## Elements of Reinforcement Learning

A policy, a reward signal, a value function, and, optionally, a model of the environment.

A **policy** defines the learning agent’s way of behaving at a given time. Roughly speaking, a policy is a mapping from perceived states of the environment to actions to be taken when in those states. In some cases the policy may be a simple function or lookup table, whereas in others it may involve extensive computation such as a search process. The policy is the core of a reinforcement learning agent in the sense that it alone is sufficient to determine behavior. In general, policies may be stochastic, specifying probabilities for each action.

A **reward signal** defines the goal of a reinforcement learning problem. On each time step, the environment sends to the reinforcement learning agent a single number called the reward. The agent’s sole objective is to maximize the total reward it receives over the long run. The reward signal is the primary basis for altering the policy; if an action selected by the policy is followed by low reward, then the policy may be changed to select some other action in that situation in the future. In general, reward signals may be stochastic functions of the state of the environment and the actions taken.

Whereas the reward signal indicates what is good in an immediate sense, a **value function** specifies what is good in the long run. *Roughly speaking, the value of a state is the total amount of reward an agent can expect to accumulate over the future, starting from that state.*

Whereas rewards determine the immediate, intrinsic desirability of environmental states, values indicate the long-term desirability of states after taking into account the states that are likely to follow and the rewards available in those states. For example, a state might always yield a low immediate reward but still have a high value because it is regularly followed by other states that yield high rewards.

Without rewards there could be no values, and the only purpose of estimating values is to achieve more reward. Nevertheless, it is values with which we are most concerned when making and evaluating decisions. Action choices are made based on value judgments. We seek actions that bring about states of highest value, not highest reward, because these actions obtain the greatest amount of reward for us over the long run.

The **model** is a model of the environment. This is something that mimics the behavior of the environment, or more generally, that allows inferences to be made about how the environment will behave. For example, given a state and action, the model might predict the resultant next state and next reward. Models are used for planning, by which we mean any way of deciding on a course of action by considering possible future situations before they are actually experienced. Methods for solving reinforcement learning problems that use models and planning are called model-based methods, as opposed to simpler model-free methods that are explicitly trial-and-error learners—viewed as almost the opposite of planning.

While we are playing, we change the values of the states in which we find ourselves during the game. We attempt to make them more accurate estimates of the probabilities of winning. To do this, we “back up” the value of the state after each greedy move to the state before the move. More precisely, the current value of the earlier state is updated to be closer to the value of the later state. This can be done by moving the earlier state’s value a fraction of the way toward the value of the later state.

If we let St denote the state before the greedy move, and St+1 the state after the move, then the update to the estimated value of St, denoted V (St), can be written as:

**V (St) V (St) + α[V (St+1) − V (St)]**

where α is a small positive fraction called the step-size parameter, which influences the rate of learning.

This update rule is an example of a **temporal-difference learning method**, so called because its changes are based on a difference, V(St+1) − V(St), between estimates at two successive times.

1. If the step-size parameter is reduced properly over time, then this method converges, for any fixed opponent, to the true probabilities of winning from each state given optimal play by our player. Furthermore, the moves then taken (except on exploratory moves) are in fact the optimal moves against this (imperfect) opponent. In other words, the method converges to an optimal policy for playing the game against this opponent.
2. If the step-size parameter is not reduced all the way to zero over time, then this player also plays well against opponents that slowly change their way of playing.

## Differences between evolutionary methods and methods that learn value functions.

To evaluate a policy an **evolutionary method** holds the policy fixed and plays many games against the opponent or simulates many games using a model of the opponent. The frequency of wins gives an unbiased estimate of the probability of winning with that policy and can be used to direct the next policy selection.

But each policy change is made only after many games, and only the final outcome of each game is used: what happens during the games is ignored.

**Value function methods**, in contrast, allow individual states to be evaluated. In the end, evolutionary and value function methods both search the space of policies, but *learning a value function takes advantage of information available during the course of play*.

It is a striking feature of the reinforcement learning solution that it can achieve the effects of planning and lookahead without using a model of the opponent and without conducting an explicit search over possible sequences of future states and actions.

The most important feature distinguishing reinforcement learning from other types of learning is that it uses training information that evaluates the actions taken rather than instructs by giving correct actions. This is what creates the need for active exploration, for an explicit search for good behavior.

1. **Purely evaluative feedback** indicates how good the action taken was, but not whether it was the best or the worst action possible.
2. **Purely instructive feedback**, on the other hand, indicates the correct action to take, independently of the action actually taken.

## Action Value Methods

If we let Action space = k, then each of the k actions has an expected or mean reward given that that action is selected; let us call this the value of that action.

We denote the action selected on time step t as At, and the corresponding reward as Rt. The value then of an arbitrary action a, denoted q\*(a), is the expected reward given that a is selected:

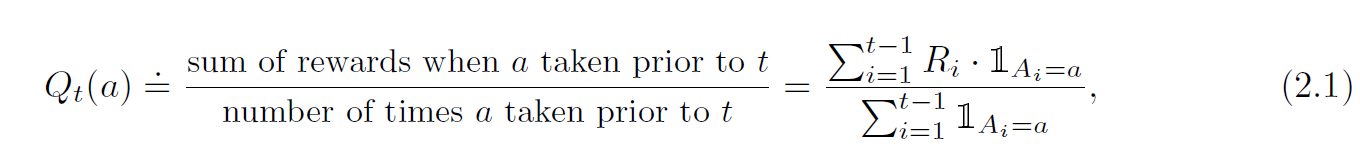
**q\*(a) = E[Rt |At=a]**

We denote the estimated value of action **a**, at time step t as Qt(a). We would like Qt(a) to be close to q\*(a).

If you maintain estimates of the action values, then at any time step there is at least one action whose estimated value is greatest. We call these the **greedy actions**. When you select one of these actions, we say that you are **exploiting** your current knowledge of the values of the actions.

If instead you select one of the **nongreedy actions**, then we say you are **exploring**, because this enables you to improve your estimate of the nongreedy action’s value.

The true value of an action is the mean reward when that action is selected. One natural way to estimate this is by averaging the rewards actually received:



where denotes the random variable that is 1 if predicate is true and 0 if it is not. If the denominator is zero, then we instead define Qt(a) as some default value, such as 0. As the denominator goes to infinity, by the law of large numbers, Qt(a) converges to q\*(a). We call this the sample-average method for estimating action values because each estimate is an average of the sample of relevant rewards.

## Action Selection (Policy)

The simplest action selection rule is to select one of the actions with the highest estimated value, that is, one of the greedy actions as defined in the previous section. If there is more than one greedy action, then a selection is made among them in some arbitrary way, perhaps randomly.

We write this **greedy action selection method** as

**At = argmaxa Qt(a)**  (2.2)

where argmaxa denotes the action a, for which the expression that follows is maximized (again, with ties broken arbitrarily).

**Greedy action selection always exploits current knowledge** to maximize immediate reward; it spends no time at all sampling apparently inferior actions to see if they might really be better. A simple alternative is to behave greedily most of the time, but every once in a while, say with small probability ϵ, instead select randomly from among all the actions with equal probability, independently of the action-value estimates.

We call methods using this near-greedy action selection rule **ϵ-greedy methods**. An advantage of these methods is that, in the limit as the number of steps increases, every action will be sampled an infinite number of times, thus ensuring that all the Qt(a) converge to q\*(a). This of course implies that the probability of selecting the optimal action converges to greater than 1 − ϵ, that is, to near certainty.

**The advantage of ϵ-greedy over greedy methods depends on the task**. For example, suppose the reward variance had been larger, say 10 instead of 1. With noisier rewards it takes more exploration to find the optimal action, and ϵ-greedy methods should fare even better relative to the greedy method.

On the other hand, if the reward variances were zero, then the greedy method would know the true value of each action after trying it once.

**It is also possible to reduce** ϵ **over time to try to get the best of higher exploration initially and higher exploitation later, as the stability of rewards increases with time(steps).**

*Exercise 2.1:* In ϵ-greedy action selection, for the case of two actions and ϵ = 0.5, what is the probability that the greedy action is selected?

*Solution*:

|  |
| --- |
| p(exploration) = ϵ = 0.5.  p(exploitation) = (1- ϵ) = 0.5 |
| Probability = p(exploration) + p(tie break) |
| = ϵ + (1- ϵ)/2 |
| = 0.5 + 0.5\*0.5 = 0.75 = 75% |

*Exercise 2.2:* Consider a k-armed bandit problem with k = 4 actions, denoted 1, 2, 3, and 4. Consider applying to this problem a bandit algorithm using ϵ-greedy action selection, sample-average action-value estimates, and initial estimates of Q1(a) = 0, for all a.

Suppose the initial sequence of actions and rewards is

|  |
| --- |
| A1 = 1, R1 = −1, |
| A2 = 2, R2 = 1, |
| A3 = 2, R3 = −2, |
| A4 = 2, R4 = 2, |
| A5 = 3, R5 = 0. |

On some of these time steps the ϵ case may have occurred, causing an action to be selected at random. On which time steps did this definitely occur? On which time steps could this possibly have occurred?

*Solution*:

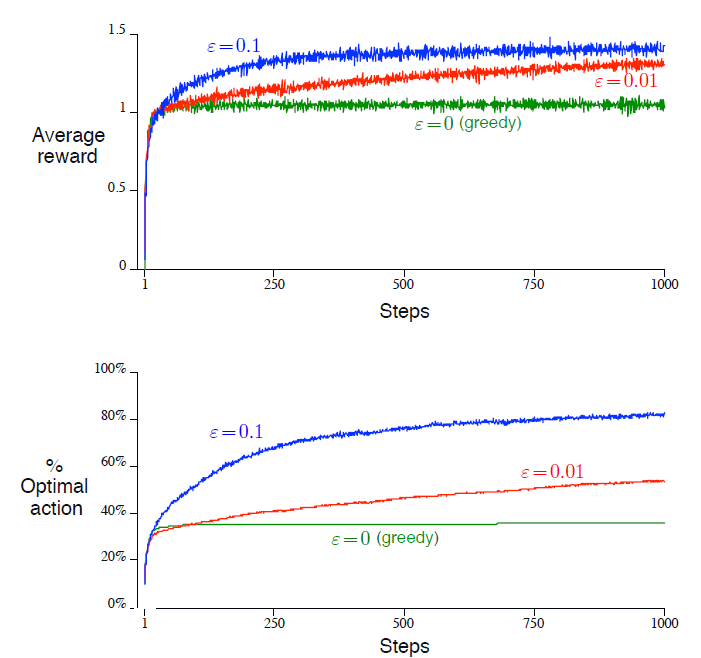
The sequence of values for tuples S1, S2, S3, S4, S5, at each time-step is presented below:

* Time 1: 0, 0, 0, 0, 0
* Time 2: -1, 0, 0, 0, 0
* Time 3: -1, 1, 0, 0, 0
* Time 4: -1, -1, 0, 0, 0
* Time 5: -1, 1, 0, 0, 0
* Time 6: -1, 1, 0, 0, 0

The sequence of steps associated to the certainty of ε having occurred is shown next:

1. Maybe: Q(*ak*) = 0 ∀k at this point, so choosing A₁ = 1 could have happened as a random choice due to ε, or by randomly choosing an action when tie-breaking.
2. Maybe: Q(*a1*) < 0 at this point, so choosing A2 = 2 could have happened as a random choice due to ε, or by randomly choosing an action when tie-breaking.
3. Maybe: Q(*a1*) < 0, Q(*a2*) = 1 at this point, so choosing A3 = 2 could have happened due to ε, or by selecting the option action with highest Q at this point, which is a2.
4. Yes: Q(ak) < 0 for k ∈ {1, 2} at this point, so choosing A4 = 2 is necessarily a random decision, otherwise A3, A4,or A5 with Q=0 would have been chosen.
5. Yes: (*a1*) < 0, Q(*a2*) = 1 at this point, so choosing A5 = 3 is necessarily a random decision, otherwise A2 would have been chosen.

*Exercise 2.3:*In the comparison of an algorithm with ε = 0.01 with another one with ε = 0.1, which method will perform best in the long run in terms of cumulative reward and probability of selecting the best action? How much better will it be? Express your answer quantitatively.



*Solution*:

The method that will perform best in the long run is ε-greedy with ε = 0.01. Though it seems like the ε-greedy method with ε = 0.1 works better for 1000 steps, in the long run ε = 0.01 should outperform it.

This is true because in the limit when steps tend to infinity, both methods will have the expected rewards converge to the true reward value. *Since ε = 0.01 has a lower degree of randomness, it will capture a higher amount of reward*.

In either case any method will have an infinite reward, though the area under the curve will be bigger for ε = 0.01 if we were to select a given step much bigger than 1000 and evaluate the curve until that timestamp.

Quantitatively, we can estimate the reward obtained in each step by calculating the expected reward knowing q\*(a) (the real value for each action).

The expected reward value of a random guess is approximately 0, which means that a random guess' expected reward is 0.

For ε = 0.1, there's a 90% chance of having a 1.5 reward value, and a .1% of having a random guess, which means a 0 reward. This results in an expected reward of 1.5 \* .9 = 1.35.

We can estimate the reward for ε = 0.01 in a similar way. For ε = 0.01, there's a **99% chance** of having a 1.5 reward value, and a **.01% chance** of having a random guess with 0 reward.

This results in an expected reward of 1.5 \* .99 = 1.485. Thus, ε-greedy with ε = 0.01 results in an expected reward value which is 0.135 units higher than ε-greedy with ε = 0.1.

## The action-value methods we have discussed so far, all **estimate action values as sample averages of observed rewards**.

How can these averages be computed in a computationally efficient manner, and with constant memory and constant per-time-step computation?

Let Ri denote the reward received after the ith selection of this action, and let Qn denote the estimate of its action value after it has been **selected n − 1 times**, which we can now write simply as:

A picture containing font, text, line, diagram

Description automatically generated

This computation can be simplified as shown below:

A picture containing text, handwriting, font, diagram

Description automatically generated

which holds even for n = 1, obtaining Q2 = R1 for arbitrary Q1. This implementation requires memory only for Qn and n, and only the small computation (2.3) for each new reward.

The general form of this calculation is:



The expression [Target− OldEstimate] is an error in the estimate. It is reduced by taking a step toward the “Target.” The target is presumed to indicate a desirable direction in which to move, though it may be noisy. In the case above, for example, the target is the nth reward.

The step-size parameter (**StepSize**) used in the incremental method (2.3) changes from time step to time step. In processing the nth reward for action a, the method uses the step-size parameter 1/n. We denote the step-size parameter by α or, more generally, by αt(a).

|  |
| --- |
| # Qn+1 = Qn + (1/n)[(Rn - Qn)]  def sample\_average(self, Q, last\_action, last\_reward, N):      Q[last\_action] += (last\_reward - Q[last\_action]) / N[last\_action] |

## Non-stationary Reward

The above averaging method is appropriate for stationary bandit problems, that is, for bandit problems in which the reward probabilities do not change over time.

We often encounter reinforcement learning problems that are **effectively nonstationary**. In such cases it makes sense to *give more weight to recent rewards than to long-past rewards*.

One of the most popular ways of doing this is to use a constant step-size parameter.

The incremental update rule (2.3) for updating an average Qn of the n − 1 past rewards can be modified to be:

Qn+1 = Qn + α [Rn – Qn] (2.5)

where the step-size parameter α Є (0, 1] is constant.

This results in Qn+1 being a weighted average of past rewards and the initial estimate Q1:

A picture containing text, font, screenshot, handwriting

Description automatically generated

The weight, (1 − α)n−i, given to the reward Ri depends on how many rewards ago, (n−i) it was observed. The quantity 1− α is less than 1, and thus *the weight given to Ri decreases as the number of intervening rewards increases*. **In fact, the weight decays exponentially according to the exponent on 1 − α**.

(If 1− α = 0, then all the weight goes on the very last reward, Rn, because of the convention that 00 = 1.) Accordingly, this is sometimes called an **exponential recency-weighted average**.

|  |
| --- |
| #               n      # (1-α)^n Q1 +  Σ  α(1 − α)^(n−i) Ri      #              i=1      def exponential\_recency\_weighted\_average(self, Q, last\_action, last\_reward,              Q1, R, alpha, \*\*kwargs):    n = len(R[last\_action])    Q[last\_action] = pow(1 - alpha, n) \* Q1[last\_action]    for i in range(1, n):              Q[last\_action] += alpha \* pow(1-alpha, n-i) \* R[last\_action][i] |

## Upper-Confidence-Bound Action Selection

Exploration is needed because there is always uncertainty about the accuracy of the action-value estimates. The greedy actions are those that look best at present, but some of the other actions may actually be better. ϵ-greedy action selection forces the non-greedy actions to be tried, but *indiscriminately*, with no preference for those that are nearly greedy or particularly uncertain. It would be better to select among the non-greedy actions according to their potential for actually being optimal, taking into account both how close their estimates are to being maximal and the uncertainties in those estimates.

One effective way of doing this is to select actions according to:

A picture containing font, text, line, white

Description automatically generatedA picture containing font, white, logo, graphics

Description automatically generated

where ln t denotes the natural logarithm of t, Nt(a) denotes the number of times that action **a** has been selected prior to time t (0 ... t-1), and the number **c > 0** controls the **degree of exploration**.

**If Nt(a) = 0, then a is considered to be a maximizing action.** The idea of this upper confidence bound (UCB) action selection is that the square-root term is a *measure of the uncertainty or variance in the estimate of a’s value*. The quantity being max’ed over is thus a sort of upper bound on the possible true value of action a, with c determining the confidence level.

Each time a is selected the uncertainty is presumably reduced: Nt(a) increments, and, as it appears in the denominator, the uncertainty term decreases. On the other hand, each time an action other than a is selected, t increases but Nt(a) does not; because t appears in the numerator, the uncertainty estimate increases.

The use of the natural logarithm means that the increases get smaller over time but are unbounded; all actions will eventually be selected, but actions with lower value estimates, or that have already been selected frequently, will be selected with decreasing frequency over time.

This is basically an improvement over ϵ-greedy action selection. It forces the non-greedy actions to be tried, but *not* *indiscriminately*. Instead Actions with higher value estimates will be picked more often. Hence UCB converges fast to the ideal reward and maintains the curve better than ϵ-greedy.

A picture containing text, screenshot, line, plot

Description automatically generated

A graph with blue and orange lines

Description automatically generated with low confidence