Anton Nesterov · an@vski.sh · July 18, 2026

Deep Q-Networks

Reinforcement Learning · Deep Learning · Q-Learning

DQN is the algorithm that taught a single neural network to beat human players across 49 Atari games from raw pixels. This is the preface: what Q-learning is, why a network breaks it, and the three tricks that put it back together.

In 2015 a single neural network learned to play **49 Atari 2600 games** — Pong, Breakout, Space Invaders — at or above human level, from nothing but raw pixels and a score signal. It was the same architecture and the same hyperparameters for every game. Nobody told it what a paddle was, or a brick, or an invader; the only feedback it ever received was *the score went up* or *the score went down*. That system was the **Deep Q-Network**, or DQN, and it is the moment most people point to when they say modern reinforcement learning began (Mnih et al., *Nature* 2015).

This essay is the preface. Before any of the variants there is one idea to understand: **what Q-learning is, why a neural network breaks it, and the three tricks that put it back together.** Every DQN-flavoured paper since 2015 is a refinement of that core. Get the core straight and the rest is commentary.

> *The throughline is **a number called** Q. Q-learning is the rule that computes it from experience. A neural network is the function approximator that lets us scale Q to states we have never seen. The deadly triad is the reason that combination explodes, and experience replay, target networks, and Double DQN are the three fixes. Five ideas, one algorithm.*

Decisions with consequences that arrive later

A reinforcement-learning agent interacts with an environment in a loop: at each step it observes a **state** s, picks an **action** a, and receives a **reward** r plus a new state s'. Atari is the canonical example. The state is the last four video frames (so the agent can infer motion), the action is one of the joystick buttons (typically 4 to 18 discrete actions), and the reward is the change in game score — `+1` for a point won in Pong, `-1` for a life lost in Breakout.

What makes this hard is the time lag. The reward you get *now* is rarely the true value of the action you just took. In Breakout, the move that matters is tunnelling through the side of the wall — an action whose payoff arrives dozens of frames later when the ball ricochets along

the top, clearing rows. A policy that only ever chased the next reward would never dig the tunnel. The agent has to learn to take actions whose value is realised *in the future*.

This is the structural difference from supervised learning. There is no labelled dataset of "correct actions." The agent has to discover, from the sequence of rewards it actually receives, which actions in which states tend to pay off over the long run. The quantity that captures that long-run value is called **Q**.

Q-learning

Q is a number for every (state, action) pair.

Define $Q(s, a)$ as the **expected total discounted reward** of taking action a in state s, and then behaving optimally forever after. "Discounted" means rewards further in the future count for less, weighted by a factor $\gamma \in [0, 1)$ per step:

$$Q(s, a) \;=\; \mathbb{E}\big[\, r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots \;\big|\; s_t = s,\; a_t = a \,\big]$$

If we knew Q exactly for every (s, a), the optimal policy would be trivial: in every state, pick the action with the largest Q. The whole problem is that we do not know Q. We have to *estimate* it from experience.

The optimal Q satisfies a recursive identity called the **Bellman equation**. The value of taking action a in state s is the immediate reward plus the discounted value of behaving optimally from the *next* state:

$$Q^*(s, a) \;=\; \mathbb{E}_{s'}\Big[\, r + \gamma \max_{a'} Q^*(s', a') \,\Big]$$

The expectation is over the environment's randomness — given s and a the next state s' is usually probabilistic. **Q-learning** turns this identity into a sample-based update rule. Every time the agent takes action a in state s and lands in s' with reward r, it nudges its current estimate $Q(s, a)$ toward the **Bellman target** $r + \gamma \max_{a'} Q(s', a')$:

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

The bracketed term is the **TD error** — the gap between the bootstrap estimate (looking one step ahead) and the current estimate. Push $Q(s, a)$ a fraction α of the way toward the target, repeat across thousands of transitions, and the estimates converge to Q^*. That is the whole of tabular Q-learning, and on small problems with discrete states it works beautifully (Sutton & Barto, *RL: An Introduction*, 2nd ed., Ch. 6).

Why a network, and why a network breaks it

Tabular Q-learning keeps one number per (s, a) pair. That is fine for a grid world with a hundred states. It is hopeless for Atari, where the state is four 84×84 greyscale frames.

The number of possible pixel configurations is astronomical; the agent will see almost none of them more than once. There is no way to fill a table.

The fix is the same fix every other field of ML reached for: **approximate** Q with a parameterised function $Q_\theta(s, a)$, where θ is a vector of parameters learned from data. Use a neural network — specifically, for visual input, a small **convolutional** network that takes the frames as input and outputs one Q-value per action. The Bellman target becomes the regression target, and the network's parameters are tuned by gradient descent to minimise the squared TD error:

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

This is **DQN**. The architecture is plain; the loss is just Q-learning written as a regression. The difficulty is that **it does not converge**. Naively trained, the loss explodes, the Q-values run off to infinity, and the policy collapses. This is the central obstacle of deep RL, and it has a name.

The deadly triad — why DQN diverges

Sutton & Barto call it the **deadly triad**: reinforcement learning with function approximation tends to diverge whenever three things are present at once (Sutton & Barto, 2nd ed., §11.3):

1. **Function approximation** — we are estimating Q with a neural network, not a table.
2. **Bootstrapping** — our target uses our *own* current estimates ($\max_{a'} Q_\theta(s', a')$), not ground truth.
3. **Off-policy data** — the transitions in the buffer were collected by an older behaviour policy, not the policy we are currently evaluating.

Remove any *one* of the three and convergence theory largely returns. DQN has all three by construction — that is the point of the algorithm — so the designer's job becomes *controlling* the instability rather than eliminating it. The 2015 paper's contribution was not the idea of putting Q-learning inside a neural network (people had tried). It was the three engineering tricks that made the combination actually train.

Trick 1 — experience replay (break the temporal correlation)

When the agent plays Atari, the transitions it experiences are **sequential** and **heavily correlated**. Frame t is almost identical to frame $t + 1$. Stochastic gradient descent assumes samples are roughly independent; feeding it a stream of near-duplicates produces biased, high-variance gradients that chase local correlations instead of the underlying return. There is also a pure efficiency argument: every transition costs an environment step, and throwing each one away after a single update is wasteful.

Experience replay is the fix. Instead of training on transitions as they arrive, store each (s, a, r, s') tuple in a large buffer — Mnih et al. used **1,000,000 transitions** — and at every training step sample a **random mini-batch** of 32 from it. Two things happen at once:

- The samples in a batch are now drawn from many different parts of many different episodes, so the temporal correlation is broken and the gradient behaves like an i.i.d. estimator.
- Each transition is **reused** many times for updates, which is a major sample-efficiency win.

The replay buffer also has a subtler effect: because the batch is drawn uniformly from a long history, the *data distribution* the network sees is an average over its past behaviour policies, not its current one. That is a form of off-policy learning, and it interacts with the deadly triad — but in practice the decorrelation benefit dominates.

Trick 2 — the target network (stop the bootstrap from moving)

Look again at the loss function. The target $r + \gamma \max_{a'} Q_\theta(s', a')$ contains the network's **own** parameters θ. Every gradient step changes θ, which changes the target, which changes the direction of the next gradient step. The thing the network is trying to fit is moving underneath it. Chasing a moving target is a classic source of instability in any bootstrapped regression; in deep RL it produces feedback loops where an overestimated Q-value in one part of the state space inflates targets in another, which inflates *its* neighbours, until the whole thing runs away.

The **target network** is a clean fix. Keep **two** copies of the network: the **online network** Q_θ that receives gradient updates, and a **target network** Q_{θ^-} whose parameters are held frozen. Compute the Bellman target from the frozen copy:

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

Now the bootstrap reference is fixed for a stretch of training, and the online network has a stable target to fit. Every C steps (Mnih et al. used $C = 10{,}000$), copy the online weights into the target network — a **hard update** — and the target jumps forward to a more recent estimate. The jumping introduces a small amount of staleness on purpose, in exchange for short-term stability. (A smoother variant, **soft / Polyak updating**, blends the weights a tiny amount every step: $\theta^- \leftarrow \tau\theta + (1-\tau)\,\theta^-$ with $\tau \ll 1$. This was popularised later by DDPG and is now standard in the actor-critic family.)

Experience replay and target networks together were enough to make DQN converge on Atari. That was the 2015 result. Everything below is *the same algorithm, with one specific failure mode patched at a time*.

Exploration — being greedy most of the time, random some of the time

A DQN is only as good as the transitions in its buffer. If the agent only ever picks the action that currently looks best, it will keep repeating the first thing that looked good and never discover that a different action pays off better. The standard fix is ε-**greedy exploration**:

$$a_t = \begin{cases} \text{uniformly random action} & \text{with probability } \varepsilon \\ \arg\max_a Q_\theta(s,a) & \text{with probability } 1-\varepsilon \end{cases}$$

The schedule matters. Start with $\varepsilon = 1.0$ (full random, the agent is exploring) and **anneal** it down to a small floor over the first portion of training. Mnih et al. annealed linearly from 1.0 to **0.1** over the first million frames. (Many implementations drop the floor to 0.05; the exact value is a hyperparameter, not a law.) The idea is intuitive: explore a lot at the start when the network knows nothing, then progressively commit to what you have learned. The agent is still always off-policy — the buffer collects exploratory moves, but the network is trained to predict the value of the *greedy* action.

ε-greedy is not elegant. It explores blindly, in random directions, regardless of where the model is uncertain. But it is simple, robust, and was the exploration strategy in the original DQN paper. More sophisticated methods (Noisy Networks, parameter-space noise, count-based curiosity bonuses) replace it; they are not in the core algorithm.

Trick 3 — Double DQN (stop the max from lying)

The Bellman target takes a `max` over actions: $\max_{a'} Q(s',a')$. The `max` of noisy numbers is itself biased upward. If Q_θ is even slightly over-optimistic about some action in state s', that action wins the `max`, and the inflated value flows back as the target for $Q_\theta(s,a)$. Next update, more actions in more states are slightly inflated, the `max` picks those, the bias compounds. This is the **overestimation bias** of Q-learning, and it is severe enough on Atari to noticeably hurt the final policy (van Hasselt, Guez, & Silver, AAAI 2016).

The diagnosis is that the *same* noisy estimate is used twice: once to **select** the best action, and again to **evaluate** it. **Double DQN** splits the two roles between the two networks we already have:

$$a^* = \arg\max_{a'} Q_\theta(s',a') \qquad \text{(online net picks the action)}$$

$$y = r + \gamma\, Q_{\theta^-}(s',a^*) \qquad \text{(target net evaluates it)}$$

The online network — which has the freshest information — decides which action is best in s'. The target network — which is frozen, with no particular incentive to be over-optimistic about that specific action — scores it. The two networks' noises are now only weakly correlated, and the upward bias shrinks dramatically. The cost is one extra forward pass per update. The benefit is policies that are noticeably better and noticeably less prone to the run-

away-Q-value pathology. In modern code, “DQN” almost always means Double DQN; the original is treated as a known-buggy baseline.

The full picture, in one loop

With all three tricks in place, DQN is one tight training loop. Put a description in pseudocode:

```
algorithm Double DQN is
    inputs: replay buffer D (capacity 1e6), online net Q_θ, target net Q_θ⁻
    copy θ⁻ ← θ                                    // target starts equal to online
    for each episode do
        s ← env.reset()
        for t = 1 to T do
            a ← ε-greedy(Q_θ, s)                   // explore early, exploit late
            take a, observe r, s'
            D.append((s, a, r, s'))                // log transition

            batch ← uniform_sample(D, 32)          // experience replay
            for each (s_b, a_b, r_b, s'_b) in batch do
                a* ← argmax_a' Q_θ(s'_b, a')       // Double: online picks
                y  ← r_b + γ · Q_θ⁻(s'_b, a*)      // Double: target evaluates
                                                       // (zero the bootstrap if s'_b
is terminal)
            gradient step on L(θ) = mean[ (y - Q_θ(s_b, a_b))² ]

            every C steps: θ⁻ ← θ                  // hard target update
            s ← s'
```

That is the algorithm. Every block in the pseudocode is one of the pieces above: ε-greedy for exploration, the replay buffer for decorrelation, the target network for a stable bootstrap, and the decoupled argmax/evaluate split for Double DQN. Read it once and you have read the 2015 Nature paper plus its 2016 follow-up.

What came after

The reason DQN spawned a literature is that each of its components has a visible weakness, and a paper exists for almost every one of them. A brief map:

- **Prioritized Experience Replay** (Schaul et al., 2016) replaces uniform sampling from the buffer with sampling proportional to the magnitude of the TD error. The transitions that *surprised* the network the most get replayed more often. It beat uniform-replay DQN on 41 of the 49 Atari games.
- **Dueling DQN** (Wang et al., ICML 2016) splits the network’s output head into a state-value stream $V(s)$ and an advantage stream $A(s, a)$, combined as $Q(s, a) = V(s) + A(s, a)$. This lets the network learn *which states are good* without having to evaluate

every action in states where the action choice barely matters — useful when there are many actions and most of them are equivalent.

- **Rainbow** (Hessel et al., AAAI 2018) is the audit paper: it bundles six DQN extensions together, then ablates them one at a time to see which ones actually carry the performance. (Spoiler: PER and Double DQN do most of the work; multi-step returns add a meaningful chunk; the others are marginal.)
- **Distributional RL** (C51, QR-DQN, IQN) replaces the single number $Q(s, a)$ with a *distribution* over possible returns. The mean is still what the agent acts on, but predicting the distribution is a richer learning signal that empirically helps.

The common shape is: *DQN works, here is one specific way it is suboptimal, here is the patch.* There is no DQN v2 that replaced it; there is just a stack of well-justified tweaks, of which Double DQN is the only one that has become effectively mandatory.

What DQN is *not* good at

A short tour of the boundaries, because the algorithm is oversold as often as it is undersold.

- **Continuous action spaces.** The Bellman target takes an `argmax` over actions, which is cheap when there are 4 or 18 of them and impossible when there are infinitely many. Continuous control (robotics) uses a different family entirely — DDPG, TD3, SAC — that learns a separate *actor* policy instead of maximising.
- **Sample efficiency.** DQN is a glutton for data. The Nature paper trained for 200 million frames, roughly 38 days of real-time play per game. If each environment step is expensive — a real robot, a chemistry simulation, anything where you cannot just spin up thousands of parallel copies — DQN is the wrong tool. Model-based methods and actor-critic methods with off-policy correction (SAC, etc.) are typically far more sample-efficient.
- **Multi-agent settings.** DQN assumes a stationary environment. Add a second agent that is also learning, and the environment becomes non-stationary from the first agent's point of view. Q-values chase a moving distribution and the convergence story breaks. The multi-agent variant (independent DQN) is widely used in practice but is known to be unstable.
- **Sparse rewards with no shaping.** If the agent almost never reaches the goal, the reward signal is effectively zero and there is nothing to learn from. This is the same problem the bandits literature calls the cold-start freeze, sharper. Solutions exist (curiosity-driven exploration, hindsight experience replay, reward shaping) but they are not in the vanilla algorithm.

Where this sits

DQN is the simplest interesting point in deep RL. Below it is tabular Q-learning, which is pedagogically clean but cannot scale. Beside it are the policy-gradient methods (REINFORCE, PPO, A3C), which learn a stochastic policy directly instead of a value function and dominate continuous control and modern LLM alignment. Above it are the actor-critic methods (SAC, TD3), which carry both a policy and a value function and currently set the state-of-the-art on most continuous benchmarks.

What makes DQN worth learning first is that every difficulty it has is a difficulty *all* of deep RL has, just exposed in their purest form. The deadly triad does not go away in PPO — PPO just controls it with clipped importance weights instead of a target network. Exploration does not get easier in SAC — SAC just replaces ε-greedy with maximum-entropy randomism. Overestimation does not vanish in actor-critic — TD3 invented its own twin-critic trick to handle it. Once you can read the DQN pseudocode above and point to where each instability lives and which trick addresses it, the rest of the field is variants on a theme you already know.

References

Primary sources

- Mnih, V. et al. *Human-level control through deep reinforcement learning*, Nature 518, 2015. The DQN paper. https://www.nature.com/articles/nature14236. Full PDF: https://web.stanford.edu/class/psych209/Readings/MnihEtAlHassibis15NatureControlDeepRL.pd
- van Hasselt, H., Guez, A., & Silver, D. *Deep Reinforcement Learning with Double Q-Learning*, AAAI 2016. https://arxiv.org/abs/1509.06461.
- Schaul, T., Quan, J., Antonoglou, I., & Silver, D. *Prioritized Experience Replay*, ICLR 2016. https://arxiv.org/abs/1511.05952.
- Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M., & Silver, D. *Dueling Network Architectures for Deep Reinforcement Learning*, ICML 2016. https://arxiv.org/abs/1511.06581.
- Hessel, M. et al. *Rainbow: Combining Improvements in Deep Reinforcement Learning*, AAAI 2018. https://arxiv.org/abs/1710.02298.
- Lillicrap, T. et al. *Continuous Control with Deep Reinforcement Learning* (DDPG; introduces Polyak/soft target updates), ICLR 2016. https://arxiv.org/abs/1509.02971.

Textbook & pedagogical

- Sutton, R. & Barto, A. *Reinforcement Learning: An Introduction*, 2nd ed., MIT Press, 2018. Ch. 6 (TD learning, Q-learning), Ch. 11 (the deadly triad). Free PDF: https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf. Authors' page: http://incompleteideas.net/book/the-book-2nd.html.

- OpenAI Spinning Up in Deep RL (Joshua Achiam). The standard free introduction to deep-RL methods and their taxonomy. https://spinningup.openai.com/en/latest/.
- Berkeley CS188 textbook, Model-Free Learning (TD, Q-learning). https://inst.eecs.berkeley.edu/~cs188/textbook/rl/mfl.html.
- van Hasselt, Doron, Mnih et al. *Deep Reinforcement Learning and the Deadly Triad* (arXiv, 2018). Empirical study of when and why DQN diverges. https://arxiv.org/abs/1812.02648.