Deep Q-Network

Deep Reinforcement Learning

Sai Sampath Kedari
sampath@umich.edu

Contents

1 Tabular Q-Learning

1.1 Bellman Optimality Operator

Consider a finite discounted Markov decision process with finite state and action spaces and discount factor $\gamma \in [0, 1)$. The optimal state-action value function satisfies the Bellman optimality equation

$$Q^*(s, a) = r(s, a) + \gamma \sum_{s'} p(s' \mid s, a) \max_{a' \in \mathcal{A}} Q^*(s', a') \tag{1}$$

$$= \mathbb{E}_{(S_{t+1}, R_{t+1}) \sim p(\cdot, \cdot \mid s, a)} \left[R_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q^*(S_{t+1}, a') \right]. \tag{2}$$

Define the Bellman optimality operator $T : \mathcal{Q} \to \mathcal{Q}$ by

$$\boxed{(TQ)(s, a) = \mathbb{E}_{(S_{t+1}, R_{t+1}) \sim p(\cdot, \cdot \mid s, a)} \left[R_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q(S_{t+1}, a') \right].} \tag{3}$$

The optimal state-action value function is the fixed point of this operator:

$$\boxed{TQ^* = Q^*.} \tag{4}$$

For $\gamma < 1$, the operator is a γ-contraction under the supremum norm:

$$\boxed{\|TQ_1 - TQ_2\|_\infty \leq \gamma \|Q_1 - Q_2\|_\infty.} \tag{5}$$

By the Banach fixed-point theorem, T therefore has a unique fixed point, namely Q^*, and from any initial Q_0 the iteration

$$Q_{k+1} = TQ_k \tag{6}$$

generates

$$Q_0 \xrightarrow{T} Q_1 \xrightarrow{T} Q_2 \xrightarrow{T} \cdots \xrightarrow{T} Q^*.$$

In the model-based setting, finding Q^* is thus equivalent to finding the fixed point of T by value iteration.

1.2 From Q-Value Iteration to Model-Free Q-Learning

The update $Q_{k+1} = TQ_k$ requires exact evaluation of the Bellman optimality operator, and therefore knowledge of the environment dynamics $p(\cdot, \cdot \mid s, a)$. In model-free reinforcement learning these dynamics are unknown, and the Bellman expectation is estimated from sampled transitions.

At time t, let the behaviour policy μ_t select

$$A_t \sim \mu_t(\cdot \mid S_t),$$

and let the environment generate

$$(S_{t+1}, R_{t+1}) \sim p(\cdot, \cdot \mid S_t, A_t).$$

From this single transition, define the sampled Bellman target

$$\boxed{Y_t = R_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q_t(S_{t+1}, a').} \tag{7}$$

For every fixed state-action pair (s, a),

$$\boxed{\mathbb{E}\left[Y_t \mid S_t = s,\, A_t = a\right] = (TQ_t)(s, a),} \tag{8}$$

so Y_t is a one-sample estimate of the Bellman optimality backup $(TQ_t)(S_t, A_t)$.

Q-learning moves the observed entry a fraction of the way toward this target,

$$\boxed{Q_{t+1}(S_t, A_t) = Q_t(S_t, A_t) + \alpha_t \left[R_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q_t(S_{t+1}, a') - Q_t(S_t, A_t)\right].} \tag{9}$$

Under the Robbins–Monro conditions

$$\sum_t \alpha_t = \infty, \qquad \sum_t \alpha_t^2 < \infty,$$

together with infinite visitation of every state-action pair, these iterates converge to Q^* with probability one.

Two properties are important for the transition to function approximation.

First, tabular Q-learning represents every state-action value by an independent coordinate. Updating $Q_t(S_t, A_t)$ therefore leaves every other table entry unchanged. Once the sampled target Y_t has been computed, it is held fixed during the current arithmetic update. In a self-transition, the updated coordinate may nevertheless appear in a future recomputed target, so tabular stability should not be attributed to targets that never change. It follows instead from the coordinate representation, the contraction of the Bellman optimality operator, and stochastic approximation under the preceding conditions.

Second, the behaviour policy determines which state-action pairs are sampled, but the Q-learning target applies the Bellman optimality operator and does not contain the probability with which the behaviour policy selected the stored action. Q-learning is therefore off-policy. Under sufficient coverage, no behaviour-action importance ratio is required in the tabular one-step update.

1.3 Algorithm

Algorithm 1: Online Tabular Q-Learning

Input: step size $\alpha \in (0, 1]$, exploration parameter $\varepsilon > 0$, discount factor $\gamma \in [0, 1)$, number of episodes N

Initialize $Q(s, a)$ arbitrarily for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$, with $Q(\text{terminal}, \cdot) = 0$

for $e = 1, \ldots, N$ **do**

  Initialize S_t at the episode start

  while S_t *is not terminal* **do**

    Choose $A_t \sim \mu(\cdot \mid S_t)$ from an exploratory behaviour policy, such as an ε-greedy policy with respect to Q

    Execute A_t and observe R_{t+1}, S_{t+1}

    $Y_t \leftarrow R_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q(S_{t+1}, a')$

    $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[Y_t - Q(S_t, A_t) \right]$

    $S_t \leftarrow S_{t+1}$

  end

end

The algorithm is tabular, online, model-free, off-policy, and asynchronous. It is a one-step temporal-difference control method: each observed transition $(S_t, A_t, R_{t+1}, S_{t+1})$ is used immediately for one stochastic-approximation update of the sampled entry $Q(S_t, A_t)$, while every other entry of the Q-table remains unchanged. It uses no function approximation, experience replay, minibatch optimization, or target network. The rest of this report introduces each of these in turn, and the properties just established are what make them necessary.

2 Q-Learning with Function Approximation

Tabular Q-learning cannot scale to large or continuous state spaces because it stores an independent value for every state-action pair. We therefore replace the Q-table by a parameterized state-action value function

$$\boxed{Q_t(s, a) \quad \longrightarrow \quad \hat{q}(s, a, w_t), \qquad w_t \in \mathbb{R}^d.} \tag{10}$$

The state space may now be continuous or high-dimensional. The action space is still assumed to be finite,

$$\mathcal{A} = \{a_1, \ldots, a_m\}, \qquad m = |\mathcal{A}| < \infty,$$

because every Q-learning target requires

$$\max_{a' \in \mathcal{A}} \hat{q}(S_{t+1}, a', w_t).$$

For continuous actions, this maximization becomes a separate optimization problem inside every Bellman backup.

2.1 Mean-Squared State-Action Value Error

Suppose temporarily that the optimal state-action value function Q^* is available. For a fixed behaviour policy μ, let d^μ denote its state-visitation distribution. The ideal mean-squared state-action value error is

$$\boxed{\overline{\text{QVE}}(w) = \mathbb{E}_{S \sim d^\mu} \left[\mathbb{E}_{A \sim \mu(\cdot \mid S)} \left[\left(Q^*(S, A) - \hat{q}(S, A, w) \right)^2 \right] \right].} \tag{11}$$

Here, d^μ weights states and $\mu(\cdot \mid S)$ weights actions at each state. Together,

$$d^\mu(s)\mu(a \mid s)$$

form the state-action visitation distribution induced by the behaviour policy.

The objective in (11) cannot be evaluated directly because Q^* is unknown.

2.2 Bellman Target and Semi-Gradient Update

The unavailable value $Q^*(S_t, A_t)$ is replaced by the one-step bootstrapped target

$$\boxed{Y_t = R_{t+1} + \gamma \max_{a' \in \mathcal{A}} \hat{q}(S_{t+1}, a', w_t).} \tag{12}$$

For the observed state-action pair, define the sample loss

$$L_t(w) = \frac{1}{2}\left[Y_t - \hat{q}(S_t, A_t, w)\right]^2. \tag{13}$$

Treating Y_t as constant during differentiation gives the semi-gradient

$$\nabla_w L_t(w)\big|_{w=w_t} = -\left[Y_t - \hat{q}(S_t, A_t, w_t)\right] \nabla_w \hat{q}(S_t, A_t, w_t).$$

Therefore,

$$\boxed{w_{t+1} = w_t + \alpha_t \left[Y_t - \hat{q}(S_t, A_t, w_t)\right] \nabla_w \hat{q}(S_t, A_t, w_t).} \tag{14}$$

Equations (12) and (14) define one-step semi-gradient Q-learning with function approximation.

2.3 Neural Q-Function Architectures

A neural state-action value function can be represented in two standard ways. The first network receives both the state and an action encoding and returns one scalar value. The second receives only the state and returns one value for every discrete action.

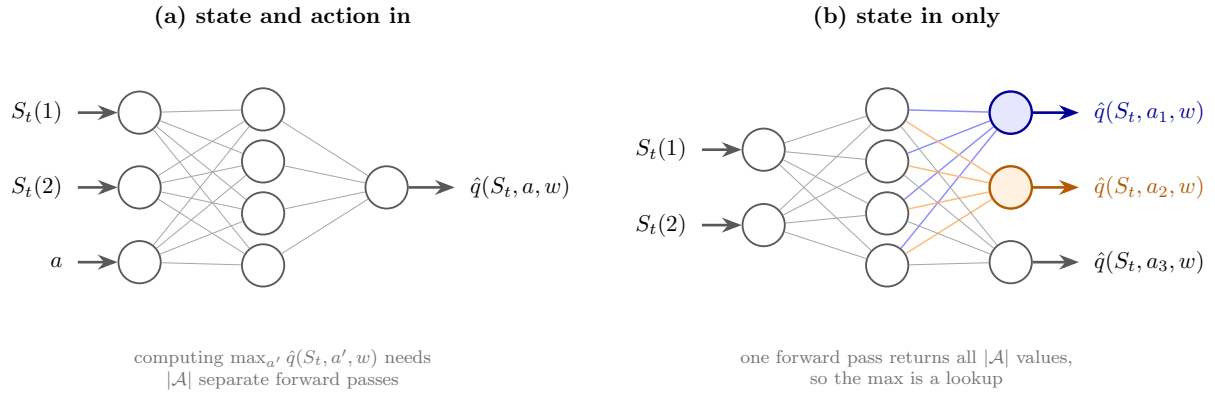


Figure 1: Two ways to represent $\hat{q}$ with a neural network. In (a) the action is part of the input and the network returns a single value, so each Bellman target costs $|\mathcal{A}|$ forward passes. In (b) the network takes only the state and devotes one output unit to each action, so a single pass yields every value and the maximization is a lookup over the output vector. DQN uses (b) for that reason; (a) is the form required once the action space is continuous and the output layer can no longer enumerate it.

Architecture (b) is what makes the neural version practical, but it does not change the mathematics: (12) and (14) are the same update either way. Putting them together with an exploratory behaviour policy gives the direct neural analogue of Algorithm 1, in which each transition is used once and then discarded.

2.4 Naive Online Q-Learning

The direct online procedure uses each transition once, the moment it is observed.

Initialize parameters w.

repeat

1. Sample an action from the behaviour policy:

$$A_t \sim \pi_{b,t}(\cdot \mid S_t), \qquad \pi_{b,t} = \varepsilon\text{-greedy}(\hat{q}(S_t, \cdot, w_t)) .$$

2. Sample one transition from the environment:

$$(S_{t+1}, R_{t+1}) \sim p(\cdot, \cdot \mid S_t, A_t).$$

3. Compute the bootstrapped target

$$Y_t = R_{t+1} + \gamma \max_{a' \in \mathcal{A}} \hat{q}(S_{t+1}, a', w_t).$$

4. Apply one semi-gradient update:

$$w_{t+1} = w_t + \alpha_t \left[Y_t - \hat{q}(S_t, A_t, w_t) \right] \nabla_w \hat{q}(S_t, A_t, w_t).$$

5. Set $S_t \leftarrow S_{t+1}$ and discard the transition.

repeat

Q-network $\hat{q}(\cdot, \cdot, w)$

supplies the behaviour policy, the target, and the prediction

Figure 2: Naive online Q-learning with nonlinear function approximation. Each environment transition drives exactly one semi-gradient update and is then discarded. A single network plays three roles at once: it selects actions, constructs the bootstrapped target, and supplies the prediction being fitted.

This is the direct neural analogue of tabular Q-learning. Unlike tabular Q-learning, however, it has no general convergence guarantee and may oscillate or diverge when nonlinear function approximation, bootstrapping, and off-policy learning interact.

3 Stabilizing Neural Q-Learning

In the tabular case the gradient of $\hat{q}(S_t, A_t, w)$ with respect to w is the indicator of a single entry, so an update touches nothing else. With a network it is dense, and two consequences follow at once.

The update at (S_t, A_t) can also change $\hat{q}(S_{t+1}, a', w)$ whenever their parameter gradients overlap, so the target in (12) moves in response to the very step taken to reach it. The regression has no fixed label. And the update changes $\hat{q}$ across the rest of the state-action space, so a correction made at one pair can undo a correction made at another.

Neither effect is an accident of implementation. Both are consequences of generalization, which is the reason function approximation was introduced. The result is that this procedure inherits no

convergence guarantee, an instability conventionally attributed to the *deadly triad*:

$$\boxed{\begin{array}{c}\text{function approximation} + \text{bootstrapping} + \text{off-policy learning} \\ \Longrightarrow \text{possible instability or divergence.}\end{array}} \tag{15}$$

Off-policy learning has two distinct roles here. In tabular Q-learning, off-policy sampling is compatible with convergence under sufficient state-action coverage and suitable step sizes. Under function approximation, however, the interaction among off-policy sampling, bootstrapping, and shared parameters can produce unstable projected updates. At the same time, Q-learning's one-step optimality target makes replay of transitions generated by earlier behaviour policies possible without behaviour-action importance ratios.

3.1 Experience Replay

Before addressing the moving target, consider what the data stream itself does to the update. Training on the most recent transition and then discarding it creates five distinct problems.

1. **Consecutive transitions are strongly correlated.** Writing $Z_t = (S_t, A_t, R_{t+1}, S_{t+1})$, the next transition begins from the state contained in the current one, so

   $$\mathbb{P}(Z_{t+1} \mid Z_t) \neq \mathbb{P}(Z_{t+1}).$$

   Nearby transitions produce similar TD errors and therefore correlated gradient directions, so a sequence of updates pushes repeatedly in one direction from a narrow region of the state space.

2. **The training distribution moves with the behaviour policy.** The transition distribution at time t is

   $$\nu_t(s, a, r, s') = d^{\mu_t}(s)\, \mu_t(a \mid s)\, p(r, s' \mid s, a),$$

   where μ_t is the behaviour policy and d^{μ_t} its state-visitation distribution. Since μ_t is derived from the network being trained, $\nu_{t+1} \neq \nu_t$ in general: the data stream is nonstationary.

3. **Updates concentrate on recently visited pairs.** A single online update touches only (S_t, A_t), so a long trajectory through one region trains the network on that region alone and supplies no corrective signal elsewhere.

4. **Shared parameters overwrite what was already learned.** The update at (S_t, A_t) changes the prediction at any other pair (s, a) by approximately

   $$\hat{q}(s, a, w_{t+1}) - \hat{q}(s, a, w_t) \approx \alpha_t \delta_t \, \nabla_w \hat{q}(s, a, w_t)^\top \nabla_w \hat{q}(S_t, A_t, w_t).$$

   Whenever the two gradients are not orthogonal, fitting the current transition perturbs estimates at pairs that were never visited, and may destroy accurate ones.

5. **Each transition is used once.** Rare rewards, failures, and terminal events supply a single gradient and are then thrown away, so their information is both expensive to obtain and easy to overwrite.

Problems 1, 2, 3, and 5 are all properties of the *order* in which data arrives rather than of the update rule, and all four are addressed by decoupling collection from training. Store transitions in a buffer

$$\boxed{\mathcal{D} = \{(S_i, A_i, R_{i+1}, S_{i+1})\}, \qquad |\mathcal{D}| \le C,} \tag{16}$$

with capacity C; when the buffer is full, inserting a new transition evicts the oldest. Each update then draws a uniform mini-batch

$$\boxed{\mathcal{B} = \{(S_i, A_i, R_{i+1}, S_{i+1})\}_{i=1}^{B} \sim \text{Uniform}(\mathcal{D})} \tag{17}$$

rather than the transition just observed, computes targets

$$Y_i = R_{i+1} + \gamma \max_{a' \in \mathcal{A}} \hat{q}(S_{i+1}, a', w_t), \tag{18}$$

and applies

$$\boxed{w_{t+1} = w_t + \frac{\alpha_t}{B} \sum_{i=1}^{B} [Y_i - \hat{q}(S_i, A_i, w_t)] \nabla_w \hat{q}(S_i, A_i, w_t).} \tag{19}$$

This is legitimate only because Q-learning is off-policy. The target (18) contains no reference to the policy that generated the transition, so stale samples from long-superseded behaviour policies require no correction.

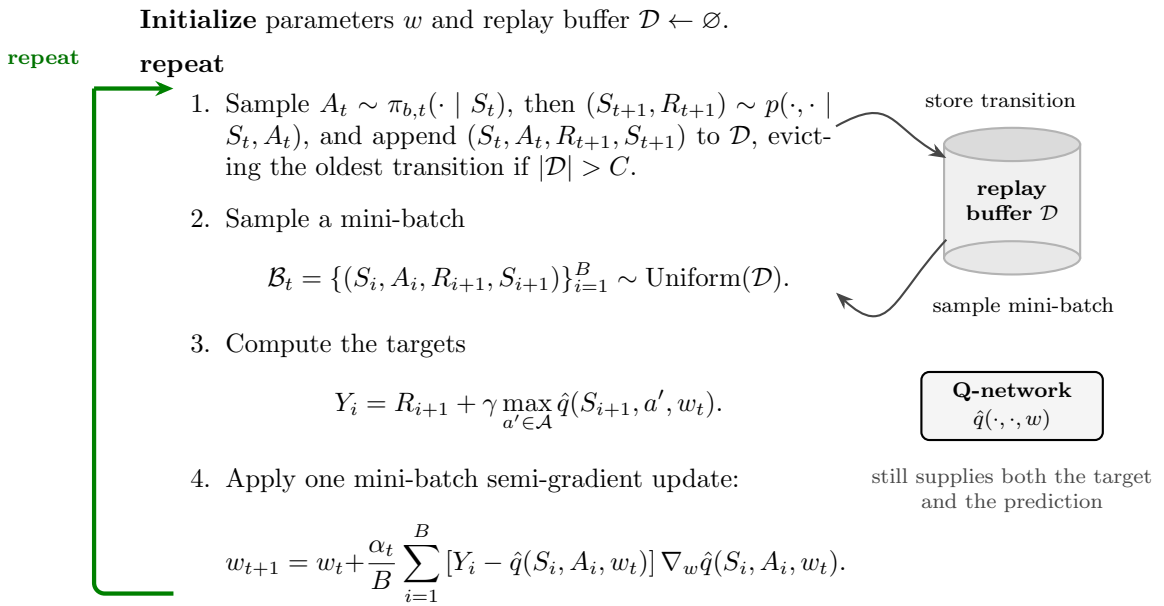


Figure 3: Neural Q-learning with experience replay. Each newly collected transition is appended to the replay buffer, and every update is computed from a mini-batch drawn uniformly at random from that buffer rather than from the transition just observed. A single network still supplies both the prediction in step 4 and the bootstrapped target in step 3, so the regression labels continue to move with every gradient step.

Replay decorrelates the updates, spreads them across the buffer rather than the current trajectory, reuses each transition many times, and separates the rate of data collection from the rate of optimization.

It leaves two things untouched. Problem 4 is not fixed here or anywhere later in this report: interference between state-action pairs is inherent to sharing parameters across states, and it is the price paid for the generalization that made function approximation worth using. The moving target is not fixed either, since the same network still supplies both sides of the regression. That one does have a remedy, and it is the subject of the next subsection.

3.2 Target Network

Experience replay fixed the order in which data arrives, but not what the data is regressed onto. The target is still built from the parameters being optimized,

$$Y_i = R_{i+1} + \gamma \max_{a' \in \mathcal{A}} \hat{q}(S_{i+1}, a', w_t),$$

so every gradient step on w_t also moves the labels that the next step will be fitted to. This is what distinguishes the update from ordinary regression: in supervised learning the labels are supplied from outside and held fixed, whereas here they are recomputed after each step from the function being learned. Nothing prevents the prediction and the target from chasing one another.

The remedy is to hold the label-generating function still. Introduce a second parameter vector $\bar{w}_t$, initialized as $\bar{w}_0 \leftarrow w_0$, and compute the targets from it:

$$\boxed{Y_i = R_{i+1} + \gamma \max_{a' \in \mathcal{A}} \hat{q}(S_{i+1}, a', \bar{w}_t).} \tag{20}$$

The online network is then fitted by minimizing

$$L_t(w) = \frac{1}{2B} \sum_{i=1}^{B} \left[Y_i - \hat{q}(S_i, A_i, w) \right]^2, \tag{21}$$

whose semi-gradient step is

$$\boxed{w_{t+1} = w_t + \frac{\alpha_t}{B} \sum_{i=1}^{B} \left[Y_i - \hat{q}(S_i, A_i, w_t) \right] \nabla_w \hat{q}(S_i, A_i, w_t).} \tag{22}$$

Only w_t is changed by (22). Since $\bar{w}_t$ is held fixed between synchronizations, the labels in (20) are genuine constants throughout that interval, and the update really is regression while it lasts.

Hard update. Classical DQN replaces the target parameters periodically:

$$\boxed{\bar{w}_{t+1} = \begin{cases} w_{t+1}, & t+1 \equiv 0 \pmod{K}, \\ \bar{w}_t, & \text{otherwise}, \end{cases}} \tag{23}$$

where K is the target-network update interval.

Soft update. A continuous alternative mixes a small fraction of the online parameters into the target at every step:

$$\boxed{\bar{w}_{t+1} = (1-\tau)\bar{w}_t + \tau w_{t+1}, \qquad 0 < \tau \ll 1.} \tag{24}$$

Setting $\tau \approx 1/K$ makes the two schemes comparable in how quickly the old target is forgotten. The hard update proceeds in bursts, so the lag between $\bar{w}$ and w is maximal just before a copy and zero just after; the soft update keeps that lag constant. Classical DQN uses (23), while (24) is standard in the off-policy actor-critic algorithms that follow.

Choosing the interval. Both schemes trade the same two quantities against each other. Refresh too often and the labels start moving again, recovering the instability the target network was introduced to remove. Refresh too rarely and learning stalls for a different reason: reward information propagates backwards through the state space roughly one step per refresh, since a state can only acquire a nonzero value once its successor has one *in the target network*. Between refreshes the online network fits labels that have not yet seen that information. In the worst case a horizon of H steps therefore needs H refreshes before value reaches the initial state. Generalization softens this considerably, since a network assigns similar values to similar states without waiting for a copy, but the tension is real and it is why K is a hyperparameter rather than something to be set as large as possible.

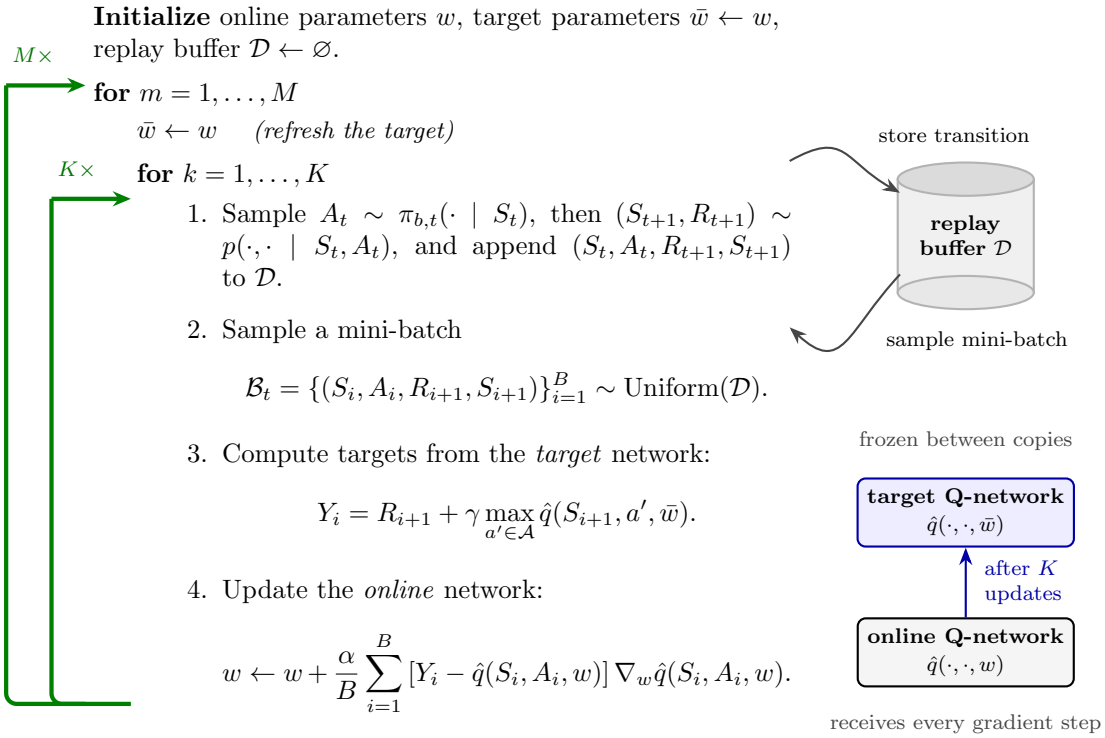


Figure 4: The DQN algorithm. Each inner iteration collects one transition, stores it in the replay buffer, samples a mini-batch, and takes a single gradient step on the online network. The targets are computed from the target network, whose parameters stay frozen for all K inner iterations; the outer loop copies the online parameters across and begins again.

This is DQN. Replay addressed the order in which data arrives; the target network addressed what that data is fitted to. Neither touches the max in (20), which is the source of a separate and systematic error taken up in the next report.

3.3 Generalized Q-Learning

DQN was presented as a single loop with a fixed schedule: one transition collected, one gradient step taken, the target refreshed every K updates. That schedule is a choice, not a requirement. Underneath it are three separate activities:

1. **Data collection.** Act in the environment and append transitions to the replay buffer, evicting the oldest when it is full.
2. **Target refresh.** Copy the online parameters into the target network.
3. **Q-function regression.** Draw mini-batches from the buffer and take gradient steps on the online parameters.

These three touch each other only in two places. Collection writes to $\mathcal{D}$ and regression reads from it, so the buffer is the only channel between the agent's interaction with the world and its learning. Refresh reads w and writes $\bar{w}$, so it reaches regression only through the labels. Neither channel requires the two ends to run at the same speed. Each process may therefore be run as often or as rarely as we like.

The symbol K is used in Figure 6 for the number of transitions collected between target refreshes, which coincides with the update interval of (23) exactly when $U = 1$.

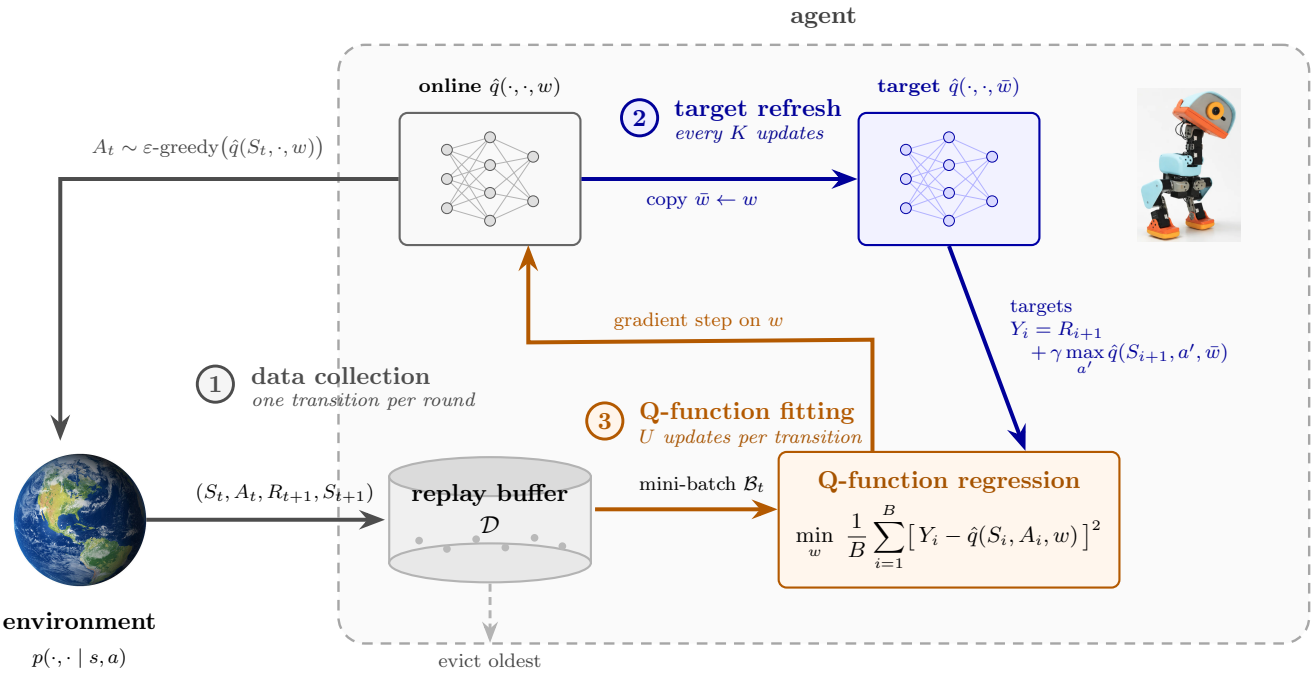


Figure 5: Generalized Q-learning as three processes. Process 1 acts with an ε-greedy policy derived from the online network and appends transitions to the replay buffer, evicting the oldest when it is full. Process 3 draws a mini-batch and takes one gradient step on w, regressing the online network onto targets built from the target network. Process 2 periodically copies w into $\bar{w}$; the two networks share an architecture and differ only in how recently their weights were set. Collection and learning meet only at the buffer, and the target enters learning only through the labels, so the three may run at any relative rates.

Three algorithms, one recipe. Three familiar methods are three settings of these rates. Online Q-learning evicts immediately, so $|\mathcal{D}| = 1$ and everything advances in lockstep. DQN keeps a buffer of about 10^6 transitions, couples collection and regression one-to-one, and refreshes the target every 10^4 updates. Fitted Q-iteration, the batch method that predates DQN, stops collecting altogether: it holds a fixed dataset, computes targets from a frozen network, and runs the regression to convergence before refreshing and repeating. The names differ; the recipe does not.

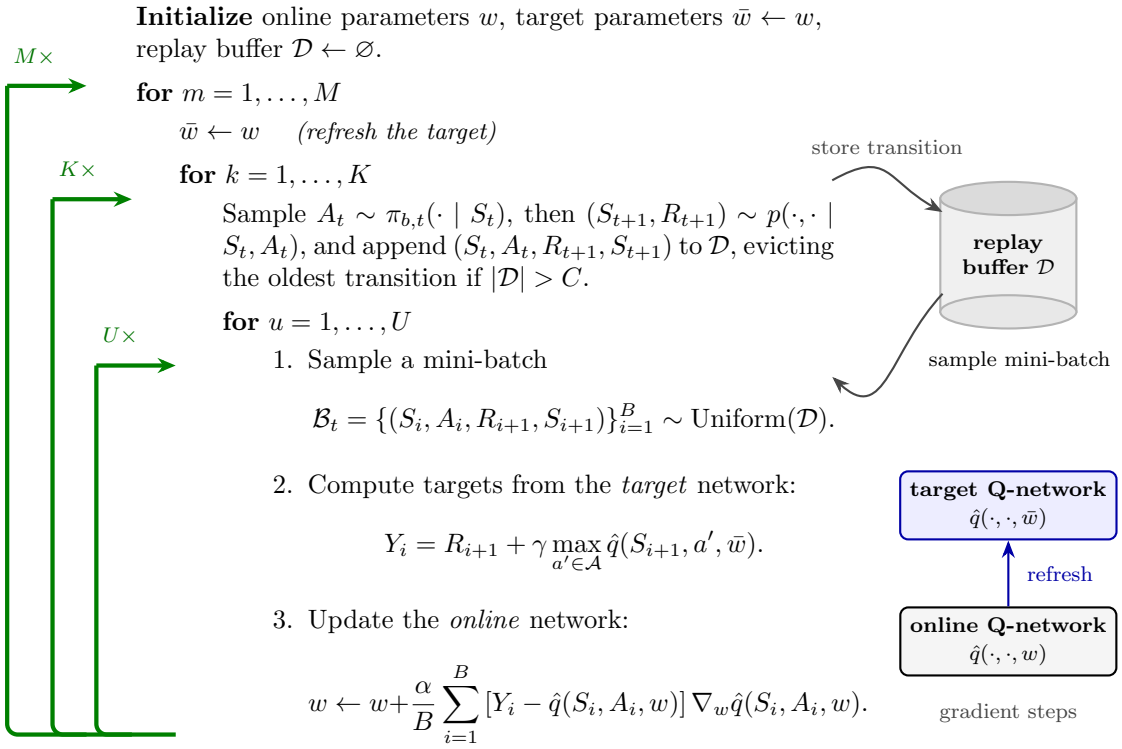
Update-to-data ratio. The rate that matters most in practice is

$$\boxed{\text{UTD} = \frac{\text{gradient updates}}{\text{transitions collected}} = U,} \tag{25}$$

the number of gradient steps taken per environment step. DQN sets $U = 1$, which is convention rather than necessity, and there is no reason the two should be coupled at all now that the buffer sits between them.

Raising U extracts more learning from each transition, so return improves faster per environment step at a compute cost proportional to U. That is a good trade whenever interaction is the scarce resource and computation is not, which on real hardware is the usual situation: a robot arm supplies transitions at the speed of physics while a GPU can take many gradient steps in the interval between them.

It cannot be raised without limit. Beyond some point each additional update fits the same buffer harder without adding information, and the network overfits it. The failure is worse than ordinary overfitting because of the max in the target. Fitting error is not symmetric once a maximum is taken over it, so repeated updates inflate the targets rather than merely scattering them, and the inflation is largest exactly where the network is least reliable. In practice U past roughly five or ten needs testing rather than assumption. That optimism has a cause worth deriving on its own, and it is the subject of the next section.



	$\lvert\mathcal{D}\rvert$	U (UTD ratio)	K
Online Q-learning	1	1	1
DQN	$\sim 10^6$	1	$\sim 10^4$
Fitted Q-iteration	fixed dataset	large	N/A

Figure 6: Generalized Q-learning as three processes running at independent rates. Process 1 collects transitions and writes them to the replay buffer; process 2 refreshes the target parameters; process 3 samples mini-batches and takes gradient steps on the online network. The buffer is the only channel between collection and learning, so the three rates need not be coupled. The ratio U of gradient updates to collected transitions is the *update-to-data* ratio: raising it speeds learning per environment step at proportionally higher compute cost, until the network begins overfitting the buffer and, because of the max in the target, does so optimistically.