Value-Based Reinforcement Learning

Lecture 2

Peter Henderson

COS 435 / ECE 433

Thanks to helpful slides/notes by Ben Van Roy, Emma Brunskill, Ben Eysenbach, and Csaba Szepesvári.

Today's Agenda

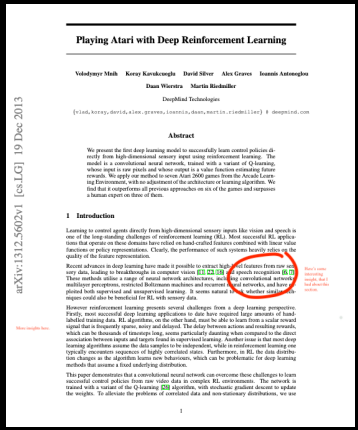
1. Recap: Value Iteration
2. Discussion: Speeding Up Value Iteration
3. Convergence of Asynchronous Value Iteration
4. Value Function Learning (Model-Free)
5. Bias-Variance Tradeoff in Multi-Step Backups
6. Q-Learning, SARSA, Expected SARSA
7. Deep Q-Networks (DQN) and Rainbow

Recap from Lecture 1

Logistics

Starting next week: submit reading responses.
Question about format:

1. One paragraph summary with one criticism and one observation per required paper.
2. A marked up pdf with your thoughts.



arXiv:1312.5602v1 [cs.LG] 19 Dec 2013

Playing Atari with Deep Reinforcement Learning

Volodymyr Mnih Koray Kavukcuoglu David Silver Alex Graves Ioannis Antonoglou
Daan Wierstra Martin Riedmiller
DeepMind Technologies
{vlad,koray,david,alex.graves,ioannis,daan,martin.riedmiller} @ deepmind.com

Abstract

We present the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. The model is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards. We apply our method to seven Atari 2600 games from the Arcade Learning Environment, with no adjustment of the architecture or learning algorithm. We find that it outperforms all previous approaches on six of the games and surpasses a human expert on three of them.

1 Introduction

Learning to control agents directly from high-dimensional sensory inputs like vision and speech is one of the long standing challenges of reinforcement learning (RL). Most successful RL applications that operate on these domains have relied on hand-crafted features combined with linear value functions or policy representations. Clearly, the performance of such systems heavily relies on the quality of the feature representation.

Recent advances in deep learning have made it possible to extract high-level features from raw sensory data, leading to breakthroughs in computer vision [11, 22, 16] and speech recognition [6, 7]. These methods utilise a range of neural network architectures, including convolutional networks, multilayer perceptrons, restricted Boltzmann machines and recurrent neural networks, and have exploited both supervised and unsupervised learning. It seems natural to ask whether similar techniques could also be beneficial for RL with sensory data.

However reinforcement learning presents several challenges from a deep learning perspective. Firstly, most successful deep learning applications to date have required large amounts of hand-labelled training data. RL algorithms, on the other hand, must be able to learn from a scalar reward signal that is frequently sparse, noisy and delayed. The delay between actions and resulting rewards, which can be thousands of timesteps long, seems particularly daunting when compared to the direct association between inputs and targets found in supervised learning. Another issue is that most deep learning algorithms assume the data samples to be independent, while in reinforcement learning one typically encounters sequences of highly correlated states. Furthermore, in RL the data distribution changes as the algorithm learns new behaviours, which can be problematic for deep learning methods that assume a fixed underlying distribution.

This paper demonstrates that a convolutional neural network can overcome these challenges to learn successful control policies from raw video data in complex RL environments. The network is trained with a variant of the Q-learning [23] algorithm, with stochastic gradient descent to update the weights. To alleviate the problems of correlated data and non-stationary distributions, we use

1

Here's some interesting insight, that I wouldn't have otherwise noticed.

Wow, insightful here.

Recap from Lecture 1

Recap: The MDP Framework

An MDP is a tuple $(\mathcal{S}, \mathcal{A}, T, R, \gamma)$:

- $\mathcal{S}$: State space $\mathcal{A}$: Action space
- $T(s'|s,a)$: Transition dynamics $R(s,a)$: Reward function
- $\gamma \in [0,1)$: Discount factor

Key Value Functions

- $V^\pi(s) = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty} \gamma^t r_{t+1} \mid s_0 = s\right]$ — *"How good is state s?"*
- $Q^\pi(s,a) = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty} \gamma^t r_{t+1} \mid s_0 = s, a_0 = a\right]$ — *"How good is action a in state s?"*

Relationship: $V^\pi(s) = \sum_a \pi(a|s) Q^\pi(s,a)$ **Optimal:** $\pi^*(s) = \arg\max_a Q^*(s,a)$

Recap: Bellman Equations

Value functions satisfy a recursive relationship: *Value now = Immediate reward + Discounted future value*

Bellman Expectation Equation (for policy π)

$$V^\pi(s) = \sum_a \pi(a|s) \left[R(s,a) + \gamma \sum_{s'} T(s'|s,a) V^\pi(s') \right]$$

Bellman Optimality Operator $\mathcal{B} : \mathbb{R}^{|\mathcal{S}|} \to \mathbb{R}^{|\mathcal{S}|}$

$$(\mathcal{B}V)(s) = \max_{a \in \mathcal{A}} \left[R(s,a) + \gamma \sum_{s'} T(s'|s,a) V(s') \right]$$

Recap: Value Iteration

Value Iteration Algorithm

Initialize: $V_0(s) = 0$ for all s

For $k = 0, 1, 2, \ldots$ until convergence ($\|V_{k+1} - V_k\|_\infty < \epsilon$):

For each state s:

$$V_{k+1}(s) = \max_a \left[R(s,a) + \gamma \sum_{s'} T(s'|s,a) V_k(s')\right]$$

In operator notation: $V_{k+1} = \mathcal{B} V_k$

Extract policy: $\pi(s) = \arg\max_a \left[R(s,a) + \gamma \sum_{s'} T(s'|s,a) V(s')\right]$

Recap: Contraction and Convergence

Theorem (Contraction Mapping)

$\|\mathcal{B}V - \mathcal{B}V'\|_\infty \leq \gamma \|V - V'\|_\infty$ for all $V, V' \in \mathbb{R}^{|\mathcal{S}|}$

Theorem (Convergence of Value Iteration)

The sequence $V_0, V_1, \ldots$ with $V_{k+1} = \mathcal{B}V_k$ converges to V^*:
$\|V^* - V_k\|_\infty \leq \gamma^k \|V^* - V_0\|_\infty \to 0$

Convergence rate is geometric: error shrinks by factor γ per iteration.

Discussion: Speeding Up Value Iteration

Discussion: Speeding Up Value Iteration

Turn to your neighbor and discuss/recall:

Question

What strategies can we use to speed up convergence of value iteration?

Take 3–4 minutes to brainstorm with your neighbor.

Strategies for Speeding Up Value Iteration

1. Gauss-Seidel VI: Update states in order, use new values immediately.

- When computing $V(s_i)$, use already-updated $V(s_1), \ldots, V(s_{i-1})$
- Information propagates *within* a single sweep — often converges faster

2. Asynchronous VI: Update states in any order, even in parallel.

- Each processor updates its own subset of states
- Converges as long as every state is updated infinitely often

3. Prioritized Sweeping: Focus updates on states with the largest Bellman error.

- Maintain a priority queue ordered by $|V(s) - (\mathcal{B}V)(s)|$
- Focus computation where it matters most

Convergence of Asynchronous Value Iteration

Discussion: Proving Asynchronous Convergence

Turn to your neighbor and discuss:

Question

Under what conditions does asynchronous value iteration converge? Under what conditions does it not?

Take 2–3 minutes to brainstorm with your neighbor.

Asynchronous Value Iteration

Theorem (Asynchronous Convergence)

Fix a finite MDP $(\mathcal{S}, \mathcal{A}, T, R)$ and $\gamma \in [0, 1)$. If $\mathcal{S}_0, \mathcal{S}_1, \ldots$ is a sequence of subsets of $\mathcal{S}$ such that each state $s \in \mathcal{S}$ appears infinitely often, then for any V_0:

$$V_{k+1}(s) = \begin{cases} (\mathcal{B}V_k)(s) & s \in \mathcal{S}_k \\ V_k(s) & s \notin \mathcal{S}_k \end{cases}$$

converges to V^*.

The contraction property plus visiting all states infinitely often does the heavy lifting — even partial updates make progress toward V^*.

Proof of Asynchronous Convergence

Proof: Define $e_k = \|V^* - V_k\|_\infty$. We show $e_k \to 0$.
For any updated state $s \in \mathcal{S}_k$:

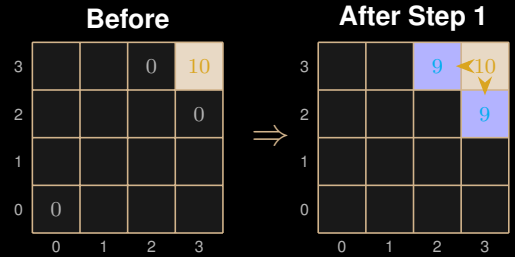
$$|V^*(s) - V_{k+1}(s)| = |(\mathcal{B}V^*)(s) - (\mathcal{B}V_k)(s)| \leq \gamma \|V^* - V_k\|_\infty = \gamma\, e_k$$

For non-updated states $s \notin \mathcal{S}_k$: $|V^*(s) - V_{k+1}(s)| = |V^*(s) - V_k(s)| \leq e_k$.
So $e_{k+1} \leq e_k$ (error never increases). But we need it to *strictly* decrease.
Since each state appears infinitely often, for any s there exists $k_s > k$ with $s \in \mathcal{S}_{k_s}$:

$$|V^*(s) - V_{k_s+1}(s)| \leq \gamma\, e_{k_s} \leq \gamma\, e_k$$

After a "full cycle" where every state has been updated at least once, the max error contracts by at least γ. Repeating: $e \to 0$. □

Async Value Propagation: First Wave



Setup: 4×4 grid, deterministic. Goal $(3,3) = +10$, $\gamma = 0.9$. All other values start at 0.

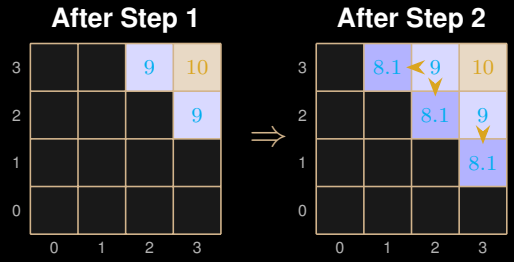
Step 1: Update $(2,3)$ and $(3,2)$ — the goal's neighbors:

$$V(2,3) = \gamma \cdot V(3,3)$$
$$= 0.9 \times 10 = \mathbf{9}$$

Similarly $V(3,2) = 9$.

Crucially: These **fresh values** are available *immediately* for the next update — we don't wait for a full sweep to finish.

Async Value Propagation: Values Continue to Back Up



Step 2: Update the *next ring* — $(1,3)$, $(2,2)$, $(3,1)$:

$$V(1,3) = \gamma \cdot V(2,3) = 0.9 \times \mathbf{9} = \mathbf{8.1}$$

The **9** came from Step 1 — *not* the old value of 0.

Connector effect: $(2,3)$ and $(3,2)$ act as **connectors** — once they receive value from the goal, they immediately relay it deeper into the grid.

Intuition/Analogy: Ripples

In async VI, you can have different “ripples” connect if you do infinite passes leading to convergence.

From Planning to Learning

From Planning to Learning

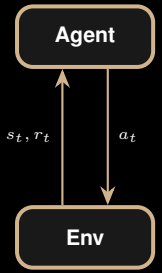
So far: **Dynamic programming** — assumes known T and R. But in most real RL problems, we don't know the dynamics or reward!

Key Question

How do we compute value functions and policies **from data**, without a model?

Two big ideas:

1. **Value function learning**: Estimate V^π or Q^π from trajectories
2. **Control without a model**: Find Q^* from interaction data



The agent-environment loop: learn from interaction, not a model.

Value Function Learning

Monte Carlo Estimation of Value Functions

Goal: Given trajectories from policy π, estimate $V^\pi(s)$.
Idea: For each state s visited, compute the **actual return** that followed, then average.
Given trajectory $\tau = (s_0, r_0, s_1, r_1, \ldots)$: $G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$

Monte Carlo Update (First-Visit)

For each first visit to state s in an episode with observed return G_t:

$$V(s_t) \leftarrow \mathbb{E}[G_t]$$

Properties: Unbiased estimate of V^π, but high variance — requires many full episodes.

Temporal Difference (TD) Learning

Key insight: We don't need to wait for the end of an episode! Use the **Bellman equation** to bootstrap: update predictions with predictions.

TD(0) Update

After observing transition (s, a, r, s'): $V(s) \leftarrow V(s) + \alpha\Big(\underbrace{r + \gamma V(s')}_{\text{TD target}} - V(s)\Big)$

The quantity $\delta = r + \gamma V(s') - V(s)$ is the **TD error**.

MC vs. TD: Bias-Variance Tradeoff

Monte Carlo

- Target: $G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots$
- **Unbiased** estimate of $V^\pi(s)$
- **High variance**: many stochastic terms
- Requires complete episodes

TD(0)

- Target: $r + \gamma V(s')$
- **Biased**: bootstraps off current V
- **Lower variance**: one reward, one transition
- Can update every step

The Fundamental Tradeoff

MC uses long rollouts (low bias, high variance). TD uses short bootstraps (higher bias, lower variance). Neither is universally better — best choice depends on the problem.

Multi-Step Returns: Interpolating MC and TD

We can use n**-step returns** to smoothly interpolate:

$$G_t^{(n)} = r_t + \gamma r_{t+1} + \cdots + \gamma^{n-1} r_{t+n-1} + \gamma^n V(s_{t+n})$$

n	**Method**
$n = 1$	TD(0) — one reward + bootstrap
$n = k$	k-step TD — k rewards + bootstrap
$n = \infty$	Monte Carlo — all rewards, no bootstrap

n-step TD Update

$$V(s_t) \leftarrow V(s_t) + \alpha \left(G_t^{(n)} - V(s_t) \right)$$

Larger n: less bias, more variance. Smaller n: more bias, less variance.

Bias-Variance Tradeoff: Kearns & Singh

Formalizing the Bias-Variance Tradeoff

To prove the bound, Kearns & Singh analyze a clean "phased" version of TD(k):
In each phase t, for every state s:

1. Collect n independent trajectories of length k from s under π
2. Each trajectory i produces random rewards $r_0^i, r_1^i, \ldots, r_{k-1}^i$ with $r_\ell^i \in [-1, 1]$
3. Update by averaging over all n trajectories:

$$\hat{V}_{t+1}(s) \;=\; \frac{1}{n}\sum_{i=1}^{n}\Bigg[\underbrace{r_0^i + \gamma r_1^i + \cdots + \gamma^{k-1} r_{k-1}^i}_{k \text{ random rewards (each} \in [-1,1])} + \underbrace{\gamma^k \hat{V}_t(s_k^i)}_{\text{bootstrap}}\Bigg]$$

Meanwhile, the **true** value decomposes as:

$$V^\pi(s) \;=\; \mathbb{E}[r_0] + \gamma\,\mathbb{E}[r_1] + \cdots + \gamma^{k-1}\mathbb{E}[r_{k-1}] \;+\; \gamma^k\,\mathbb{E}[V^\pi(s_k)]$$

The update averages $\frac{1}{n}\sum_i \gamma^\ell r_\ell^i$ at each step ℓ — these are sample means of **bounded**, **independent** random variables concentrating around $\mathbb{E}[r_\ell]$. This lets us use a common trick in RL theory literature: Hoeffding's inequality.

Applying Hoeffding's Inequality

Since each $r_\ell^{(j)} \in [-1, 1]$, Hoeffding guarantees for each reward step ℓ:

$$P\left(\left|\frac{1}{n}\sum_{j=1}^{n} r_\ell^{(j)} - \mathbb{E}[r_\ell]\right| \geq \epsilon\right) \;\leq\; 2\exp\left(\frac{-2n^2\epsilon^2}{n \cdot 2^2}\right)$$

Solving for ϵ: Fix the probability of exceeding ϵ to be $\leq \delta$:

$$2\exp\left(\frac{-n\epsilon^2}{2}\right) = \delta \implies \frac{-n\epsilon^2}{2} = \log(\delta/2) \implies \epsilon = \sqrt{\frac{2\log(2/\delta)}{n}}$$

So by Hoeffding, with n samples and prob. $\geq 1 - \delta$:

$$\left|\frac{1}{n}\sum_{j} r_\ell^{(j)} - \mathbb{E}[r_\ell]\right| \;\leq\; \epsilon \;=\; \sqrt{\frac{2\log(2/\delta)}{n}}$$

Union bound over k steps: We need *all* k reward terms estimated to ϵ accuracy simultaneously. Require each to hold with prob. δ/k, so the probability we fail on *any* term is $\leq \delta$:

$$\epsilon \;=\; \sqrt{\frac{2\log(2k/\delta)}{n}}$$

Proof: Bounding the TD Error

Substitute into the k-step TD update definition:

$$\begin{aligned}\hat{V}_{t+1}(s) - V(s) &= \frac{1}{n}\sum_{j=1}^{n}\left(r_0^{(j)} + \gamma r_1^{(j)} + \cdots + \gamma^{k-1} r_{k-1}^{(j)} + \gamma^k \hat{V}_t(s_k^{(j)})\right) - V(s) \\ &= \sum_{\ell=0}^{k-1}\gamma^\ell\left(\frac{1}{n}\sum_j r_\ell^{(j)} - \mathbb{E}[r_\ell]\right) + \gamma^k\left(\frac{1}{n}\sum_j \hat{V}_t(s_k^{(j)}) - \mathbb{E}[V(s_k)]\right)\end{aligned}$$

Now upper bound the difference from $\mathbb{E}[r_\ell]$ by ϵ:

$$\begin{aligned}\hat{V}_{t+1}(s) - V(s) &\le \sum_{\ell=0}^{k-1}\gamma^\ell\epsilon + \gamma^k\left(\frac{1}{n}\sum_j \hat{V}_t(s_k^{(j)}) - \mathbb{E}[V(s_k)]\right) \\ &\le \epsilon\frac{1-\gamma^k}{1-\gamma} + \gamma^k\left(\frac{1}{n}\sum_j \hat{V}_t(s_k^{(j)}) - \mathbb{E}[V(s_k)]\right)\end{aligned}$$

The second term is bounded by Δ_{t-1}^{γ} by assumption. Hence:

$$\boxed{\Delta_t^{\gamma} \le \epsilon\frac{1-\gamma^k}{1-\gamma} + \gamma^k \cdot \Delta_{t-1}^{\gamma}}$$

■ variance (increases with k) ■ bias (decreases with k)

Iterating the Recurrence

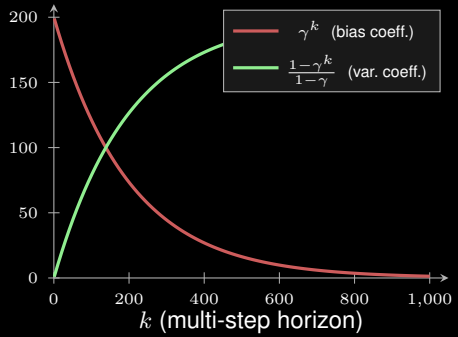
Substituting $\epsilon = \sqrt{2\log(2k/\delta)/n}$ and iterating from $\Delta_0^\gamma = 1$:

Full Learning Curve (Kearns & Singh)

$$\Delta_t^\gamma \leq \frac{1-\gamma^{kt}}{1-\gamma}\sqrt{\frac{2\log(2k/\delta)}{n}} + \gamma^{kt}$$

- **Bias** γ^{kt}: shrinks exponentially with k
- **Variance** $\frac{1}{1-\gamma}\sqrt{\cdots}$: irreducible error, grows with k
- As $t \to \infty$: only variance floor remains

Optimal: start with large k, decrease over time.



$(\gamma = 0.995)$

Why This Matters in Modern RL

The Kearns & Singh analysis reveals a fundamental tension:

- **Bias** from bootstrapping is hard to control — depends on how wrong your value function is, compounds through the Bellman backup chain
- **Variance** from sampling is **well-understood** — can get more rollouts to handle it.

Consequence for LLM-Based RL (e.g., GRPO)

Some RL for LLMs uses **Monte Carlo estimators** (full rollouts) rather than TD:

- Value function bias in high-dimensional LLM state spaces is extremely hard to diagnose or bound
- MC estimates are **unbiased by construction** — only error is variance
- Generating full rollouts is relatively cheap vs. training an accurate critic (at least for short horizons)

Discussion: MC vs. TD for LLM-Based RL

Quick Discussion:

Under what conditions might you want to switch from MC to TD with LLM+RL tasks?

Recall currently many people are using tasks like basic math questions and the horizon is until you output the answer.

What's an example of a task where you might want to go back to TD?

Take 2–3 minutes to brainstorm with your neighbor.

Break - 10 minutes

Q-Learning, SARSA, and Expected SARSA

Action Values: Extending to Q

Everything we've done with V extends naturally to **action value functions** Q.

Definitions: For policy π and optimal:

$$Q_\pi(s,a) = R(s,a) + \gamma \sum_{s'} T(s'|s,a)\, V_\pi(s') \qquad Q_*(s,a) = R(s,a) + \gamma \sum_{s'} T(s'|s,a)\, V_*(s')$$

Bellman operators for Q:

$$(F_\pi Q)(s,a) = R(s,a) + \gamma \sum_{s'} T(s'|s,a) \sum_{a'} \pi(a'|s')\, Q(s',a')$$

$$(FQ)(s,a) = R(s,a) + \gamma \sum_{s'} T(s'|s,a) \max_{a'} Q(s',a')$$

These are **contractions** — just like the V case:

$$\|F_\pi Q - F_\pi Q'\|_\infty \le \gamma \|Q - Q'\|_\infty \qquad \|FQ - FQ'\|_\infty \le \gamma \|Q - Q'\|_\infty$$

So value iteration **converges on** Q: $\lim_{k\to\infty} F_\pi^k Q = Q_\pi$ and $\lim_{k\to\infty} F^k Q = Q_*$.

From V to Q: Why Learn Q?

Problem: To extract a policy from V^*, we need the model!

$$\pi^*(s) = \arg\max_a \left[R(s,a) + \gamma \sum_{s'} T(s'|s,a) V^*(s') \right]$$

Solution: Learn $Q^*(s,a)$ instead — the policy comes for free:

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

Key Definitions: On-Policy vs. Off-Policy

When learning from experience, we distinguish between two policies:

Behavior Policy μ (also called **exploration policy**)

The policy used to **collect data** (select actions during interaction).

Target Policy π

The policy we are trying to **learn** or **evaluate**.

On-Policy ($\mu = \pi$)

- Learn about the *same* policy generating data
- Must balance exploration/exploitation

Off-Policy ($\mu \neq \pi$)

- Learn about a *different* policy
- Can reuse old data, explore freely

SARSA: On-Policy TD Control

SARSA estimates Q^π — the Q-values of the *current* policy. Given transition (s, a, r, s', a') where $a' \sim \pi(\cdot|s')$:

SARSA Update

$$Q(s,a) \leftarrow Q(s,a) + \alpha\Big(\underbrace{r + \gamma Q(s',a')}_{\text{TD target}} - Q(s,a)\Big)$$

Name: **S**tate, **A**ction, **R**eward, **S**tate, **A**ction — the quintuple used in each update.
Properties:

- **On-policy**: learns Q^π for the policy π that generated the data
- Uses a single sample of the next action a'
- Converges to Q^π under Robbins-Monro step sizes

Expected SARSA: Lower Variance, Off-Policy Capable

Idea: Instead of sampling a', take the expectation over next actions.
Given transition (s, a, r, s') and a policy π:

Expected SARSA Update

$$Q(s,a) \leftarrow Q(s,a) + \alpha\Big(r + \gamma \textstyle\sum_{a'} \pi(a'|s') Q(s',a') - Q(s,a)\Big)$$

Two key advantages over SARSA:

1. **Lower variance:** No randomness from sampling a' — we average over all actions
2. **Off-policy capable:** π in the expectation can differ from the data-collecting policy

SARSA vs. Expected SARSA: Bias-Variance Analysis

Key result: SARSA and Expected SARSA have the same bias but Expected SARSA has lower variance.

See Van Seijen, Harm, et al. “A theoretical and empirical analysis of Expected Sarsa.” ADPRL, 2009.

Blackboard proof.

Q-Learning: Learning Q^* Directly

Q-Learning (Watkins, 1989): Learn Q^* using the max over next actions. Given (s, a, r, s') from *any* behavior policy:

Q-Learning Update

$$Q(s,a) \leftarrow Q(s,a) + \alpha\Big(\underbrace{r + \gamma \max_{a'} Q(s',a')}_{\text{TD target}} - Q(s,a)\Big)$$

Key properties:

- **Off-policy**: Learns Q^* regardless of which policy collected the data
- Uses the Bellman *optimality* equation (with $\max$) not the expectation equation

Q-Learning: Complete Algorithm

Q-Learning with ϵ-Greedy Exploration

Initialize: $Q(s, a) = 0$ for all $s \in \mathcal{S}, a \in \mathcal{A}$
For each episode:

1. Initialize s
2. **For each step:**
   - Choose a via ϵ-greedy w.r.t. Q
   - Take action a, observe r, s'
   - $Q(s, a) \leftarrow Q(s, a) + \alpha\big[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\big]$
   - $s \leftarrow s'$
3. **Until** s is terminal

Convergence (Jaakkola et al., 1993; Tsitsiklis, 1994)

Converges to Q^* if: (1) each (s, a) updated ∞-often, (2) $\sum_t \alpha_t = \infty$, $\sum_t \alpha_t^2 < \infty$.

Q-Learning as Noisy Value Iteration

Key insight: Q-learning is a *stochastic* version of value iteration on Q-values.

Real-Time Value Iteration:

$$Q_{t+1}(s,a) \leftarrow (FQ_t)(s,a)$$

Computes the *full expectation*:

$$r(s,a) + \gamma \sum_{s'} P^a_{ss'} \max_{a'} Q_t(s',a')$$

Q-Learning:

$$Q_{t+1}(s,a) \leftarrow Q_t(s,a) + \alpha_t \big[\underbrace{r + \gamma \max_{a'} Q_t(s',a')}_{\approx FQ_t} - Q_t(s,a) \big]$$

Uses a *single transition* instead.

Why the Step Size α_t?

Without smoothing ($\alpha = 1$), single-sample updates cause Q-values to chatter. The step size **averages out noise** over many updates.

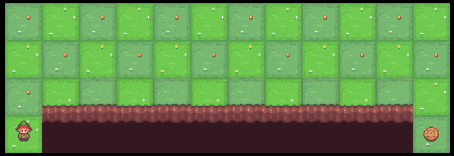
Comparing SARSA, Expected SARSA, and Q-Learning

	SARSA	Expected SARSA	Q-Learning
Target	$r + \gamma Q(s', a')$	$r + \gamma \sum_{a'} \pi(a'\|s') Q(s', a')$	$r + \gamma \max_{a'} Q(s', a')$
On/Off-policy	On-policy	Either	Off-policy
Learns	Q^π	Q^π	Q^*
Variance	Higher	Lower	Lower

Cliff Walking Example:
SARSA learns a *safe path* away from the cliff (accounts for ϵ-exploration risk). Q-learning learns the *optimal path* along the edge (exploration off-policy).

Exploration

All use ϵ-greedy: with prob ϵ random action, else $\arg\max_a Q(s, a)$.

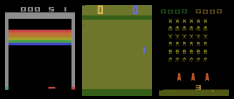


Cliff Walking (Gymnasium). Agent starts bottom-left, goal bottom-right. Brown = cliff.

Deep Q-Networks (DQN)

The Challenge: Scaling Beyond Tabular

Problem: Tabular methods store one value per (s, a) pair. For Atari: s = raw pixels ($210 \times 160 \times 3$) — the state space is astronomical.



Atari 2600: Breakout, Pong, Space Invaders. Raw pixels!

Solution: Use a **neural network** $Q_\theta(s, a)$ to approximate Q^*.

The Loss Function

$$\mathcal{L}(\theta) = \mathbb{E}\left[\left(Q_\theta(s,a) - \left(r + \gamma \max_{a'} Q_\theta(s', a')\right)\right)^2\right]$$

The Deadly Triad

Combining these three things can cause divergence:

1. **Function approximation** — neural net instead of table
2. **Bootstrapping** — target depends on current Q estimate
3. **Off-policy learning** — data from different policy than we're evaluating

Sutton & Barto (2018, Ch. 11.10)

"The potential for off-policy learning remains tantalizing, the best way to achieve it still a mystery."

DQN (Mnih et al., 2015) introduced two key tricks to tame this instability:

- **Experience replay**
- **Target networks**

DQN and Neural Network Function Approximation

Problem: Consecutive transitions are highly correlated.

Discuss with your neighbor: Why is this a problem? What are the consequences?

DQN Experience Replay

Problem: Consecutive transitions are highly correlated $\Rightarrow$ unstable SGD which assumes i.i.d. data.
Solution: Store transitions in a **replay buffer** $\mathcal{D}$ and sample random mini-batches.

Experience Replay

1. Store each transition (s, a, r, s') in buffer $\mathcal{D}$ (circular, fixed size)
2. Sample random mini-batch $\{(s_i, a_i, r_i, s_i')\} \sim \mathcal{D}$
3. Compute gradient on mini-batch and update θ

Benefits:

- **Breaks correlations:** Random sampling decorrelates training data
- **Data efficiency:** Each transition reused in multiple updates
- **Stability:** Smooths over changes in data distribution as policy changes

DQN: Target Networks

Problem: The target $r + \gamma \max_{a'} Q_\theta(s', a')$ changes every time we update θ.

$\Rightarrow$ We're chasing a moving target — makes optimization unstable.

Solution: Use a separate **target network** Q_{θ^-} with frozen weights.

DQN Target

$y = r + \gamma \max_{a'} Q_{\theta^-}(s', a')$. Update $\theta^- \leftarrow \theta$ every C steps, or soft: $\theta^- \leftarrow \tau\theta + (1-\tau)\theta^-$.

The full DQN loss:

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

DQN Pseudocode

Deep Q-Network Algorithm

Initialize: replay buffer $\mathcal{D}$, Q_θ with random weights, $\theta^- \leftarrow \theta$
For each episode:

1. Initialize state s_0 (stack of 4 frames)
2. **For each step t:**
   - Select a_t via ϵ-greedy w.r.t. Q_θ
   - Execute a_t, observe r_t, s_{t+1}
   - Store (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
   - Sample mini-batch $\{(s_i, a_i, r_i, s'_i)\}$ from $\mathcal{D}$
   - Compute targets: $y_i = r_i + \gamma \max_{a'} Q_{\theta^-}(s'_i, a')$
   - Update θ by SGD on $\sum_i (Q_\theta(s_i, a_i) - y_i)^2$
   - Every C steps: $\theta^- \leftarrow \theta$

DQN: Architecture and Results

Architecture:

- Input: stack of 4 raw frames (pixels)
- 3 conv layers $\rightarrow$ 2 FC layers
- Output: $Q(s, a)$ for all 18 actions
- Reward: change in game score
- Same architecture across all 49 games!

Key results (Mnih et al., 2015):

- Superhuman on 29/49 Atari games
- Learned from raw pixels
- No game-specific tuning

Discussion: Limitations of DQN

Quick Discussion:

1. What sorts of problems do you think might still exist in DQN?
2. What sorts of improvements do you think we can make?

Take 2–3 minutes to brainstorm with your neighbor.

DQN Improvements: Toward Rainbow

Double DQN: Fixing Overestimation Bias

Problem: $\max_{a'} Q(s', a')$ overestimates the true value because noise in Q gets amplified by the max operator.

Double Q-Learning (van Hasselt, 2010): Decouple action *selection* from action *evaluation*.

Double DQN Target (van Hasselt, Guez & Silver, 2016)

$$a^* = \arg\max_{a'} Q_\theta(s', a') \qquad y = r + \gamma Q_{\theta^-}(s', a^*)$$

Key idea: Use the **online network** Q_θ to select the best action, but evaluate it with the **target network** Q_{θ^-}. Different noise $\Rightarrow$ breaks overestimation.

Double Q-Learning: Full Algorithm

Double Q-Learning (van Hasselt, 2010)

Initialize: $Q^A(s,a)$, $Q^B(s,a)$ for all s, a; initial state s
Repeat:

1. Choose a based on $Q^A(s,\cdot)$ and $Q^B(s,\cdot)$; observe r, s'
2. Choose (e.g. random) either **UPDATE(A)** or **UPDATE(B)**:
   - **If UPDATE(A):** $a^* = \arg\max_{a'} Q^A(s',a')$
     $$Q^A(s,a) \leftarrow Q^A(s,a) + \alpha(s,a)\big[r + \gamma Q^B(s',a^*) - Q^A(s,a)\big]$$
   - **Else if UPDATE(B):** $b^* = \arg\max_{a'} Q^B(s',a')$
     $$Q^B(s,a) \leftarrow Q^B(s,a) + \alpha(s,a)\big[r + \gamma Q^A(s',b^*) - Q^B(s,a)\big]$$
3. $s \leftarrow s'$

Until end.

Action *selection* uses one network; action *evaluation* uses the other $\Rightarrow$ reduces overestimation bias.

Prioritized Experience Replay

Problem: Uniform sampling from replay buffer wastes time on “easy” transitions.
Idea (Schaul et al., 2016): Sample transitions proportional to their **TD error**:

$$p_i \propto |\delta_i|^\alpha \quad \text{where} \quad \delta_i = r_i + \gamma \max_{a'} Q_{\theta^-}(s_i', a') - Q_\theta(s_i, a_i)$$

Transitions where the agent is “most wrong” get replayed more often.
Importance sampling correction: To compensate for non-uniform sampling:

$$w_i = \left(\frac{1}{N \cdot p_i} \right)^\beta$$

Anneal $\beta \to 1$ over training to remove bias.

Double DQN with Proportional Prioritization

Algorithm: Double DQN with Proportional Prioritization (Schaul et al., 2016)

Input: minibatch k, step-size η, replay period K, size N, exponents α, β, budget T
Initialize: replay memory $\mathcal{H} = \emptyset$, $\Delta = 0$, $p_1 = 1$. Observe s_0, choose $a_0 \sim \pi_\theta(s_0)$.
For $t = 1$ **to** T:

1. Observe s_t, r_t, γ_t
2. Store $(s_{t-1}, a_{t-1}, r_t, \gamma_t, s_t)$ in $\mathcal{H}$ with priority $p_t = \max_{i<t} p_i$
3. **If** $t \equiv 0 \pmod{K}$:
   - **For** $j = 1$ **to** k:
     - Sample transition $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$
     - $w_j = (N \cdot P(j))^{-\beta} / \max_i w_i$
     - $\delta_j = r_j + \gamma_j Q_{\theta^-}(s_j, a^*) - Q_\theta(s_{j-1}, a_{j-1})$, $a^* = \arg\max_a Q_\theta(s_j, a)$
     - $p_j \leftarrow |\delta_j|$; $\quad \Delta \leftarrow \Delta + w_j \delta_j \nabla_\theta Q_\theta(s_{j-1}, a_{j-1})$
   - $\theta \leftarrow \theta + \eta\Delta$, $\Delta \leftarrow 0$; periodically $\theta^- \leftarrow \theta$
4. Choose $a_t \sim \pi_\theta(s_t)$

Dueling Networks

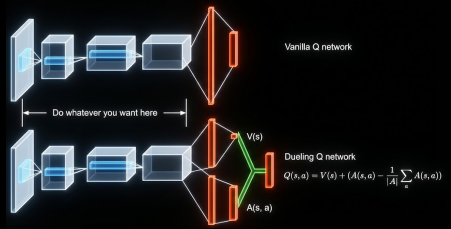
Observation: In many states, the *value of being in the state* matters more than which action you take.

Dueling DQN (Wang et al., 2016): Decompose Q into value and advantage:

Dueling Architecture

$$Q_\theta(s,a) = V_\theta(s) + \left(A_\theta(s,a) - \frac{1}{|\mathcal{A}|} \sum_{a'} A_\theta(s,a') \right)$$

- $V_\theta(s)$: how good is this state?
- $A_\theta(s,a)$: how much better is a than average?



Top: standard DQN. Bottom: dueling architecture with separate V and A streams. (Wang et al., 2016)

Distributional DQN and Classification-Based Values

Idea: Standard DQN learns the *mean* return via MSE regression. **Distributional RL** learns the full *distribution* (e.g. C51, QR-DQN). **Stop Regressing** (Farebrother et al., ICML 2024): train value functions via *classification* instead of regression.

Why classification? Discretize the value range into bins; use **categorical cross-entropy** instead of MSE. More scalable, robust to noisy targets and non-stationarity, reduces overfitting; SOTA on Atari, multi-task RL, robotics, and more.

Stop Regressing (Farebrother et al., 2024)

Discretize returns into K bins (atoms). Predict a *distribution* over bins $p_\theta(\cdot \mid s, a)$; $Q(s, a) = \sum_k v_k \, p_\theta(k \mid s, a)$. Train with cross-entropy to the (projected) Bellman target. Variants: **Two-Hot**, **HL-Gauss**, categorical.

Training Value Functions via Classification

Algorithm: Classification-Based Value Learning (Farebrother et al., 2024)

Require: number of bins K, bin support $\{v_1, \ldots, v_K\}$
Input: transition (x, a, r, x'), discount $\gamma \in [0, 1)$
Compute distributional Bellman target:

- $Q(x', a') := \sum_j v_j \, p_\theta(j \mid x', a')$ (mean over predicted distribution)
- $a^* \leftarrow \arg\max_{a'} Q(x', a')$
- Target distribution: $\mathcal{T} z_j := r + \gamma \, v_j$ for atoms v_j; project onto bin support (e.g. Two-Hot over neighboring bins)

Compute classification loss:

$$\mathcal{L} = -\sum_j \text{target}_j \log p_\theta(j \mid x, a)$$

Minimize $\mathcal{L}$ (categorical cross-entropy) w.r.t. θ. Target is the projected Bellman target distribution.

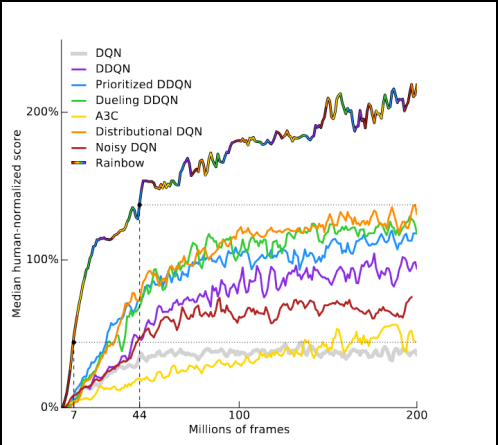
Value learning becomes classification over bins; cross-entropy replaces MSE. Scales to large networks and diverse domains (Atari, ResNets, Q-Transformers, chess, language agents)

Rainbow: Combining All the Improvements

Rainbow (Hessel et al., 2018): Combine **six** DQN improvements:

1. **Double DQN** — reduce overestimation bias
2. **Prioritized Replay** — focus on surprising transitions
3. **Dueling Networks** — separate state value from action advantage
4. **Multi-step Return Targets** — Predict a few steps ahead.
5. **Distributional RL** — learn the full *distribution* of returns, not just the mean
6. **Noisy Networks** — learned exploration via stochastic network layers (replaces ϵ-greedy)

Rainbow: Learning Curves



Rainbow achieves >200% median human-normalized score in 44M frames. (Hessel et al., 2018)

Rainbow: Detailed Results

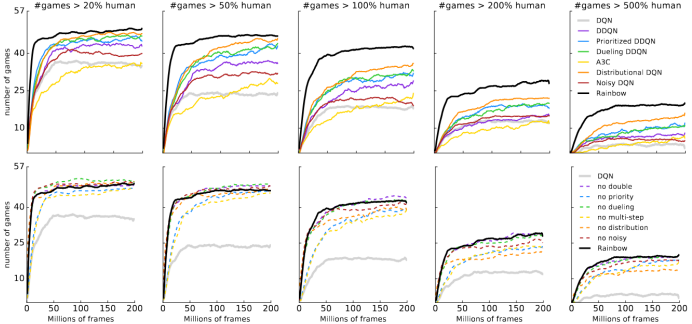


Figure 2: Each plot shows, for several agents, the number of games where they have achieved at least a given fraction of human performance, as a function of time. From left to right we consider the 20%, 50%, 100%, 200% and 500% thresholds. On the first row we compare Rainbow to the baselines. On the second row we compare Rainbow to its ablations.

Games achieving human performance thresholds. Top: Rainbow vs. baselines. Bottom: Ablations. (Hessel et al., 2018)

Under-Reported Trick: Classification Instead of Regression

Problem: MSE regression is unstable with noisy, non-stationary TD targets.
Solution: Discretize values into bins, predict a categorical distribution, use cross-entropy.

How It Works

1. Discretize $[V_{\min}, V_{\max}]$ into m bins $z_1, \ldots, z_m$
2. Network $\rightarrow$ softmax $\rightarrow$ probs $\hat{p}_i$ over bins
3. Target: two-hot encoding (interpolate between neighboring bins)
4. Cross-entropy loss; recover: $Q = \sum_i \hat{p}_i \cdot z_i$

Why it helps:

- Handles noisy targets better
- Scales to larger networks
- Bounded gradients

Used in: C51, Rainbow, MuZero, R2D2, Agent57, scaled Atari/robotics.

Farebrother et al., "Stop Regressing: Training Value Functions via Classification," 2024.

Policy-Based vs. Value-Based Methods

Policy-Based vs. Value-Based: A Preview

Value-Based (today)

- Learn Q^*/V^*, derive policy
- E.g., Q-Learning, DQN
- Pro: Low variance, sample efficient
- Pro: Off-policy (replay buffers!)
- Con: Discrete actions only
- Con: Maximization bias

Policy-Based (next lecture)

- Learn π_θ directly
- E.g., REINFORCE, PPO
- Pro: Continuous actions
- Pro: Unbiased gradients
- Con: On-policy, high variance
- Con: Less sample efficient

Actor-Critic Methods

The best of both worlds: learn *both* a policy (actor) and a value function (critic). Examples: A2C, SAC, TD3. Next after policy gradients.

Summary

Summary: What We Covered Today

1. **Async VI convergence**: Contraction property guarantees convergence even with partial updates
2. **Model-free value learning**: MC (unbiased, high var.) vs. TD (biased, low var.), multi-step returns interpolate
3. **SARSA** / **Expected SARSA** / **Q-Learning**: Three TD control flavors differing in target computation
4. **DQN**: Experience replay + target networks tame the deadly triad
5. **Rainbow**: Six complementary improvements $\Rightarrow$ massive gains

Key Takeaways

1. The **Bellman equation** is the backbone of value-based RL — from tabular DP all the way to deep Q-networks.
2. **Bias-variance tradeoff** governs the choice between MC and TD methods; multi-step returns give a tunable knob.
3. **Q-learning** enables model-free, off-policy learning of optimal policies.
4. **Function approximation** (neural nets) is essential for scaling — but introduces instability that requires careful engineering (replay, target nets, etc.).

Questions?