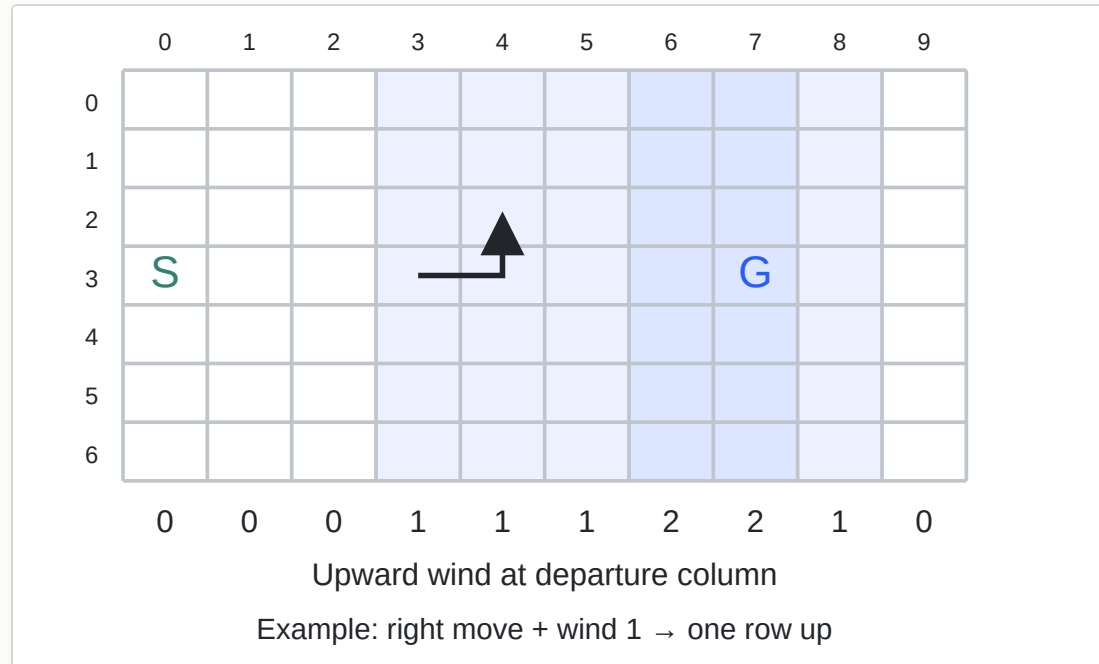
Start again from $Q = -4$. This time the next state remains -2 , the measured reward is -5 and its next maximum Q is -3 . Keep $\gamma = 0.9$ and step size 0.5 . Calculate the update.

Check your answer. Target $= -5 + 0.9(-3) = -7.7$; TD error $= -3.7$; new $Q = -4 + 0.5(-3.7) = -5.85$.

One good or bad observation does not identify the transition probability. Repeated data and an appropriate update schedule matter.

The original windy gridworld — learn the transition through experience

Four-action version, redrawn



Start S, goal G. The numbers below the columns give upward wind strength; grid boundaries clip the resulting position.

WHAT SARSA LEARNS

An intended move and the resulting displacement differ. For this convention, the wind at the **departure column** acts with the chosen move.

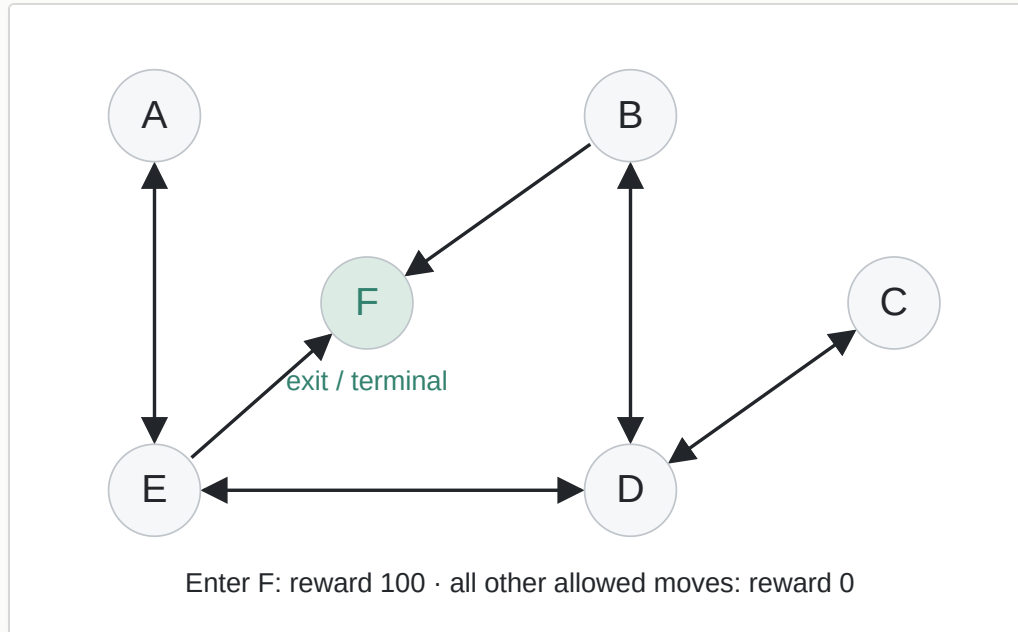
From row 3, column 3, moving right with wind 1 ends at row 2, column 4. Rows increase downward; indices start at 0.

Use reward -1 until arrival at G. SARSA learns from the **actual next state and the next exploratory action**, without being handed the wind model.

The source's preceding page 57 depicts the random-walk prediction example; page 58 is the windy control task. Prediction of a fixed policy and learning a control policy are different experiments.

The original six-room example — follow the actual numerical updates

Initial Q values 0, step size 0.5, discount 0.8



Original door connectivity. Here F ends the episode, as stated on source pp. 70 and 72. Entering F pays 100; other moves pay 0.

THREE SOURCE UPDATES

OBSERVED TRANSITION	UPDATE	NEW Q
B → F	$0 + 0.5(100 - 0)$	50
D → B	$0 + 0.5(0 + 0.8(50) - 0)$	20
B → F again	$50 + 0.5(100 - 50)$	75

Only the visited entry changes. The reward at F starts affecting actions taken **before** reaching F through bootstrapping.

The target on a terminal transition is the immediate reward. The next slide separates this episodic rule from the source's different continuing-state table.

Terminal versus continuing — why the source also shows a value of 500

source pp. 68 and 73 use a rewarded F self-loop; pp. 70–72 terminate at F

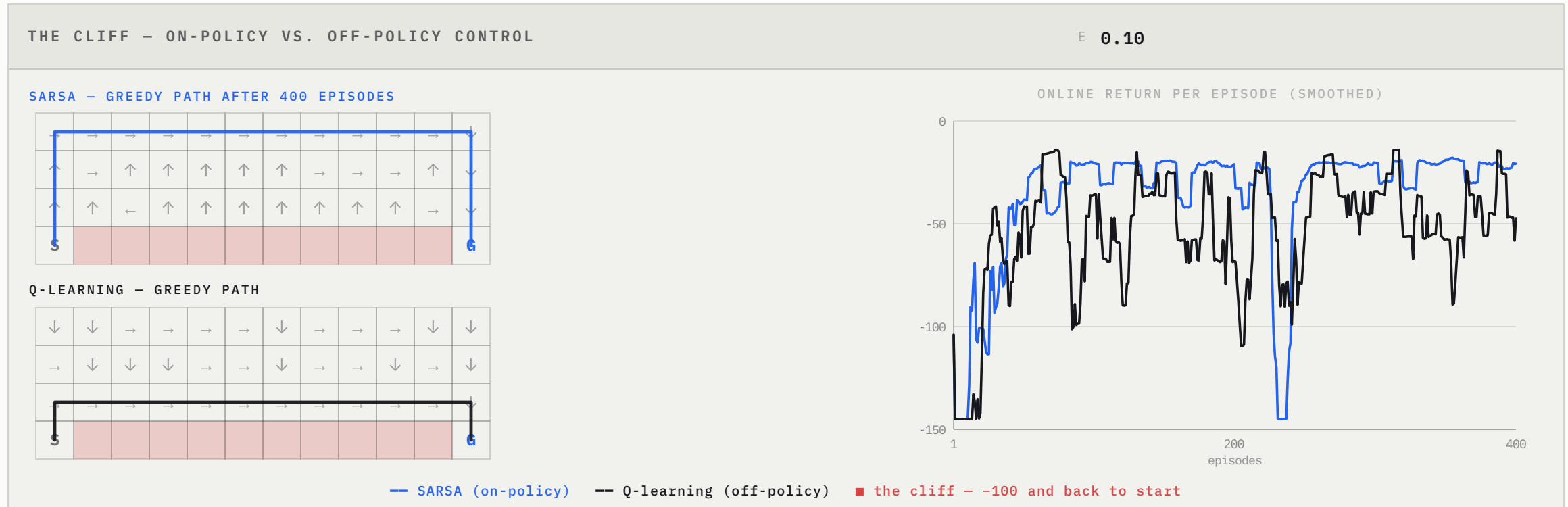
Those conventions define different MDPs. They must not share the same claimed converged Q table.

CONVENTION	VALUE AFTER REACHING F	LIMITING $Q(B, F)$
Terminal F: episode ends	$V(F) = 0$	$100 + 0.8(0) = \mathbf{100}$
Continuing F: staying pays 100 each step	$V(F) = 100 + 0.8V(F) = 500$	$100 + 0.8(500) = \mathbf{500}$

With terminal F, the best $D \rightarrow B \rightarrow F$ continuation has $Q^*(D, B) = 80$, and $C \rightarrow D \rightarrow B \rightarrow F$ has $Q^*(C, D) = 64$. These are limits, unlike the intermediate values 50, 20 and 75 on the previous slide.

A terminal mask is part of the **problem definition**. Changing it changes the correct answer, even when the room diagram and discount remain the same.

The cliff — where the difference becomes visible



Q-learning learns the *optimal* path — along the edge — but it *acts* ϵ -greedily, and the occasional random step plunges it off the cliff: higher reward in theory, worse online. **SARSA** accounts for its own exploration and learns a **cautious** detour. Neither is "better" — the cliff teaches the design choice.

Check — what makes Q-learning off-policy

SARSA updates toward $r + \gamma Q(s', a')$ using the action actually taken next; Q-learning uses $r + \gamma \max_{a'} Q(s', a')$. Why does that one difference make Q-learning off-policy?

- ☐ A Because the `max` makes the update deterministic
- ☐ B Because Q-learning needs a replay buffer and SARSA does not
- ☒ C Because the target refers to the greedy action, not the one the behaviour policy chose — so it estimates the optimal policy's value from another policy's data
- ☐ D Because SARSA converges to Q^* and Q-learning does not

SARSA evaluates *the policy that is driving*, exploration and all — on the cliff it learns the safe path, because it accounts for the chance of stepping off while exploring. Q-learning evaluates the greedy policy regardless of who is driving, so it learns the optimal edge-hugging path and then keeps falling off it. Neither is wrong; they are answers to different questions.

— ACT 4

Scaling past the table

Q4. Everything so far stores one number per (s, a) . What happens when the state is an image?

The wall — a table cannot hold the world

Q1 Evaluate?

Q2 Improve?

Q3 Explore?

Q4 Scale?

Everything so far stores one number per (s, a) . Then comes Atari:

$$\text{state} = \text{four } 84 \times 84 \text{ grayscale frames} \Rightarrow |\mathcal{S}| \approx 256^{84 \times 84 \times 4}$$

more configurations than atoms in the universe. A Q -table is impossible three times over: too many states to *visit*, too many to *store*, and — fatally — no **generalization**: a table says nothing about a state it has never seen.

The machine-learning answer: stop storing, start *approximating*.

$$Q(s, a) \approx \hat{Q}(s, a; w), \quad \text{e.g. } \hat{Q}(s, a; w) = w^\top \phi(s, a) \text{ or a neural net}$$

Now "learning" means fitting w — and similar states share answers.

Why DQN stacks frames — position alone may hide velocity

Representation is part of the state definition

Two situations can have the same current image but require different actions:

PREVIOUS BALL POSITION	CURRENT POSITION	INFERRED HORIZONTAL MOTION
10	20	moving right
30	20	moving left

A single frame at position 20 cannot distinguish them. DQN's input stacks **four recent 84×84 grayscale frames**, and a convolutional network extracts spatial features from that stack.

The network approximates Q **after a state representation has been chosen**. Frame stacking supplies short-term motion information; it does not guarantee a Markov state in every environment.

Q-learning as regression — the semi-gradient step

Treat the Bellman target as a label and minimize squared error on each transition:

$$\mathcal{L}(w) = \frac{1}{2} \left(\underbrace{r + \gamma \max_{a'} \hat{Q}(s', a'; w)}_{\text{target}} - \hat{Q}(s, a; w) \right)^2 \Rightarrow w \leftarrow w + \alpha \delta \nabla_w \hat{Q}(s, a; w)$$

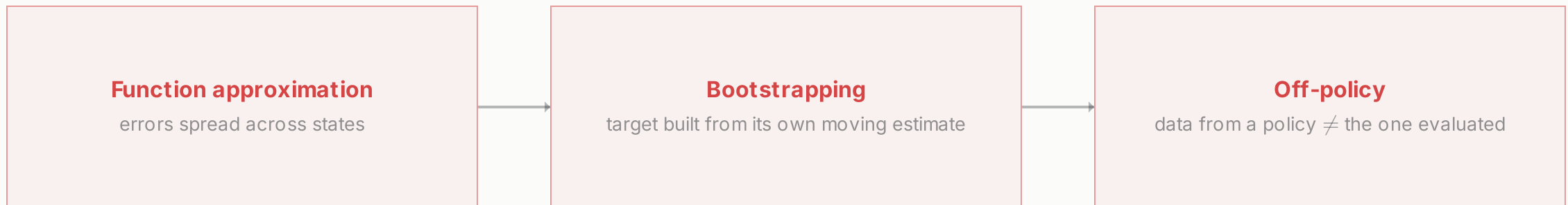
with TD error $\delta = r + \gamma \max_{a'} \hat{Q}(s', a'; w) - \hat{Q}(s, a; w)$.

It is called a **semi-gradient** method for an honest reason: we differentiate the prediction $\hat{Q}(s, a; w)$ but *treat the target as fixed*, even though it too depends on w . We are chasing a target that moves when we move.

In the tabular world that chase still converged. With function approximation, it can **diverge**. Tabular Q-learning's guarantees do not survive the jump — and we must understand why before trusting a network.

The deadly triad — why naive deep Q-learning blows up

These three ingredients together create an important source of instability. Their absence is not a universal safety guarantee for arbitrary algorithms or step sizes.



Q-learning with a neural net has *all three*. The data are also strongly **correlated** (consecutive frames) and the target **shifts** on every update. Plain SGD on $\mathcal{L}(w)$ fights itself.

The breakthrough was not a new objective — it was **two engineering tricks** that tame the triad.

The triad, switched on and off

TWO STATES, ZERO REWARDS, TRUE VALUE = 0 — WHAT COULD GO WRONG

DIVERGING
 $|V(s_1)| = 5.2e+33$



Two states, every reward 0, so the true value is 0 everywhere. With all three switches on, $|V(s_1)|$ grows without bound. Turn **any one** of them off — use a table, use the return instead of a bootstrap, or update on-policy — and the same algorithm collapses to 0. (*Tsitsiklis & Van Roy, 1997*)

DQN — the two tricks

Mnih et al., 2013; Nature 2015

Keep Q-learning's update. Add two stabilizers.

- **Experience replay.** Store transitions (s, a, r, s') in a buffer D ; train on *random minibatches* from it. This **reduces temporal correlation within minibatches** and reuses experience; the finite replay data are not independent samples from a fixed environment distribution.
- **Target network.** Compute the target with a *frozen* copy w^- , synced to w only every C steps:

$$\mathcal{L}(w) = \mathbb{E}_{(s,a,r,s') \sim D} \left[\left(r + \gamma \max_{a'} \hat{Q}(s', a'; w^-) - \hat{Q}(s, a; w) \right)^2 \right]$$

For each stored transition, the target is fixed between target-network updates. This helps stability but does not restore a general convergence guarantee.

Result: one architecture, one set of hyperparameters, **human-level play across dozens of Atari games** from raw pixels. The Bellman equation, untouched; the model, never learned; the table, replaced by a convnet.

DQN as regression — freeze the label, fit the prediction

For one stored nonterminal transition, let $r = 1$, $\gamma = 0.9$, and the target network's next values be **5 and 1**. The label is $y = 1 + 0.9(5) = 5.5$.

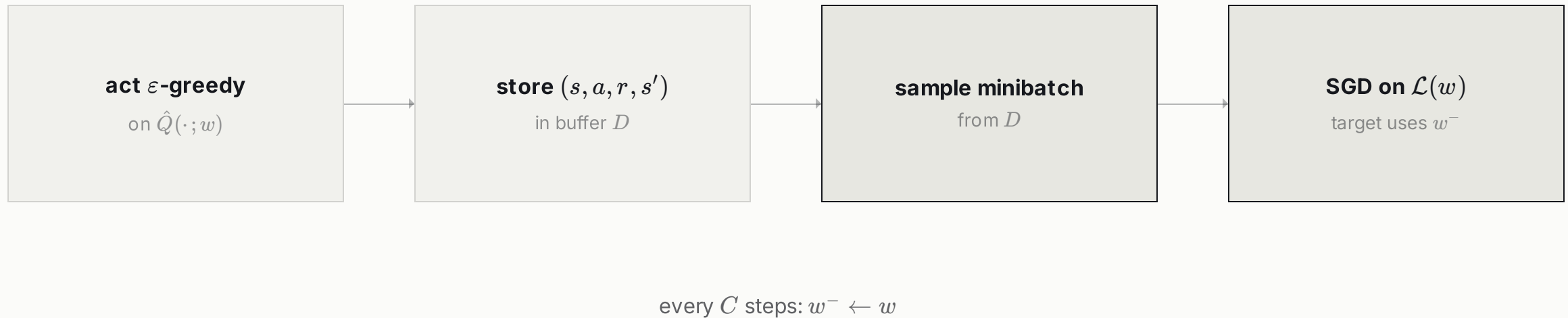
If the current network predicts 2:

$$\frac{1}{2}(y - Q_w(s, a))^2 = \frac{1}{2}(5.5 - 2)^2 = 6.125.$$

The derivative with respect to that prediction is $2 - 5.5 = -3.5$, so gradient descent pushes the prediction upward. **Do not differentiate through the label.**

Store a terminal flag d and use $y = r + \gamma(1 - d) \max_{a'} Q_{w^-}(s', a')$. For a true terminal transition, $y = r$. A time-limit truncation may still need bootstrapping.

The DQN loop — everything from today, assembled



Trace the lineage of each box: *act ϵ -greedy* (Act 2's exploration tax), *the max in the target* (Act 3's off-policy Q-learning), *bootstrap target* (Act 1's TD), *buffer* + *frozen w^-* (Act 4's triad fixes). One loop holds the whole lecture.

Double DQN — select with one estimate, evaluate with another

van Hasselt, Guez & Silver · AAAI 2016 · one nonterminal target calculation

At the next state, suppose the two networks give:

ACTION	ONLINE Q_w	TARGET Q_{w-}
A	8	4
B	7	6

For immediate reward 1 and discount 0.9:

$$y_{\text{DQN}} = 1 + 0.9 \max(4, 6) = 6.4.$$

Double DQN selects $a^* = \arg \max_a Q_w(s', a) = A$, then evaluates that action with the target network:

$$y_{\text{Double}} = 1 + 0.9 Q_{w-}(s', A) = 1 + 0.9(4) = 4.6.$$

Separating **selection** and **evaluation** reduces a source of overestimation; neither number is guaranteed correct. This is different from TD3's minimum of two critics in Lecture 10. [Double DQN](#)

What comes after DQN — one slide of horizon

The two tricks opened a decade of refinements, each patching a named flaw:

- **Double DQN** — the max over-estimates; decouple action-selection from evaluation. (*van Hasselt et al., 2016*)
- **Dueling networks** — split $\hat{Q}$ into state-value + advantage; learn "which states matter" separately. (*Wang et al., 2016*)
- **Prioritized replay** — sample high-error transitions more often. (*Schaul et al., 2016*)
- **Rainbow** — combine them; the combination improved the reported Atari benchmark results. (*Hessel et al., 2018*)

Every one keeps the same skeleton: *sampled Bellman backup* + Q for model-free greed + a stabilizer. None brings back the model — and their standard finite-action versions cannot simply enumerate a **continuous action space**. The maximization then needs extra structure or an optimizer, motivating an actor.

That wall — the max over a continuum — is exactly where [Lecture 9](#) begins.

Check — the three ingredients

The deadly triad names three things that together can make value learning diverge. Which set?

- ☐ A Exploration, discounting, and a large state space
- ☐ B Non-stationarity, correlated samples, and a moving target network
- ☐ C High learning rate, sparse reward, and long horizon
- ☒ D Function approximation, bootstrapping, and off-policy training

All three together can send an estimate to infinity even when all rewards and true values are zero. Removing a leg helps in the displayed example; it does not prove every remaining algorithm converges. DQN's two tricks are best read as attacks on this: a replay buffer decorrelates the updates and a frozen target network weakens the bootstrap. Neither removes the third leg, which is why deep RL remains fragile.

— CLOSING

Closing

Where we are — the cell, filled

	MODEL - BASED	MODEL - FREE
Static, single	optimization (<i>Lec 1</i>)	bandit (<i>Lec 8, intro</i>)
Dynamic, discrete, single	MDP / DP (<i>Lec 7 ✓</i>)	value-based RL (<i>Lec 8 ✓</i>)
Dynamic, continuous, single	optimal control (<i>Lec 9</i>)	policy-based RL (<i>Lec 10</i>)

One repeated move, three times over: **model** → **sample, wait** → **bootstrap, table** → **function**.

Lecture 7 could solve the Bellman equation,
because it owned the world. Lecture 8 *learned* to
solve it.

by sampling the expectation, storing Q instead of V , and replacing the table with a network whose stability and performance must be checked.

Questions?

From a windy gridworld to human-level Atari with one update rule — the distance between *knowing* the dynamics and *never needing to*. What stays fixed through all of it is the equation Bellman wrote; we only ever changed what we were allowed to look up.

— APPENDIX

Backup slides

Complete arguments, kept out of the narrative.

Reading guide — from a tabular target to a learned value function

one main idea to explain, one comparison, one application

ROLE	READ OR REVISIT	QUESTION TO ANSWER
Core	Mnih et al., <i>Human-level control through deep reinforcement learning</i> (Nature 2015)	What lets the Q-learning target train a network from replayed images?
Compare	van Hasselt, Guez & Silver, <i>Deep Reinforcement Learning with Double Q-learning</i> (AAAI 2016)	Why separate selecting an action from evaluating its value?
Apply	The original windy gridworld, six rooms and Atari examples	Which issue is about the target, and which is about state representation?

Use Sutton–Barto Chapters 5–6 for MC and TD prerequisites. DQN is the main research case; Double DQN explains a specific failure. Dueling, prioritized replay and Rainbow remain a brief extension map.

Backup 1— MC vs. TD, the bias–variance ledger

Both estimate $V^\pi(s) = \mathbb{E}_\pi[G_t \mid s_t = s]$ from samples; they differ in *what* they sample.

MONTE CARLO TARGET $G_t = \sum_k \gamma^k r_{t+k+1}$

- *Bias*: none — $\mathbb{E}[G_t] = V^\pi(s_t)$ exactly.
- *Variance*: high — a function of **many** random rewards and transitions along the whole episode.

TD(0) TARGET $r_{t+1} + \gamma V(s_{t+1})$

- *Bias*: present — uses the current (imperfect) estimate $V(s_{t+1})$; unbiased at the true V^π (and sometimes when errors cancel in expectation).
- *Variance*: often lower — depends on one reward and one successor estimate; no universal ordering holds for arbitrary estimates.

THE BRIDGE — n -STEP AND TD(λ)

Interpolate by bootstrapping after n steps, $G_t^{(n)} = r_{t+1} + \dots + \gamma^{n-1}r_{t+n} + \gamma^n V(s_{t+n})$; $n = 1$ is TD(0), $n = \infty$ is MC. TD(λ) averages all n geometrically. The whole family is one dial between *low-bias / high-variance* and *higher-bias / low-variance*.

Backup 2 — when does tabular Q-learning converge?

Claim (*Watkins & Dayan, 1992*): in a finite stationary MDP with bounded rewards and $0 \leq \gamma < 1$, tabular Q-learning converges to Q^* with probability 1, provided

- every state–action pair (s, a) is visited **infinitely often** (the role of exploration, e.g. $\epsilon > 0$), and
- for each state–action pair, its update step sizes satisfy the Robbins–Monro conditions $\sum_t \alpha_t = \infty$, $\sum_t \alpha_t^2 < \infty$.

Why it holds (sketch). The Bellman optimality operator $(\mathcal{T}Q)(s, a) = \mathbb{E}_{s'}[r + \gamma \max_{a'} Q(s', a')]$ is a γ -contraction in the sup-norm, so it has a unique fixed point Q^* . The Q-learning update is a **stochastic approximation** of $\mathcal{T}$: each step replaces the expectation by one sample. Robbins–Monro step sizes average out the sampling noise; infinite visitation guarantees every entry keeps being corrected. Together they drive $Q \rightarrow Q^*$.

What breaks under approximation. Replace the table by $\hat{Q}(\cdot; w)$ and the update is no longer a contraction in any norm the parameters respect — the projection onto the function class can *expand* distances. That is the formal root of the deadly triad, and the reason DQN needs its two crutches.

Backup 3 — the function-approximation gradient, in full

Linear case $\hat{Q}(s, a; w) = w^\top \phi(s, a) = \sum_k w_k \phi_k(s, a)$. Per-transition squared TD error:

$$\text{Err}(w) = \frac{1}{2} \left(r + \gamma \max_{a'} \hat{Q}(s', a'; w) - \hat{Q}(s, a; w) \right)^2$$

Differentiate, holding the target fixed (semi-gradient):

$$\frac{\partial \text{Err}}{\partial w_k} = - \left(r + \gamma \max_{a'} \hat{Q}(s', a'; w) - \sum_j w_j \phi_j(s, a) \right) \phi_k(s, a)$$

$$\Rightarrow w_k \leftarrow w_k + \alpha \left(r + \gamma \max_{a'} \hat{Q}(s', a'; w) - \hat{Q}(s, a; w) \right) \phi_k(s, a)$$

In vector form $w \leftarrow w + \alpha \delta \phi(s, a)$ — the tabular update is the special case $\phi(s, a) = \mathbf{e}_{(s,a)}$ (a one-hot indicator), which is why tabular RL is "just" function approximation with the most local possible features. For a neural net, ϕ becomes $\nabla_w \hat{Q}(s, a; w)$ and the same line holds.

Backup 4 — model-based RL, the alternative we set aside

The branch we acknowledged in Act 0 and walked away from. **Estimate the model** from the same stream of experience:

$$\hat{P}(s' \mid s, a) = \frac{\#(s, a, s')}{\#(s, a)}, \quad \hat{R}(s, a, s') = \text{mean of } r \text{ over } (s, a, \cdot, s')$$

Then solve the estimated MDP with any Lecture 7 method (value / policy iteration).

- *Pro — sample efficiency*: a learned model can be "replayed" indefinitely; each real transition informs the value of *many* states through planning.
- *Con — model bias*: errors in $\hat{P}$, $\hat{R}$ compound through planning; you optimize for a world that is slightly wrong.
- *Con — cost*: estimating a full kernel is far more than estimating the Q^* we actually act on.

This is why model-free dominated the deep-RL era — and why **Lecture 11** returns to model-based methods only once it can make the learned model carry its weight (planning, Dyna, differentiable control).

The translation table — our coordinate system for today

Every object of Lecture 7 has a *sampled* counterpart. Keep this in sight.

LECTURE 7 (MODEL-BASED)	LECTURE 8 (SAMPLED)	WHAT REPLACES THE MODEL
expectation $\sum_{s'} P(s' s, a)[\dots]$	one transition (s, a, r, s')	a sample average / a bootstrap
policy evaluation (solve with P, R)	MC or TD prediction	returns, or a one-step estimate
greedy on V: $\arg \max_a \sum_{s'} P[\dots]$	greedy on Q : $\arg \max_a Q(s, a)$	store Q , not V
value iteration	Q-learning	the max inside a sampled backup
full backup (all successors)	sample backup (one successor)	the law of large numbers
tabular, exact	function approximation (DQN)	a network $Q(s, a; w)$

Read column 3. Nothing here is a new *principle* — it is the same Bellman backup, with the one piece we don't own quietly swapped out. The art is in **what we swap it for** without the whole thing collapsing.

One honest detour — why not just learn the model?

The most literal idea: estimate P, R from data, then run Lecture 7.

$$\hat{P}(s' \mid s, a) = \frac{\#(s, a, s')}{\#(s, a)}, \quad \hat{R}(s, a, s') = \text{average } r \text{ over } (s, a, \cdot, s')$$

Plug in, solve the MDP. This is **model-based RL**, and it is perfectly valid.

But notice the waste: to choose actions we only ever need Q^* , yet here we first estimate a *whole transition kernel* — vastly more parameters than the thing we actually use. Why estimate P and R at all,

when we could estimate Q^* **directly**?

That question — *skip the model, estimate the value* — is the entire model-free programme. The rest of today walks it. (The model returns, deliberately, in Lecture 11.)