Markov Decision Processes Lecture Notes 08

Q-Learning

Authors: Benjamin Van Roy *November 5, 2025*

1 Real-Time Value Iteration

Real-time value iteration is a version of asynchronous value iteration that selects states to update via simulating a trajectory of state-action pairs. Q-learning can be viewed as a variation that avoids computing expectations required to apply value iteration updates. We begin by reviewing asynchronous value iteration and then move on to real-time value iteration and Q-learning. We will restrict attention to discounted return, with a discount factor $\gamma \in [0, 1)$, though ideas we cover extend to total return and average and relative return. We will drop subscript γ, for example, from $Q_{*,\gamma}$, with an understanding that we are working exclusively in this lecture with discounted return.

1.1 Asynchronous Value Iteration

Asynchronous value iteration is expressed by Algorithm 1. In each iteration, the algorithm updates values at states within a specified set B_k, leaving others unchanged. As we established in Lecture 05, if each state $s \in \mathcal{S}$ appears in the sequence $B_1, B_2, B_3, \ldots$ infinitely often, then the sequence V_0, V_1, V_2 converges to V_*. This follows from the fact that the Bellman operator T is a contraction with respect to $\|\cdot\|_\infty$ and V_* is its uique fixed point.

Algorithm 1 asynchronous value iteration

given state update sets $B_0, B_1, B_2, \ldots$
given initial value function V_0
for $k = 0, 1, 2, 3, \ldots$ **do**
  for $s \in \mathcal{S}$ **do**
    if $s \in B_k$ **then**
      $V_{k+1}(s) \leftarrow (TV_k)(s)$
    else
      $V_{k+1}(s) \leftarrow V_k(s)$

1.2 The Real-Time Value Iteration Algorithm

While asynchronous value iteration in each iteration updates values across a set B_k of states, real-time value iteration updates the value at a single state S_k. The sequence f states is generate via simulating a trajectory. It is natural to think of this trajectory as evolving over time, so we index the states by t instead of k. Algorithm 2 goes through the steps. The algorithm begins with some initial value function V_0 and state S_0. At each time, the algorithm applies an action A_t that is greedy with respect to V_t, simulating a transition, which arrives at the next state S_{t+1}. Note that in our description, we use two alternative ways of expressing dynamics: P for transition probabilities and (f, ν) for state update function and disturbance distribution. It is possible to describe the algorithm with just one or the other, but using both simplifies our description.

Algorithm 2 real-time value iteration

given initial value function V_0
given initial state S_0
for $t = 0, 1, 2, 3, \ldots$ **do**
  $A_t \in \arg\max_{a \in \mathcal{A}} \left(r(S_t, a) + \gamma \sum_{s' \in \mathcal{S}} P_{ass'} V_t(s') \right)$
  sample $W_{t+1} \sim \nu(\cdot | S_t, A_t)$
  $S_{t+1} \leftarrow f(S_t, A_t, W_{t+1})$
  for $s \in \mathcal{S}$ **do**
    if $s = S_t$ **then**
      $V_{t+1}(s) \leftarrow (TV_t)(s)$
    else
      $V_{t+1}(s) \leftarrow V_t(s)$

We know that asynchronous value iteration converges on V_* if each state is updated infinitely often. Hence, if the trajectory $S_0, S_1, S_2, \ldots$ visits each state $s \in \mathcal{S}$ infinitely often then real-time value iteration converges on V_* as well. However, as we will discuss later, the algorithm converges on optimal actions under weaker conditions.

1.3 Action Values

Value iteration and it asynchronous variants can be extended to operate on action values. We review that here and devise a version of real-time value iteration that operates on action values. This version is more similar to Q-learning, which we are building up to, because that algorithm works exclusively with action values.

Recall that an action value function Q maps state-action pairs to real numbers. For each stationary policy π, there is an action value function Q_π defined by

$$Q_{\pi,\gamma}(s, a) = r(s, a) + \gamma \sum_{s' \in \mathcal{S}} P_{ass'} V_\pi(s') \qquad \forall s \in \mathcal{S}, a \in \mathcal{A}. \tag{1}$$

The action value $Q_\pi(s, a)$ represents the expected discounted return starting at state s if action a is applied and actions are selected by the policy π over each subsequent time step. The optimal action value function Q_* is defined by

$$Q_*(s, a) = r(s, a) + \gamma \sum_{s' \in \mathcal{S}} P_{ass'} V_*(s') \qquad \forall s \in \mathcal{S}. \tag{2}$$

We define a Bellman operators F_π and F for action values:

$$(F_\pi Q)(s, a) = r(s, a) + \gamma \sum_{s' \in \mathcal{S}} P_{ass'} \sum_{a' \in \mathcal{A}} \pi(a' | s') Q(s', a') \qquad \forall s \in \mathcal{S}, a \in \mathcal{A}. \tag{3}$$

$$(FQ)(s, a) = r(s, a) + \gamma \sum_{s' \in \mathcal{S}} P_{ass'} \max_{a' \in \mathcal{A}} Q(s', a') \qquad \forall s \in \mathcal{S}, a \in \mathcal{A}. \tag{4}$$

The functions Q_π and Q_* are fixed points:

$$F_\pi Q_\pi = Q_\pi \qquad \text{and} \qquad FQ_* = Q_*. \tag{5}$$

The Bellman operators F_π and F enjoys properties similar to T_π and T. For example, they are monotonic: for all Q and Q' such that $Q \leq Q'$, we have $FQ \leq FQ'$. And they are a contraction mappings: for all Q and Q', we have

$$\|F_\pi Q - F_\pi Q'\|_\infty \leq \gamma \|Q - Q'\|_\infty \qquad \text{and} \qquad \|FQ - FQ'\|_\infty \leq \gamma \|Q - Q'\|_\infty.$$

As such, value iteration converges: for all Q,

$$\lim_{k\to\infty} F_\pi^k Q = Q_\pi \qquad \text{and} \qquad \lim_{k\to\infty} F^k Q = Q_*.$$

Algorithm 3 is a variation of real-time value iteration that operates on action values.

Algorithm 3 real-time value iteration with action values

given initial value function Q_0
given initial state S_0
for $t = 0, 1, 2, 3, \ldots$ **do**
  $A_t \in \arg\max_{a\in\mathcal{A}} Q_t(S_t, a)$
  sample W_{t+1}
  $S_{t+1} \leftarrow f(S_t, A_t, W_{t+1})$
  for $s \in \mathcal{S}$ **do**
    if $(s, a) = (S_t, A_t)$ **then**
      $Q_{t+1}(s, a) \leftarrow (FQ_t)(s, a)$
    else
      $Q_{t+1}(s, a) \leftarrow Q_t(s, a)$

1.4 Optimism

It follows from the contraction property that, if every state-action pair is updated infinitely often, Algorithm 3 converges on Q_*. However, there is no guarantee that the simulated trajectory will visit each state-action pair infinitely often.

One might hope that the trajectory will at least settle on the behavior of an optimal policy. That is, as time progresses, the trajectory only selects optimal actions and remains in states that would be visited by an optimal policy. But even that is not the case. In fact, it is possible that optimal actions are never even tried.

To see why optimal actions may never be tried, consider the simple example illustrated in Figure 1. There is only one state. One action generates reward of 1 and the other generates 0. The initial value estimate for the second action is correct: $Q(s, a) = 0 = Q_*(s, a)$. The value estimate for the first action is wrong: $Q_0(s, a) = -1 \neq 1/(1-\gamma) = Q_*(s, a)$. Real time real-time value iteration would only ever execute the second action. Each update would leave the initial value estimates unchanged.

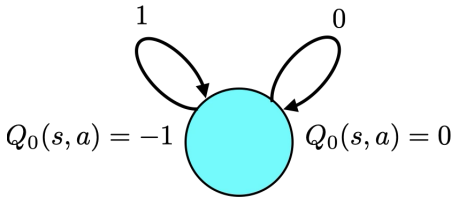
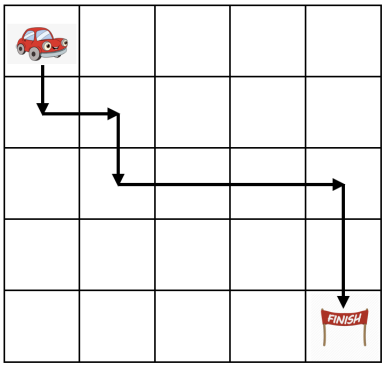


Figure 1: An optimal action may never be tried if the initial value estimate is pessimistic.

This example illustrates an important observation: a *pessimistic* initial value estimate can discourage trying an action. If that action is optimal, not trying it is problematic. This motivates a requirement: that the initial value estimates be *optimistic*. In particular, consider requiring that $Q_0 \geq Q_*$. For reasons we will later explain, under this condition, the trajectory generated by real-time value iteration will eventually select only optimal actions.

1.5 Example: Slippery Path

To further digest the workings of real-time value iteration, let us consider its application to a simple 'slippery path' MDP, which models a grid world illustrated in Figure 2 (left). A car starts in the upper left corner and tries to get to the lower right as quickly as possible. After reaching the lower right corner, the car restarts in the upper left. In each timestep, the car chooses between two actions `right` and `down`. In either case, the car moves in the desired direction with probability $1 - p_{\text{slip}}(s)$. Note that $p_{\text{slip}}(s)$ varies with the state s, which identifies the current location of the car within the grid. Figure 2 (right) provides slip probabilities at each state. If the car fails to move in the selected direction, it remains in the same state.



0.2	0.5	0.5	0.5	0.5
0.2	0.2	0.5	0.5	0.5
0.5	0.2	0.2	0.2	0.2
0.5	0.5	0.5	0.5	0.2
0.5	0.5	0.5	0.5	

Figure 2: The slppery path environment (left) and slip probabilities (right).

A reward of 1 is received when at the lower right cell, where the finish line is. Otherwise, a reward of 0 is received over each time step. We use a discount factor $\gamma = 0.99$. We initialize the value function to $Q_0(s, a) = 10$ for all state action pairs. Figure 3 plots the cumulative reward (left) and an exponential moving average of rewards (right) over time steps. The rate at which rewards accrue increases quickly and then tapers off.

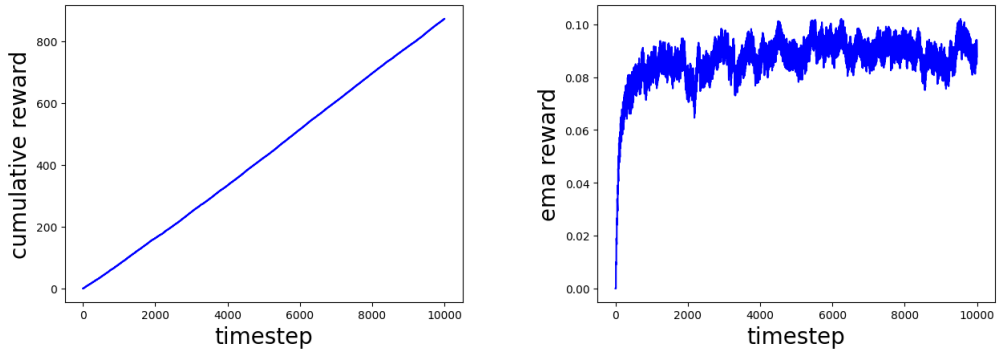


Figure 3: Performance of real-time value iteration in the slippery path environment.

1.6 Why does this work?

Some analysis that offers insight into why RLSVI with optimistic initialization "works." One important observation is that if we initialize with optimistic values then values remain optimistic. The following result formalizes the claim.

Lemma 1. (RTVI optimism) *Fix an MDP $(\mathcal{S}, \mathcal{A}, P)$, a reward function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, and a discount factor $\gamma \in [0, 1)$. If Algorithm 3 is invoked with optimistic initial values $Q_0 \geq Q$ then, for all t, $Q_t \geq Q_*$.*

Proof. It suffices to show that the update $Q_{t+1}(S_t, A_t) \leftarrow (FQ_t)(S_t, A_t)$ retains optimism. This follows from monotonicity: if $Q_t \geq Q_*$ then $FQ_t \geq FQ_* = Q_*$. □

Let

$$(F_{\overline{s},\overline{a}}Q)(s,a) = \begin{cases} (FQ)(s,a) & \text{if } (s,a) = (\overline{s},\overline{a}) \\ Q(s,a) & \text{if } (s,a) \neq (\overline{s},\overline{a}). \end{cases}$$

Each real-time value iteration update can be written as $Q_{t+1} = F_{S_t,A_t}Q_t$. It is easy to show that, for each s and a, the operator $F_{s,a}$ is monotonic. We state this as a lemma, though we omit the proof, which is almost identical to the proof that F is monotonic.

Lemma 2. (RTVI monotonicity) *Fix an MDP $(\mathcal{S}, \mathcal{A}, P)$, a reward function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, and a discount factor $\gamma \in [0, 1)$. If $Q \leq Q'$ then, for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$, $F_{s,a}Q \leq F_{s,a}Q'$.*

The following result establishes that real-time value iteration eventually generates optimal actions. It imposes an assumption that is stronger than optimism.

Theorem 3. (RTVI monotonic convergence) *Fix an MDP $(\mathcal{S}, \mathcal{A}, P)$, a reward function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, and a discount factor $\gamma \in [0, 1)$. If Algorithm 3 is invoked with initial values Q_0 such that $Q_0 \geq FQ_0$ then there exists some time τ such that $A_t \in \arg\max_{a\in\mathcal{A}} Q_*(S_t, a)$ for all $t \geq \tau$.*

Proof. Suppose $Q_t \geq FQ_t$. This implies that $Q_t \geq F_{s,a}Q_t$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$. We have

$$Q_{t+1} = F_{S_t,A_t}Q_t \geq FQ_t \geq FF_{S_t,A_t}Q_t = FQ_{t+1},$$

where the first inequality follows from the fact that $Q_t \geq FQ_t$, and the second inequality follows from monotonicity of F and the fact that $Q_t \geq F_{S_t,A_t}Q_t$. It follows from induction that $Q_t \geq FQ_t$ for all t.

Hence, we have a nonincreasing sequence of functions bounded below by Q_*. This sequence must converge. Let $\overline{Q}$ be the limit of convergence and let $\mathcal{X} \subseteq \mathcal{S} \times \mathcal{A}$ be the set of state-action pairs that are visited infinitely often. It follows that there is a time τ such that $A_t \in \arg\max_{a\in\mathcal{A}} \overline{Q}(S_t, a)$ and $(S_t, A_t) \in \mathcal{X}$ for all $t \geq \tau$. Let π denote the corresponding greedy policy. It follows that, for $t \geq \tau$, $\overline{Q}(S_t, A_t) = (F\overline{Q})(S_t, A_t) = (F_\pi\overline{Q})(S_t, A_t)$ and $P_{A_tS_ts} = 0$ for $s \notin \mathcal{X}$. Hence, $(F_\pi\overline{Q})(S_t, A_t)$ does not depend on values at state-action pairs outside of $\mathcal{X}$, and therefore, $\overline{Q}(S_t, A_t) = (F_\pi\overline{Q})(S_t, A_t)$ implies $\overline{Q}(S_t, A_t) = Q_\pi(S_t, A_t)$. In other words, $\overline{Q}(s, a) = Q_\pi(s, a)$ for all $(s, a) \in \mathcal{X}$. Since $Q_*(s, a) \leq \overline{Q}(s, a)$ and $Q_\pi(s, a) \leq Q_*(s, a)$, we have $Q_\pi(s, a) = Q_*(s, a)$ for $(s, a) \in \mathcal{X}$. Hence, π takes optimal actions. □

The proof under the weaker condition that $Q_0 \geq Q_*$ rather than $Q_0 \geq FQ_0$ is more complicated. The argument can be extracted from the analysis of Dong et al. [2022]. I could also write down a simpler, albeit still somewhat complicated, analysis if there is sufficient interest.

2 Q-learning

Each update applied by real time value iteration takes the form

$$Q_{t+1}(S_t, A_t) \leftarrow r(S_t, A_t) + \gamma \sum_{s\in\mathcal{S}} P_{A_tS_ts} \max_{a\in\mathcal{A}} Q(s, a). \tag{6}$$

Computing the right hand side requires summing over possible next states. When the state space is very large, this becomes computationally onerous. Q-learning overcomes this issue by using $Q_t(S_{t+1}, a)$ as an approximation of the expectation. Consider an update of the form

$$Q_{t+1}(S_t, A_t) \leftarrow r(S_t, A_t) + \gamma \max_{a\in\mathcal{A}} Q_t(S_{t+1}, a). \tag{7}$$

If transitions are deterministic then this update is equivalent to real-time value iteration. However, if transitions are stochastic, these updates could generate values that chatter erratically and endlessly. This is because the single sample $Q_t(S_t, a)$ is typically a very noisy approximation of its expectation. As we will see, Q-learning addresses this by using a step size to smooth across updates.

2.1 The Q-Learning Algorithm

Q-learning uses a "smoothed" update formula, which can be written as

$$Q_{t+1}(S_t, A_t) \leftarrow (1 - \alpha_t) Q_t(S_t, A_t) + \alpha_t \left(r(S_t, A_t) + \gamma \max_{a \in \mathcal{A}} Q_t(S_{t+1}, a) \right). \tag{8}$$

The scalar α_t is a *step size*, which reduces the magnitude of the change in value relative to (7). Indeed, the new value $Q_{t+1}(S_t, A_t)$ is a convex combination between the previous value $Q_{t+1}(S_t, A_t)$ and the target $r(S_t, A_t) + \gamma \max_{a \in \mathcal{A}} Q_t(S_{t+1}, a)$.

The action A_{t+1} is greedy with respect to Q_{t+1}. If $S_{t+1} \neq S_t$ then $Q_{t+1}(S_{t+1}, \cdot) = Q_t(S_{t+1}, \cdot)$ and, therefore, $Q_t(S_{t+1}, A_{t+1}) = \max_{a \in \mathcal{A}} Q_t(S_{t+1}, a)$. And if α_t is small, which is typically the case, even if $S_{t+1} = S_t$, $Q_{t+1}(S_{t+1}, \cdot) \approx Q_t(S_{t+1}, \cdot)$ and, therefore, $Q_t(S_{t+1}, A_{t+1}) \approx \max_{a \in \mathcal{A}} Q_t(S_{t+1}, a)$. As such, it feels natural to use a slightly modified update:

$$Q_{t+1}(S_t, A_t) \leftarrow (1 - \alpha_t) Q_t(S_t, A_t) + \alpha_t \left(r(S_t, A_t) + \gamma Q_t(S_{t+1}, A_{t+1}) \right), \tag{9}$$

or, equivalently,

$$Q_{t+1}(S_t, A_t) \leftarrow Q_t(S_t, A_t) + \alpha_t \left(r(S_t, A_t) + \gamma Q_t(S_{t+1}, A_{t+1}) - Q_t(S_t, A_t) \right). \tag{10}$$

The quantity $r(S_t, A_t) + \gamma \max_{a \in \mathcal{A}} Q_t(S_{t+1}, a') - Q_t(S_t, A_t)$ is called a *temporal difference*. It represents the difference between two predictions: $Q_t(S_t, A_t)$ is a prediction of future return starting at time t, while $r(S_t, A_t) + \gamma Q_t(S_{t+1}, A_{t+1})$ is a more informed prediction of the same future return but based on knowledge of the transition to state S_{t+1}. This update formula can be interpreted as nudging the value up if the more informed prediction is larger and down otherwise.

This update formula leads to Algorithm 4. This version of Q-learning resembles real-time value iteration except that it avoids taking the expectation $\sum_{s \in \mathcal{S}} P_{A_t S_t s} \max_{a \in \mathcal{A}} Q_t(s, a)$, using instead the single-sample estimate $Q_t(S_{t+1}, A_{t+1})$.

Algorithm 4 Q-learning

given initial value function Q_0
given initial state S_0
for $t = 0, 1, 2, 3, \ldots$ **do**
  $A_t \in \arg\max_{a \in \mathcal{A}} Q_t(S_t, a)$
  sample W_{t+1}
  $S_{t+1} \leftarrow f(S_t, A_t, W_{t+1})$
  $Q_{t+1}(S_t, A_t) \leftarrow Q_t(S_t, A_t) + \alpha_t(r(S_t, A_t) + \gamma Q_t(S_{t+1}, A_{t+1}) - Q_t(S_t, A_t))$

Some insight into Q-learning is offered by results pertaining to a more general algorithm – asynchronous stochastic value iteration – which is presented as Algorithm 5. This algorithm uses the same sort of update rule as Q-learning but does not necessarily update states along a single simulated trajectory. Instead, like asynchronous value iteration, any sequence of updates is allowed. Results in Jaakkola et al. [1993], Tsitsiklis [1994] provide sufficient conditions for convergence of Q_k to Q_*. Based on those results, it suffice for each state-action pair to be updated infinitely often and for the step size sequence to satisfy $\sum_k \alpha_k = \infty$ and $\sum_k \alpha_k^2 < \infty$.

Algorithm 5 asynchronous stochastic value iteration

given state-action update sets $B_0, B_1, B_2, \ldots$
given initial action value function Q_0
for $k = 0, 1, 2, 3, \ldots$ **do**
  for $(s, a) \in \mathcal{S} \times \mathcal{A}$ **do**
    if $(s, a) \in B_k$ **then**
      sample next state $s' = f(s, a, W)$, where $W \sim \nu(\cdot|s, a)$
      $Q_{k+1}(s, a) \leftarrow Q_k(s, a) + \alpha_k(r(s, a) + \gamma \max_{a \in \mathcal{A}} Q_k(s', a') - Q_k(s, a))$
    else
      $Q_{k+1}(s, a) \leftarrow Q_k(s, a)$

2.2 Example: Slippery Path

Figure 4 plots results from applying Q-learning, with a fixed step size of $\alpha_t = 0.1$, to the slippery path environment. As with real-time value iteration, the performance improves in early training. But the pace of improvement is about an order of magnitude slower with Q-learning. The highest level of performance is similar to that reached by real-time value iteration. However, the exponential moving average generated by Q-learning is more volatile.

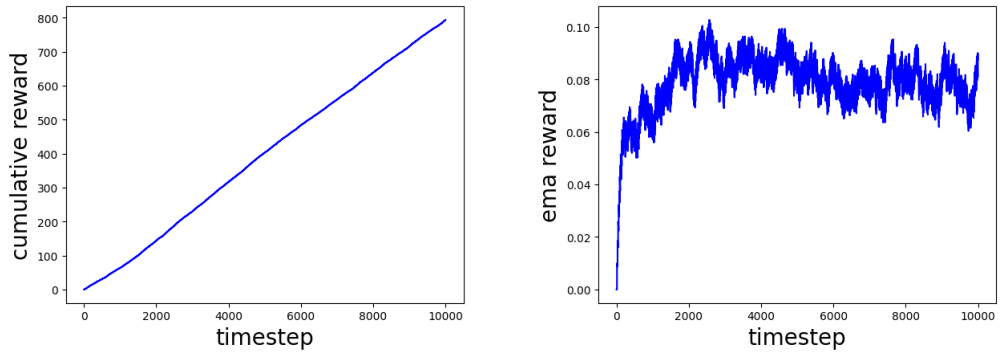


Figure 4: Performance of Q-learning in the slippery path environment.

2.3 Smoothing with a Step Size

Q-learning uses a step size to smooth noisy updates. To understand why this works, let us think about a simpler contexts that suffice to capture the key insights. To start with, consider an iid scalar sequence $X_1, X_2, \ldots$ with finite mean $X_* = \mathbb{E}[X_k]$ and finite variance $\mathbb{E}[X_k^2]$. A common law of large numbers ensure that the sample average

$$\overline{X}_T = \frac{1}{T} \sum_{t=1}^{T} X_t$$

converges on X_*.

One can alternatively express the sample average via a iteration

$$\overline{X}_{t+1} = \overline{X}_t + \frac{1}{t+1}(X_{t+1} - \overline{X}_t).$$

This iteration can be interpreted as nudging $\overline{X}_t$ toward X_{t+1} with a step size of $\alpha_t = 1/(t+1)$. A more general update rule

$$\overline{X}_{t+1} = \overline{X}_t + \alpha_t(X_{t+1} - \overline{X}_t)$$

offers flexibility in the choice of step size. A more general law of large numbers ensures convergence of this sequence to X_* if $\sum_t \alpha_t = \infty$ and $\sum_t \alpha_t^2 < \infty$. Note that these conditions are satisfied by $\alpha_t = 1/(t+1)$. But they allow for many alternatives. There are even laws of large numbers that ensure convergence on X_* for step sizes that are stochastic, with α_t possibly depending on $X_1, \ldots, X_t$.

In this iterative update, it is natural to think of X_* as a target, since we nudge X_t toward a noise estimate of it. This target is fixed. In Q-learning, on the other hand, the target is FQ_t since we move Q_t toward a noisy estimate of that. F is a contraction with fixed point Q_*. So we can think of FQ_t as an intermediate target that ultimately gets us to Q_*. This raises the question of whether taking a target to be the value supplied by a contraction mapping retains convergence guarantees. As we will discuss, more general laws of large numbers address this.

The sample mean iteration can be rewritten as

$$\overline{X}_{t+1} = \overline{X}_t + \alpha_t(X_* + W_{t+1} - \overline{X}_t),$$

where $W_{t+1} = X_{t+1} - X_*$ is zero-mean noise. Consider a more general form of update

$$\overline{X}_{t+1} = \overline{X}_t + \alpha_t(f(X_t) + W_{t+1} - \overline{X}_t),$$

where f is a contraction mapping with fixed point X_* and contraction factor $\gamma \in [0, 1)$. Since the argument is scalar, this means that, for all $x, x' \in \mathbb{R}$, $|f(x) - f(x')| \leq \gamma|x - x'|$. There are laws of large numbers that guarantee convergence on X_* so long as $\sum_t \alpha_t = \infty$ and $\sum_t \alpha_t^2 < \infty$. The convergence results of Jaakkola et al. [1993], Tsitsiklis [1994] represent generalizations of this sort of law of large numbers.

In practice, Q-learning is often executed with step size that do not vanish and thus $\sum_t \alpha_t^2 = \infty$. A special case of this is a fixed step size, as we used in the slippery path simulation of the experiment. With a fixed step size, we generally do not expect convergence. However, as in the case of the slippery path experiment, performance often still improves and if successful, the algorithm chatters near optimal performance. There is theory suggesting that, when this happens, the mean squared error of value estimates is of order $O(\alpha_t^2)$. Smaller step sizes reduce the pace of learning but in the long term reduce mean squared error.

2.4 Optimism

The convergence analyses of Jaakkola et al. [1993], Tsitsiklis [1994] require that each state-action pair is updated infinitely often. Our version of Q-learning 4 updates state-action pairs along a simulated trajectory. This does not generally include each state-action pair infinitely often. A different kind of analysis is required to ensure some sort of convergence.

Q-learning can be thought of as a noisy version of real-time value iteration. The latter can fail to converge if the initial value function is not optimistic. Hence, an analysis of Q-learning ought to also rely on optimism.

If we initialize real-time value iteration with an optimistic value function $Q_0 \geq Q_*$ then value functions Q_t remain optimistic for all t. This is not the case, however, with Q-learning. The reason is that updates are noisy and can produce pessimistic values by chance. To guarantee convergence, we need to maintain optimism. This can be done by adding an appropriately sized optimistic boost to each update, as shown in Algorithm 6. Dong et al. [2022] establish results that guarantee convergence on optimal behavior under suitable assumptions on the initial value function, optimistic boosts, and the step sizes.

Algorithm 6 Q-learning

given initial value function Q_0
given initial state S_0
for $t = 0, 1, 2, 3, \ldots$ **do**
  $A_t \in \arg\max_{a \in \mathcal{A}} Q_t(S_t, a)$
  sample W_{t+1}
  $S_{t+1} \leftarrow f(S_t, A_t, W_{t+1})$
  $Q_{t+1}(S_t, A_t) \leftarrow Q_t(S_t, A_t) + \alpha_t(r(S_t, A_t) + \gamma Q_t(S_{t+1}, A_{t+1}) + \Delta_t - Q_t(S_t, A_t))$

2.5 ϵ-Greedy Exploration

As we have discussed, optimism can induce convergent behavior while focusing computation on updating of state-action pairs that matter. An alternative approach that induces convergence involves *dithering*. This involves randomly perturbing actions so that each is occasionally tried. If the MDP is communicating (it is possible to get from any state to any other state) and step sizes do not vanish too quickly, dithering can ensure that each state-action pair is update infinitely often.

While this can be quite inefficient relative to approaches that more judiciously apply computation, dithering is used widely due to its simplicity and robustness. Algorithm 7 presents Q-learning with the most common form of dithering, which is called *ϵ-greedy exploration*. While each version of Q-learning that we discussed earlier always takes a greedy action, ϵ-greedy exploration takes a greedy action with probability $1 - \epsilon$. With probability ϵ, an action is selected uniformly at random from $\mathcal{A}$.

Algorithm 7 Q-learning with a ϵ-greedy exploration

given initial value function Q_0
given initial state S_0
for $t = 0, 1, 2, 3, \ldots$ **do**

$$A_t \sim \begin{cases} \text{unif}(\mathcal{A}) & \text{with probability } \epsilon \\ \arg\max_{a \in \mathcal{A}} Q_t(S_t, a) & \text{otherwise} \end{cases}$$

sample W_{t+1}
$S_{t+1} \leftarrow f(S_t, A_t, W_{t+1})$
$Q_{t+1}(S_t, A_t) \leftarrow Q_t(S_t, A_t) + \alpha_t(r(S_t, A_t) + \gamma Q_t(S_{t+1}, A_{t+1}) - Q_t(S_t, A_t))$

2.6 Parameterized Value Functions

We have discussed in previous lectures approximate version of policy iteration and linear programming that work with parameterized value functions. Likewise, Q-learning is often used with parameterized value functions. Consider an parameterized value function $Q_\theta : \mathcal{SA} \to \mathbb{R}$. Each parameter vector $\theta \in \mathbb{R}^K$ identifies an action value function.

Algorithm 8 presents a version of Q-learning that updates parameters θ and selects actions via ϵ-greedy exploration. The temporal difference provides a reinforcement signal used to nudge θ. The gradient $\nabla_\theta Q_\theta(S_t, A_t)$ is the direction that θ should be nudged to maximally increase the action value. This gradient serves to direct the change in response to the reinforcement signal and to scale its intensity.

Algorithm 8 Q-learning with a parameterized approximator

given initial parameters θ
given initial state S_0
for $t = 0, 1, 2, 3, \ldots$ **do**

$$A_t \sim \begin{cases} \text{unif}(\mathcal{A}) & \text{with probability } \epsilon \\ \arg\max_{a \in \mathcal{A}} Q_\theta(S_t, a) & \text{otherwise} \end{cases}$$

sample W_{t+1}
$S_{t+1} \leftarrow f(S_t, A_t, W_{t+1})$
$\theta \leftarrow \theta + \alpha_t \nabla_\theta Q_\theta(S_t, A_t)(r(S_t, A_t) + \gamma Q_\theta(S_{t+1}, A_{t+1}) - Q_\theta(S_t, A_t))$

A *tabular representation* Q_θ is one where each parameter expresses the value of one state-action pair. If we index components of θ by state-action pairs, we can write $Q_\theta(s, a) = \theta_{s,a}$. Such a representation should be equivalent to one where we encode the Q-function exactly. It is interesting to note that, with a tabular representation, the update used for a parameterized Q-function reduces to the one used by the exact

Q-learning update. In particular, for a tabular representation

$$(\nabla_\theta Q_\theta(s,a))_{s',a'} = \begin{cases} 1 & \text{if } (s,a) = (s',a') \\ 0 & \text{otherwise.} \end{cases}$$

Hence,

$$Q_\theta(S_t, A_t) \leftarrow Q_\theta(S_t, A_t) + \alpha_t(r(S_t, A_t) + \gamma Q_\theta(S_{t+1}, A_{t+1}) - Q_\theta(S_t, A_t)),$$

while values for other state-action pairs remain unchanged. This is equivalent to the update used, for example, in Algorithm 7.

3 DQN for Atari

The application of Q-learning to Atari video games more than a decade ago garnered a lot of interest [Mnih et al., 2013, 2015]. Indeed, it ignited credence in the coming of artificial general intelligence, which until then was viewed as a distant possibility. In this section, we describe the application domain, parameterized value function, algorithm used for computing parameters, and results.

3.1 Atari

Mnih et al. [2013, 2015] experimented with 49 games from the Atari arcade learning environment [Bellemare et al., 2013]. These games were played via the Atari 2600 video game console, which was developed in 1977. Figure 5 depicts the console and screenshots of representative games.

Figure 5: The Atari video game console and screenshots of representative games.

The challenge was to develop an algorithm that, using taking screenshots as input and applying actions of a joystick interface, could learn to play any of these games from scratch to attain human-level performance. The results of Mnih et al. [2015] demonstrated this possibility using a version of Q-learning.

3.2 MDP

Images rendered by the Atari console are 210x160 pixel with 128-color palatte. In order to reduce this complexity, the system preprocessed each image by cropping, downscaling, and extracting the Y-channel. This resulted in an 84x84 lossy encoding for each image. Images were generated at a rate of 60Hz.

To formulate an MDP $(\mathcal{S}, \mathcal{A}, f, \nu)$ for each game, the timestep was taken to be the duration of four frames or, equivalently, 1/15th of a second. Each state S_t was taken to be the most recent four 84x84 images. Each action A_t identifies one of 9 joystick positions and whether or not the button is pressed, as illustrated in Figure 6. Hence, $|\mathcal{A}| = 18$. In simulating game dynamics, the joystick configuration is assumed to be fixed over the four consecutive frames. Each game's MDP has a terminal state that indicates end of the game.

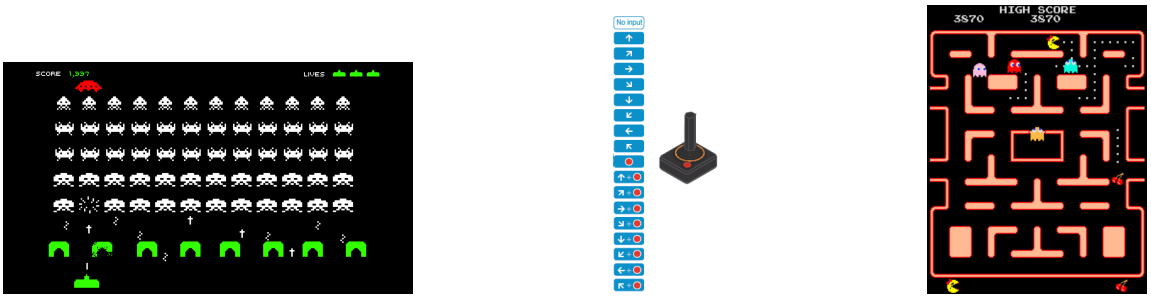


Figure 6: 18 actions afforded by the Atari joystick.

3.3 Reward

The reward signal was taken to be 1 if the score increases over the next time step, -1 if it decreases, and 0 otherwise. Hence, the reward function $r(S_t, A_t)$ in our notation would be the expected value of this reward over the next time step condtioned on the state.

A reason for using this discrete reward signal rather than actual game score increases was that the magnitudes of scores vary dramatically across games. This made it difficult to design a single algorithm that could learn to play all games using score increases as reward signals.

3.4 Value Function Architecture

The value function is represented by a convolutional neural network, as illustrated in Figure 7. The neural network takes the state S_t as input and generates 18 outputs, one per action, each expressing an action value $Q_\theta(S_t, a)$ for one of the 18 actions $a \in \mathcal{A}$. Millions of parameters are stored in θ.

3.5 Algorithm

Parameters are updated over episodes, each spanning the start to the end of a game. A variation of Q-learning, called *DQN* and presented as Algorithm 9, is used. DQN incorporates some new features motivated by observations and computational considerations that surfaced during the course of empirical research. These new features include:

- a replay buffer: the one million most recent transitions of the form $(S_t, A_t, R_{t+1}, S_{t+1})$ are buffered in $\mathcal{D}$,
- target value function: parameters $\hat{\theta}$ of a target value functions are only occasionally updated,
- minibatches: instead of updating at one state-action pair at a time, update for a minibatch $\mathcal{B}$ of size 32, sampled uniformly from $\mathcal{D}$,

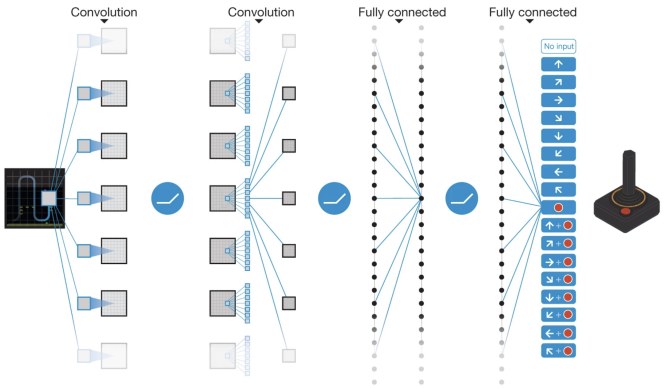


Figure 7: The convolutional neural network used to represent an action value function.

- RMSprop: this adaptive step size gradient descent algorithm was used,
- annealing: ϵ was annealed linarly from 1.0 to 0.1 over one milion games and then remained at 0.1 thereafter.

The discount factor was set at $\gamma = 0.99$.

Algorithm 9 Deep Q-learning with Experience Replay

given initial state S_0
initialize replay buffer $\mathcal{D}$ with capacity N
initialize with random weights θ
$\hat{\theta} \leftarrow \theta$ ▷ target
for $\ell = 0, 1, 2, 3, \ldots$ **do** ▷ episodes
  $t \leftarrow 0$
  while $S_t \neq$ `terminal` **do**
    execute $A_t \sim \begin{cases} \text{unif}(\mathcal{A}) & \text{with probability } \epsilon \\ \arg\max_{a \in \mathcal{A}} Q_t(S_t, a) & \text{otherwise} \end{cases}$
    observe R_{t+1} and S_{t+1}
    insert $(S_t, A_t, R_{t+1}, S_{t+1})$ in $\mathcal{D}$
    sample a minibatch $\mathcal{B} \subseteq \mathcal{D}$
    perform a gradient step to reduce $\sum_{(s,a,r,s') \in \mathcal{B}} (y - Q_\theta(s,a))^2$
      where
      $y \leftarrow \begin{cases} r(s,a) + \gamma \max_{a \in \mathcal{A}} Q_{\hat{\theta}}(s', a') & \text{if } s' \neq \texttt{terminal} \\ r(s,a) & \text{otherwise} \end{cases}$
    $t \leftarrow t + 1$
    every C times steps, $\hat{\theta} \leftarrow \theta$ ▷ update target

Note that if the gradient step were to simply used a fixed step size across time and parameters, the update for $s \neq$ `terminal` would take the form

$$\theta \leftarrow \theta + \alpha \sum_{(s,a,r,s') \in \mathcal{B}} \nabla_\theta Q_\theta(s,a) \left(r(s,a) + \gamma \max_{a \in \mathcal{A}} Q_{\hat{\theta}}(s', a') - Q_t(s,a) \right),$$

akin more familiar versions of Q-learning.

3.6 Results

Figure 8 from [Mnih et al., 2015] provides representative plots indicating how DQN improves the game performance. For each game, DQN is initialized and then updates parameters over the course of many episodes of that game. Performance improves with the number of episodes.

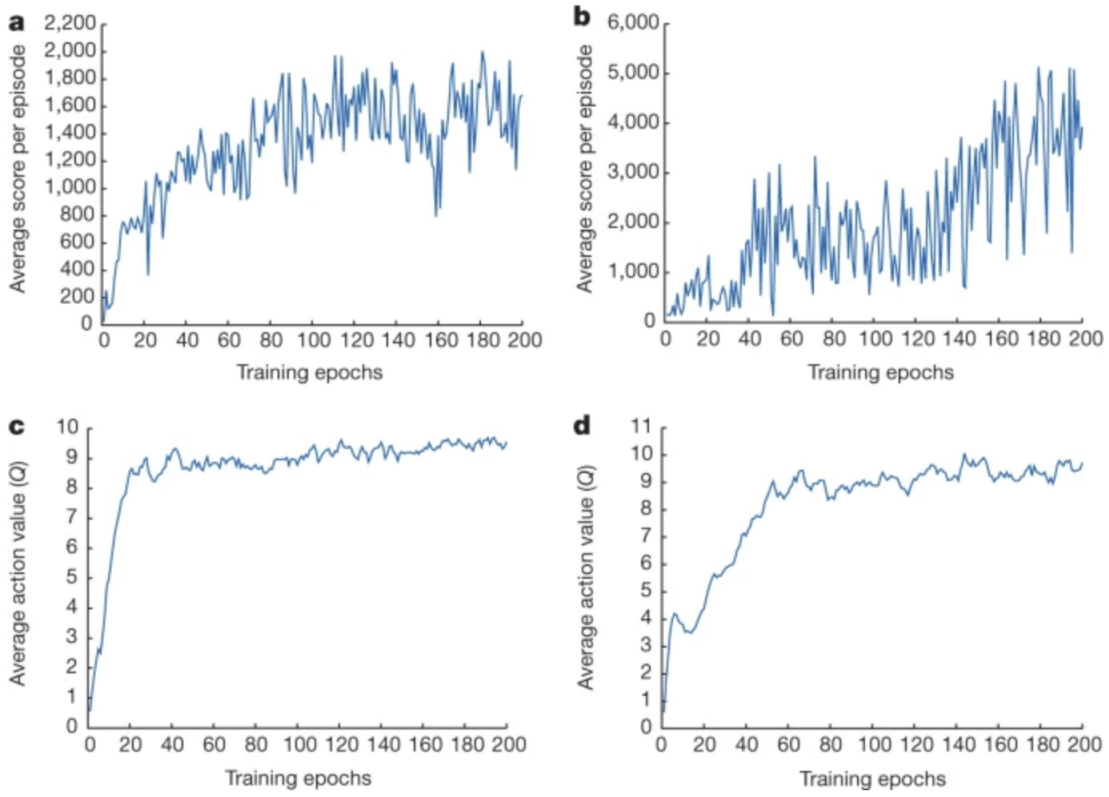


Figure 8: Progress in performance over episodes applying DQN.

Figure 9, also from [Mnih et al., 2015], compares performance against another algorithm and also indicates how DQN after a large number of episodes fares relative to expert human players. DQN outperformed expert humans in most of the games.

References

M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, June 2013. ISSN 1076-9757. doi: 10.1613/jair.3912. URL `http://dx.doi.org/10.1613/jair.3912`.

Shi Dong, Benjamin Van Roy, and Zhengyuan Zhou. Simple agent, complex environment: Efficient reinforcement learning with agent states. *Journal of Machine Learning Research*, 23(255):1–54, 2022.

Tommi Jaakkola, Michael Jordan, and Satinder Singh. Convergence of stochastic iterative dynamic programming algorithms. *Advances in neural information processing systems*, 6, 1993.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

John N Tsitsiklis. Asynchronous stochastic approximation and Q-learning. *Machine learning*, 16(3):185–202, 1994.

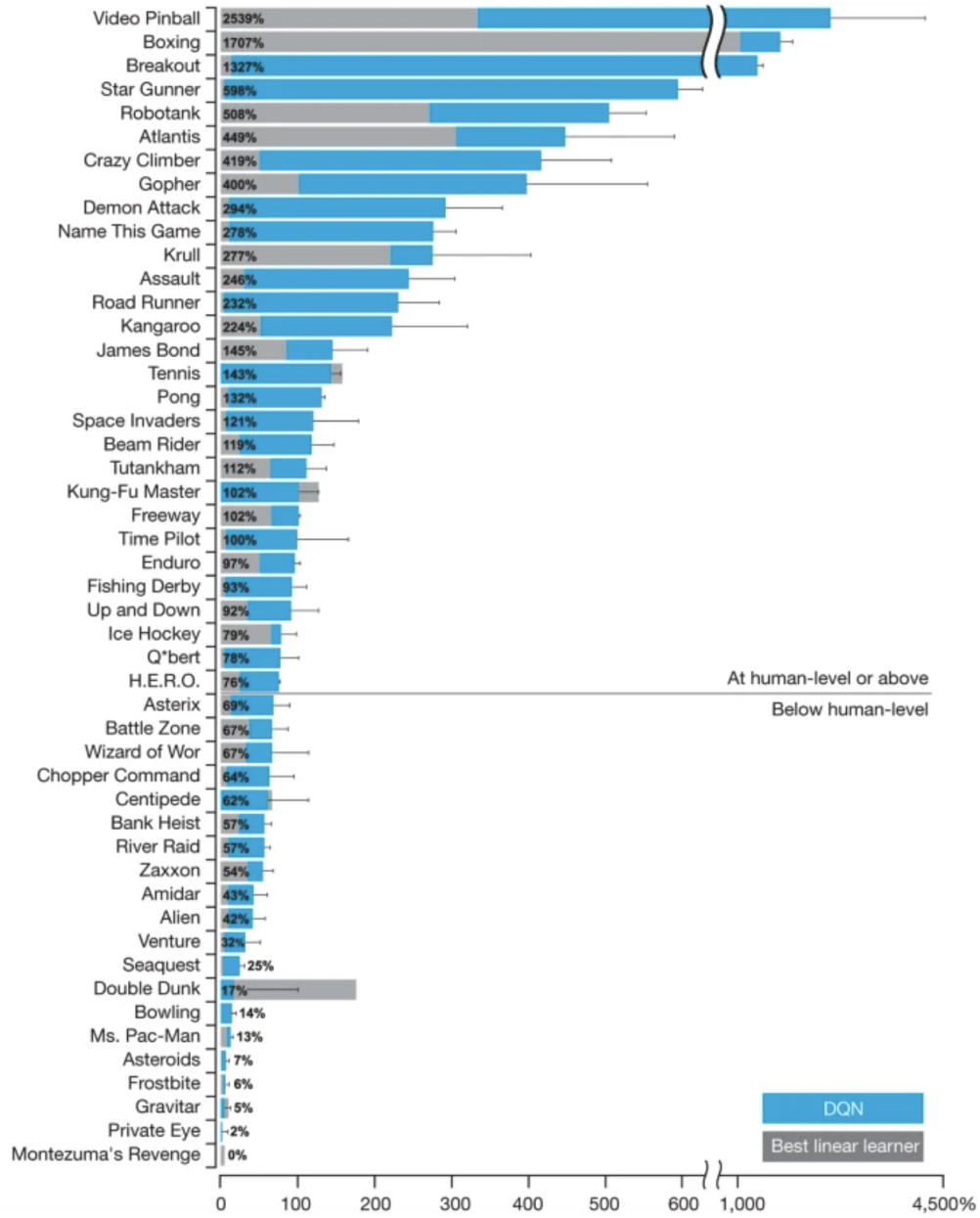


Figure 9: DQN performance relative to another algorithm and relative to expert human players.