Reinforcement Learning

An Introduction

Part I: Tabular Solution Methods

- Multi-Armed Bandits
  - Action-Value Methods
    - Greedy vs. ε-greedy Methods
  - Estimation
    - Incremental Estimation
    - Recency-Weighted Estimation
    - Initial Value Bias
  - Selection
    - UCB Action Selection
  - Gradient Bandit Algorithms
  - Associative Search
- Finite MDPs
  - Concepts
    - Agent-Environment Interface
    - Goals and Rewards
    - Returns and Episodes
    - Policies and Value Functions
  - Bellman Equations: $v_\pi; q_\pi; v_*; q_*$
- Dynamic Programming
  - Policy Evaluation
  - Policy Improvement
  - Policy Iteration
  - Value Iteration
- Monte Carlo Methods
  - MC (on-policy) Prediction
    - Exploring Starts
  - MC (on-policy) Control
    - Without Exploring Starts
  - Off-Policy Prediction
    - Importance Sampling
    - Incremental Implementation
  - Off-Policy Control

1 Multi-Armed Bandits

RL uses training information that provides evaluative rather than instructive feedback – this creates the need for exploration.
We consider a nonassociative setting – the agent needs to learn to act in just one situation.

1.1 A k-armed Bandit Problem

A k-armed bandit is named by analogy to a slot machine with k levers, each of which corresponds to an action which yields a different reward.

Each action has an expected reward, called its *value*, defined as

$$q_*(a) = \mathbb{E}\left[R_t \mid A_t = a\right].$$

Our estimate of this value at time t is denoted $Q_t(a)$.

1.2 Action-Value Methods

Action-value methods are those that estimate the values of actions and use these estimates to select actions.

One natural way to estimate $Q_t(a)$ is by averaging the reward:

$$Q_t(a) = \frac{\sum_{i=1;A_i=a}^{t-1} R_i}{\sum_{i=1;A_i=a}^{t-1} 1}.$$

The simplest action selection rule is to greedy select the action with the highest estimated value:

$$A_t = \operatorname*{argmax}_a Q_t(a).$$

We may also use ε-greedy methods, which select a non-greedy action with probability ε, to ensure that all actions are sampled and we converge to $q_*(a)$.

The following subsections treat either the estimation method (Estim.) or the selection method (Selec.).

1.3 The 10-armed Testbed

Greedy methods usually don't select the optimal action or achieve optimal reward.

A higher value of ε means that the initial stages see high returns, but lower values overtake these in the long run.

The advantage of ε-greedy methods comes from the variance in the reward, and is (roughly speaking) proportional to it.

1.4 (Estim.) Incremental Implementation (Efficient Estimation)

We have seen that the naive way to estimate reward is to average it. Assuming a single action,

$$Q_n = \frac{R_1 + \cdots + R_{n-1}}{n-1}.$$

This is highly memory- and computation-intensive. An incremental update process is more efficient:

$$Q_{n+1} = Q_n + \frac{1}{n}[R_n - Q_n].$$

This is a specific case of a general form of update rule which we will encounter repeatedly:

$$\text{newEstimate} \leftarrow \text{oldEstimate} + \text{stepSize}[\text{target} - \text{oldEstimate}].$$

1.5 (Estim.) Tracking a Nonstationary Problem

In a nonstationary problem (where the reward probabilities change over time), it makes sense to give more weight to recent rewards. This is achieved by using a constant step-size α in place of $\frac{1}{n}$. This results in the estimate being an exponential recency-weighted average of the form

$$Q_{n+1} = (1-\alpha)^n Q_1 + \sum_{i=1}^{n} \alpha(1-\alpha)^{n-1} R_i.$$

Necessary conditions on the α_n sequence for convergence are that its sum should diverge and the sum of the squares of the elements should converge – this is satisfied for the sample-average case of $\alpha_n = \frac{1}{n}$ but not in the constant case. In the latter, therefore, the estimates do not converge but fluctuate according to the most recent rewards (which is not necessarily undesirable).

1.6 (Estim.) Optimistic Initial Values

There is always a bias introduced by the initial estimate Q_1. This disappears in the sample-average case but not in the recency-weighted case. Thus these values are parameters that the user needs to select.

Using high initial values encourages exploitation (temporarily) as all actions yield "low" rewards initially. This is effective on stationary problems, but there are better ways to encourage exploration in general.

1.7 (Selec.) Upper-Confidence-Bound Action Selection

We have seen greedy and ε-greedy action selection methods.

We can select actions non-greedily according to their potential for being optimal. One effective way of this is the decision given by

$$A_t = \underset{a}{\operatorname{argmax}} \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right].$$

The parameter c controls the degree of exploration. The sqrt term is a measure of the variance in the estimate of a's value (it decreases the more we pick a); thus the objective function is an "upper bound" on the possible true value of a.

1.8 (Estim. + Selec.) Gradient Bandit Algorithms

Rather than directly selecting actions, we may assign them *preferences* $H_t(a)$ and select them according to a probability distribution

$$\pi_t(a) = \operatorname{softmax}(H_t)(a).$$

We can apply stochastic gradient ascent to this process by comparing the obtained reward with the current average reward.

$$\begin{aligned} H_{t+1}(A_t) &= H_t(A_t) + \alpha(R_t - \bar{R}_t)(1 - \pi_t(A_t)) \\ H_{t+1}(a) &= H_t(a) - \alpha(R_t - \bar{R}_t)pi_t(a) \text{ for all } a \neq A_t. \end{aligned}$$

1.9 Associative Search

Associative search tasks involve learning a mapping from situations (states) to actions, rather than actions alone. These differ from the full reinforcement learning in that actions have no effect on the next state, but only on the reward.

Such problems are considered in the next chapter.

2 Finite Markov Decision Processes

Finite MDPs involve evaluative feedback, but in an associative setting. Here, actions influence immediate rewards as well as subsequent states, and therefore involve delayed reward and the tradeoff of this with immediate reward.

Here, the estimanda are $q_*(s, a)$ (the value of an action in a state) and $v_*(s)$ (the value of a state given optimal action selection).

2.1 The Agent-Environment Interface

We assume a discrete-time setting, where at each timestep, the agent can observe the state $S_t \in \mathcal{S}$, select an action $A_t \in \mathcal{A}(s)$, and in the next timestep receive a reward $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ and observe the next state S_{t+1}.

The dynamics of an MDP are determined by the probability

$$p(s', r \mid s, a) = \Pr\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\}.$$

This matrix can be marginalized to find various other quantities associated with the process, like

$$p(s' \mid s, a) = \Pr\{S_t = s' \mid S_{t-1} = s, A_{t-1} = a\}$$
$$r(s, a) = \mathbb{E}\left[R_t \mid S_{t-1} = s, A_{t-1} = a\right]$$
$$r(s, a, s') = \mathbb{E}\left[R_t \mid S_{t-1} = s, A_{t-1} = a, S_t = s'\right]$$

2.2 Goals and Rewards

The reward hypothesis, a distinct feature of RL, is a formalization of the notion of "purpose" for the agent – the agent maximizes the expected value of a cumulative sum of a received signal.

2.3 Returns and Episodes

The expected return is some function of the reward sequence, in the simplest case a direct sum of the rewards

$$G_t = R_{t+1} + \cdots + R_T.$$

This is most naturally applicable in cases where the interaction can be delineated into episodes, which are repeated with resets in between – these are called episodic tasks.

Continuing tasks, on the other hand, may go on without limit; here, in order to ensure convergence of G_t we discount the future rewards:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}.$$

2.4 Policies and Value Functions

We have seen how value functions are defined for states and state-action pairs. A *policy* is a mapping from states to probability distributions over actions.

The value function of a state s under a policy π is

$$v_\pi(s) = \mathbb{E}_\pi \left[G_t \mid S_t = s \right];$$

correspondingly for a state-action pair (s, a), we have

$$q_\pi(s, a) = \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a \right].$$

We call estimation methods for these functions that rely on repeated sampling *Monte Carlo methods.*

These value functions satisfy recursive relationships; for example,

$$v_\pi(s) = \sum_{a \in \mathcal{A}(s)} \pi(a \mid s) \sum_{s', r} p(s', r \mid s, a)[r + \gamma v_\pi(s')].$$

This is called the Bellman equation for v_π; v_π is the unique solution to this equation.

Exercise 3.17.

$$q_\pi(s, a) = \sum_{s', r} p(s', r \mid s, a) \left[r + \gamma \sum_{a' \in \mathcal{A}(s')} \pi(a' \mid s') q_\pi(s', a') \right]$$

Exercise 3.18.

$$v_\pi(s) = \sum_{a \in \mathcal{A}(s)} \pi(a \mid s) q_\pi(s, a).$$

Exercise 3.19.

$$q_\pi(s, a) = \sum_{s', r} p(s', r \mid s, a)[r + v_\pi(s')].$$

2.5 Optimal Policies and Optimal Value Functions

Finite MDPs admit a precise definition for optimal policies. We say that π is at least as good as π' if $v_\pi(s) > v_{\pi'}(s)$ for all $s \in \mathcal{S}$; the root(s) of this partial ordering is (are) then the optimal policy(ies) π_*.
They all share the state-value function v_*, which satisfies

$$v_*(s) = \max_\pi v_\pi(s),$$

and the same action-value function, which satisfies

$$q_*(s, a) = \max_\pi q_\pi(s, a).$$

The relationship between these two quantities is

$$q_*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a\right].$$

They also satisfy recursive Bellman optimality equations

$$v_*(s) = \max_a \sum_{s',r} p(s', r \mid s, a)[r + \gamma v_*(s)]$$
$$q_*(s,a) = \sum_{s',r} p(s', r \mid s, a)[r + \gamma \max_{a'} q_*(s', a')].$$

Once the optimal value functions are obtained, it is easy to see that in a finite MDP, any policy that assigns nonzero probabilities only to the maximum-value actions (*i.e.*, which is greedy w.r.t v_*) is an optimal policy.

Note that such a procedure is rarely applicable in real-life scenarios, because of three core assumptions:

- we accurately know the dynamics of the environment;
- we can feasibly solve the Bellman equations symbolically;
- the system is Markovian.

3 Dynamic Programming

DP methods are used to compute optimal policies given a perfect model of the environment as an MDP. While this is unrealistic, the study of DP algorithms is the basis for more advanced RL methods that rely on fewer assumptions.

3.1 Policy Evaluation (Prediction)

We have seen the Bellman equation for finding the value function of a policy π:

$$v_\pi(s) = \sum_{a \in \mathcal{A}(s)} \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a)[r + \gamma v_\pi(s')].$$

We can treat this equation as an update equation, starting with an arbitrary function v_0:

$$v_{k+1}(s) = \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a)[r + \gamma v_k(s')].$$

Then v_π is a fixpoint of this rule, and it can be shown that this rule converges to it.
In implementation, one may also update the state values in place, so that sometimes the new values are used on the RHS instead of the old; this version of the algorithm converges even faster.

3.2 Policy Improvement

If the process is in state s, should we follow the current policy π or not? One way of checking this is to examine the value of

$$q_\pi(s,a) = \sum_{s',r} p(s',r \mid s,a)[r + \gamma v_p i(s')],$$

for $a \neq \pi(s)$, and if it is greater than $v_\pi(s)$, we can pick a instead of $\pi(s)$.

This is a special case of the policy improvement theorem: If π and π' are deterministic policies such that

$$q_\pi(s, pi'(s)) \geq v_\pi(s)$$

for all $s \in \mathcal{S}$, then

$$v_{\pi'}(s) \geq v_\pi(s)$$

for all $s \in \mathcal{S}$.

Following this line of reasoning, one way of (greedily) updating policies is to define

$$\pi'(s) = \operatorname*{argmax}_a q_\pi(s,a).$$

It can be proved that this only fails to improve upon π when it is already optimal.

3.3 Policy Iteration

The two above procedures give us a general method to find the optimal policy – repeatedly evaluate the policy, and use the value function to find a better policy, until we reach the optimal one.

3.4 Value Iteration

It is possible to optimize the policy iteration process via *value iteration*:

$$v_{k+1}(s) = \max_a \sum_{s',r} p(s',r \mid s,a)[r + \gamma v_k(s')].$$

This effectively combines, in one sweep, one sweep each of policy evaluation and policy improvement. This converges much faster than policy iteration.

3.5 Asynchronous Dynamic Programming

Sweeps of the state set may be prohibitively expensive; *asynchronous* DP algorithms are in-place algorithms that do not have this limitation. They update the state values in any order, using the available values of other states. They converge as long as each state is updated (asymptotically) infinitely often.

3.6 Generalized Policy Iteration

Generalized policy iteration (GPI) refers to the general idea of letting policy-evaluation and policy-improvement processes interact, at any level of granularity. The method we have seen lets each process complete before starting the next one; we may also interleave them at the state level.

3.7 Efficiency of Dynamic Programming

DP methods work in polynomial time (in the number of states and actions). This is much better than any direct search in policy space. Furthermore, they have wider applicability than linear programming methods.

4 Monte Carlo Methods

Monte Carlo methods require experience in exchange for complete knowledge of environment.

Here, we consider only episodic tasks, and updates occur only between episodes (not online). We only average complete returns (for methods that use partial returns, see next chapter).

The problem here is now nonstationary, since the return after taking an action in one state depends on actions taken in later states of the same episode. Thus we adapt the idea of GPI to *learn* value functions from experience (rather than compute them from knowledge of the model).

4.1 Monte Carlo Prediction

First, consider the state-value function. The obvious way to estimate it is to average the returns observed after visits to that state. This should converge to the expected value.

The *first-visit MC method* estimates $v_\pi(s)$ as the average of returns following first visits to s – thus a given episode supplies at most one value to the average. The *every-visit MC method* estimates $v_\pi(s)$ as the average of returns following every visit to s; a given episode may supply any number of values to the average.

These methods both rely on generating a complete episode, and looping through it backwards, accumulating the reward and checking the history appropriately.

They both converge to $v_\pi(s)$ as the number of (first) visits goes to infinity.

An important fact about MC methods is that they rely on independent estimates – they do not bootstrap like DP methods.
They are also independent of the number of states.

4.2 Monte Carlo Estimation of Action Values

Without a model, state values alone are insufficient; therefore one of our primary goals is to estimate q_*.

We have analogous methods to the state value function estimation in this case as well – first-visit and every-visit MC methods behave as expected.

The only complication is that many state-action pairs may not be visited, as the $\mathcal{S} \times \mathcal{A}$ space is much greater. This is especially a problem in the case of deterministic policies.
This is the general problem of maintaining exploration, for which we have the assumption of *exploring starts* – we cause the episode to start in a state-action pair and ensure that any pair may be selected as the start.

When learning from actual interaction, and not simulations, the exploring starts assumption is not useful; in this case, the most common approach is to consider only stochastic policies that assign nonzero probability to all actions.

4.3 Monte Carlo Control

The overall idea is to proceed according to GPI; we maintain an approximate policy and an approximate value function, both of which are repeatedly updated.

An MC version of classical policy iteration would complete each evaluation and improvement step before starting the next, alternately. Improvement is done by making the policy greedy, as before:

$$\pi(s) = \operatorname*{argmax}_a q(s, a).$$

This method is guaranteed to converge to the optimal policy.

However, there are two unlikely assumptions: exploring starts, and infinitely long policy evaluation.

For now, consider removing the second assumption. This is an issue even in DP methods, and, as in that case, we have two potential solutions:

- compute error bounds in estimates and use as many episodes as it takes to push them below some threshold – this is unlikely to be feasible – or
- the idea underlying GPI, which is to simply move the estimate towards q_{π_k}, but not get close until after several steps.

An algorithm following the latter method would generate a whole episode, loop through it backwards, at each step updating $q_\pi(s, a)$ and $\pi(s)$.

It is easy to see that if this method converges, the policy must be optimal.

4.4 Monte Carlo Control without Exploring Starts

Now, we try to do away with the first assumption.

The only way to ensure that all actions are selected infinitely often is for the agent to continue to select them; this is achieved by using *soft* policies, which have a nonzero probability assigned to each pair.

Here, we consider ε-greedy policies. Thus the update procedure at the end of each episode is now

$$\pi(a \mid S_t) = \begin{cases} 1 - \varepsilon + \frac{\varepsilon}{|\mathcal{A}(S_t)|} & a = A^* \\ \frac{\varepsilon}{|\mathcal{A}(S_t)|} & a \neq A^*. \end{cases}$$

In all other aspects, the algorithm is identical to the one described in the previous section.

The ε-greedy policy π' is at least as good as π, which can be proved as follows:

$$\begin{aligned} q_\pi(s, \pi'(s)) &= \sum_a \pi'(a \mid s) q_\pi(s,a) \\ &= \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) + (1-\varepsilon) \max_a q_\pi(s,a) \\ &\geq \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) + (1-\varepsilon) \sum_a \frac{\pi(a \mid s) - \frac{\varepsilon}{|\mathcal{A}(s)|}}{1-\varepsilon} q_\pi(s,a) \\ &= \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) - \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a q_\pi(s,a) + \sum_a \pi(a \mid s) q_\pi(s,a). \\ &= v_\pi(s). \end{aligned}$$

Suppose that equality holds – we want to prove that π and π' are optimal. Note that we are only considering optimality in the space of ε-soft policies. To operationalize this restriction, we consider a modified environment, which is identical to the old one except that, when an action a is taken from a state s, then with probability ε, it repicks the action and implements that instead. Thus, a policy π is optimal among ε-soft policies iff it is optimal in this environment. Let the optimal functions of this environment be $\tilde{v}_*$ and $\tilde{q}_*$.A

We know, by definition, that $\tilde{v}_*$ is the unique solution to

$$\begin{aligned} \tilde{v}_* = (1-\varepsilon) \max_a \sum_{s',r} p(s', r \mid s, a)[r + \gamma \tilde{v}_*(s')] \\ + \frac{\varepsilon}{|\mathcal{A}(s)|} \sum_a \sum_{s',r} p(s', r \mid s, a)[r + \gamma \tilde{v}_*(s')]. \end{aligned}$$

We know that

$$q_\pi(s,a) = \sum_{s',r} p(s', r \mid s, a)[r + \gamma v_\pi(s')].$$

Substituting this in the equality that we assumed for $v_\pi(s)$, we obtain the same equation. Since $\tilde{v}_*$ is the unique solution to this, we must have $\tilde{v}_* = v_\pi$.

4.5 Off-Policy Prediction via Importance Sampling

All the methods we have seen so far are *on-policy* methods – they decide actions based on the same policy that is being improved. Methods that use a different policy to decide actions, or *off-policy* methods, are the focus of this section.

The dilemma of learning control methods is to learn action values conditional on optimal behaviour, while maintaining suboptimal behaviour for the sake of exploration. This is the compromise of the previous section.
Off-policy methods use two policies – the target policy, being learned about, and the behaviour policy, used to direct the agent's exploration.

Consider the prediction problem for off-policy methods. Here, both the target π and behaviour b policies are fixed. We have episodes following b, but not π.
A fundamental assumption here is the assumption of coverage:

$$\pi(a \mid s) > 0 \implies b(a \mid s) > 0.$$

This implies that b must be stochastic in states where it is different from π, which may be deterministic.

We utilize *importance sampling*, by weighting returns according to the relative probability of their trajectories occurring under π and b. This value is

$$\rho_{t:T-1} = \prod_{k=t}^{T-1} \frac{\pi(A_k \mid S_k)}{b(A_k \mid S_k)}.$$

Each individual probability depends on the MDP parameters, but this cancels out in the ratio.

We have that $\mathbb{E}\left[G_t \mid S_t = s\right] = v_b(s)$, but we do not want this quantity; we want $v_\pi(s)$, which we obtain as

$$\mathbb{E}\left[\rho_{t:T-1} G_t \mid S_t = s\right] = v_\pi(s).$$

Ordinary importance sampling simply averages the estimated returns

$$V(s) = \frac{\sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1} G_t}{|\mathcal{T}(s)|},$$

where $\mathcal{T}(s)$ represents the set of timesteps that we are considering for state s.
We may also use weighted importance sampling, which uses a weighted average

$$V(s) = \frac{\sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1} G_t}{\sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1}}.$$

In the first-visit case, o. importance sampling is unbiased, while w. importance sampling has a nonzero bias that converges asymptotically to zero.
On the other hand, o. importance sampling has unbounded variance, while the variance of w. importance sampling converges to zero.

In the every-visit case, both methods are biased, although the bias still vanishes in the limit.

4.6 Incremental Implementation

In the case of on-policy methods, the techniques used in previous chapters directly apply. Off-policy methods require separate consideration for o. and w. importance sampling.

In the ordinary case, we can use incremental methods, but using scaled returns $\rho_{t:T(t)-1}$ instead of rewards as in Chapter 2.

As, in w. importance sampling, we need a weighted average, the incremental algorithm has to be modified. This can be achieved with the update rules

$$V_{n+1} = V_n + \frac{W_n}{C_n}[G_n - V_n]$$

$$C_{n+1} = C_n + W_{n+1}$$

where $C_0 = 0$ and V_1 is arbitrary. G_n is the single additional return at each datapoint.

4.7 Off-Policy Monte Carlo Control

An algorithm implementing all the ideas outlined above (off-policy learning, weighted importance sampling, and incremental implementation) would then proceed as follows.

For a single episode, pick a soft policy b that is nonzero wherever π is nonzero. Generate an episode and loop backwards through it. Maintain Q and C as denoted above.
At each timestep t, update

$$\begin{aligned} G &\leftarrow \gamma G + R_{t+1} \\ C(S_t, A_t) &\leftarrow C(S_t, A_t) + W \\ Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)}[G - Q(S_t, A_t)] \\ W &\leftarrow W \frac{1}{b(A_t \mid S_t)} \end{aligned}$$

Then, update π according to the current estimate of Q.

[didn't understand this part]
A potential problem with this method is that it only learns from the tails of

episodes which only contain greedy actions. This can make learning extremely slow, particularly when nongreedy actions are common.

5 Temporal-Difference Learning

TD learning is a combination of MC and DP ideas – it enables learning directly from experience, without a model of the environment, at the same time utilizing bootstrapping to update estimates.

5.1 TD Prediction

Both TD and MC methods update their estimate V of v_π for the nonterminal states S_t in that experience. While MC methods wait to know the return G_t following the visit, *e.g.*,

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)]$$

(which we call *constant-*α MC), TD methods wait only until the next step, using previous estimates for their updates. For example,

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)].$$

This can be done at each step of the episode.

Effectively, we estimate the target to be $R_{t+1} + \gamma V(S_{t+1})$. This is called one-step TD, or TD(0).
TD therefore combines the sampling of MC with the bootstrapping of DP.

The term in brackets is the error, which we see throughout RL:

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t),$$

which is computable at timestep $t+1$.

Since the array V does not change during the episode in MC methods, the MC error can be written as the sum of TD errors:

$$\begin{aligned} G_t - V(S_t) &= R_{t+1} + \gamma G_{t+1} - V(S_t) + \gamma V(S_{t+1}) - \gamma V(S_{t+1}) \\ &= \delta_t + \gamma(G_{t+1} - V(S_{t+1})) \\ &= \sum_{k=t}^{T-1} \gamma^{k-t} \delta_k. \end{aligned}$$

5.2 Advantages of TD Prediction Methods

One obvious advantage of TD models is that they do not require a model of the environment.

They are also naturally implemented in an online, incremental fashion. This avoids the consideration of long episodes, continual tasks and episodes which include experimental actions.

TD(0) also converges to v_π with probability 1. In practice, this has even found to be faster than constant-α MC methods.

5.3 Optimality of TD(0)

Suppose that only a finite amount of experience is available. In this case, we may present the experience repeatedly until the method converges. Thus the current estimate V is changed only once per pass (over the experience), by the sum of all the increments. This is called batch updating.

Under batch updating, TD(0) converges to a single answer (as long as α is sufficiently small); constant-α MC also converges under similar conditions, but to a different answer.

In fact, constant-α MC converges to the sample average of the actual returns, *i.e.*, the estimates that minimize RMS error on the training data. TD(0), on the other hand, finds the maximum-likelihood estimate of the data.

5.4 Sarsa: On-Policy TD Control

As usual, we follow the idea of GPI, only this time using TD methods for evaluation and prediction. In this section, we consider on-policy control.

The first step is to learn an action-value function for the current policy:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)].$$

This uses every element from the quintuple $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$.

It is straightforward to use this update in a GPI-style algorithm that continually estimates q_π and updates π simultaneously.

Sarsa converges with probability 1 to an optimal policy as long as all state-value pairs are visited infinitely often and the policy converges to the greedy policy.

5.5 Q-Learning: Off-Policy TD Control

The development of Q-learning, an off-policy TD algorithm, was one of the early breakthroughs in RL. This is defined by

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)].$$

The policy still has an effect here, as it determines which state-action pairs are visited; however, as long as all pairs continue to be updated, this converges to q_*.

5.6 Expected Sarsa

What if we use the expected value instead of the maximum? This would take into account how likely each action is under the current policy, as in:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \sum_a \pi(a \mid S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t)].$$

This moves deterministically in the same direction as Sarsa moves in expectation – therefore it is called *expected Sarsa*.

The advantage of expected Sarsa is that it eliminates variance due to the random selection of A_{t+1}. It generally performs better than Sarsa for the same experience. It can also work with $\alpha = 1$, while Sarsa suffers under larger values of α.

If π is the greedy policy, but episodes are generated according to an exploratory policy b, then expected Sarsa is exactly Q-learning.

5.7 Maximization Bias and Double Learning

We have implicitly assumed so far that the estimate of the true maximum can be taken to be the maximum of the estimates. However, this leads to a positive *maximization bias*.

One way to avoid this bias is to divide the sample space of actions (plays). Suppose that we do this and obtain two independent estimates $Q_1(a)$ and $Q_2(a)$. Then we use one of these to determine the maximizing action, and the other to find its value. We can repeat this process in reverse as well.

$$Q_2(A^*) = Q_2(\operatorname*{argmax}_a Q_1(a))$$

$$Q_1(A^*) = Q_1(\operatorname*{argmax}_a Q_2(a))$$

We therefore maintain two estimates of Q, and update each with a 50% probability at each timestep, using the other to find the value of the maximizing action. For example,

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q_2 \left(S_{t+1}, \operatorname*{argmax}_a Q_1(S_{t+1}, a) \right) - Q_1(S_t, A_t) \right].$$

Thus the two approximations are treated completely symmetrically.

The behaviour policy can be informed by both estimates; *e.g.*, it may be ε-greedy in $Q_1 + Q_2$.

Double learning can be applied to Sarsa and expected Sarsa as well.

5.8 Games, Afterstates and Other Special Cases

In some cases, we have complete knowledge of the state of the system after taking a certain action in a certain state. Furthermore, many state-action pairs may lead to the same "afterstate," making it redundant to learn estimates for each of them separately. Thus it makes sense to estimate the value of these afterstates, instead of the current state or the state-action pairs. One common example of this is deterministic games like chess or tic-tac-toe.

The principles developed above (GPI; on- or off-policy; double learning) apply equally to afterstate methods.

6 n-Step Bootstrapping

n-Step TD methods are a generalization of MC and one-step TD methods. The benefit of this method is that it allows you to slow down bootstrapping (as it is most effective over longer periods), while still changing the action at every time step.

6.1 n-Step TD Prediction

We have seen that MC methods rely on the entire sequence of rewards until the completion of the episode, while TD(0) simply uses the existing update as a proxy along with one future reward (bootstraps). A natural intermediate process would be to consider some finite number of future rewards, but not go all the way to the end of the episode – this is the idea behind n-step TD methods.

Thus, the generalized target is

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n}),$$

which is an apprxoimation of the full return for any n.

Note that any algorithm using this estimate has to wait n timesteps to update:

$$V_{t+n}(S_t) = V_{t+n-1}(S_t) + \alpha[G_{t:t+n} - V_{t+n-1}(S_t)].$$

The n-step returns have what is called the error-reduction property:

$$\max_s |\mathbb{E}_\pi \left[G_{t:t+n|S_t=s} - v_\pi(s)\right] \leq \gamma^n \max_s |V_{t+n-1}(s) - v_\pi(s)|.$$

Thus these methods converge to the correct predictions.

6.2 n-Step Sarsa: On-Policy Control

For Sarsa, we redefine n-step returns in terms of action values:

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n Q_{t+n-1}(S_{t+n}, A_{t+n}).$$

Then we naturally have

$$Q_{t+n}(S_t, A_t) = Q_{t+n-1}(S_t, A_t) + \alpha[G_{t:t+n} - Q_{t+n-1}(S_t, A_t)].$$

For expected Sarsa, we define the target as

$$G_{t:t+n} = R_{t+1} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \bar{V}_{t+n-1}(S_{t+n}),$$

where

$$\bar{V}_t(s) = \sum_a \pi(a \mid s) Q_t(s, a).$$

6.3 n-Step Off-Policy Learning

To make a simple off-policy version of n-step TD, we can just weight the update by the importance sampling ratio:

$$V_{t+n}(S_t) = V_{t+n-1}(S_t) + \alpha \rho_{t:t+n-1}[G_{t:t+n} - V_{t+n-1}(S_t)],$$

where

$$\rho_{t:h} = \prod_{k=t}^{\min(h, T-1)} \frac{\pi(A_k \mid S_k)}{b(A_k \mid S_k)}.$$

This can be extended to Sarsa in the natural way:

$$Q_{t+n}(S_t, A_t) = Q_{t+n-1}(S_t, A_t) + \alpha \rho_{t+1:t+n}[G_{t:t+n} - Q_{t+n-1}(S_t, A_t)].$$

Note that the importance sampling ratio starts and ends one step later, since we are updating a state-action pair.

In the case of expected Sarsa, the equation would of course have the modified definition of G that we have seen above; furthermore, the importance sampling ratio would only be $\rho_{t+1:t+n-1}$ as the action actually taken is unimportant, since we take the average over all actions.

6.4 Off-Policy Learning without Importance Sampling: The n-Step Tree Backup Algorithm

We can implement an off-policy algorithm without importance sampling using the tree backup method.

Consider a tree of states and actions up to a depth n. At each point, the actions not taken create "dangling" nodes. We include these in the estimate of the value, by weighting them with their probability of occurring under policy π. The actual actions taken do not contribute.

Concretely, the $n = 1$ case gives us the target

$$G_{t:t+1} = R_{t+1} + \gamma \sum_a \pi(a \mid S_{t+1}) Q_t(S_{t+1}, a).$$

The $n = 2$ case gives us

$$G_{t:t+2} = R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a \mid S_{t+1}) Q_{t+1}(S_{t+1}, a)$$
$$+ \gamma \pi(A_{t+1} \mid S_{t+1}) \left(R_{t+2} + \gamma \sum_{a} \pi(a \mid S_{t+2}) Q_{t+1}(S_{t+2}, a) \right).$$

Thus we have the recursive definition

$$G_{t:t+n} = R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a \mid S_{t+1}) Q_{t+n-1}(S_{t+1}, a) + \gamma \pi(A_{t+1} \mid S_{t+1}) G_{t+1:t+n},$$

and the update rule

$$Q_{t+n}(S_t, A_t) = Q_{t+n-1}(S_t, A_t) + \alpha [G_{t:t+n} - Q_{t+n-1}(S_t, A_t)].$$

This rule is applied at each step of the episode, followed by an update of π to ensure that it is greedy w.r.t Q.