Reinforcement Learning Notes

UNIT-I

Introduction

Overview

Supervised learning = learning with labels defined by human; Unsupervised learning = finding patterns in data. Reinforcement learning is a 3rd machine learning paradigm, inwhich the agent tries to maximize its reward signal.

Exploration versus exploitation problem - agent wants to do what it has already done to maximize reward by exploitation, but there may be a bigger reward available if it were to explore.

RL is based on the model of human learning, similar to that of the brain's reward system.

Elements of Reinforcement Learning

Policy Defines the agent's way of behaving at any given time. It is a mapping from the perceived states of the environment to actions to be taken when in those states.

Reward Signal The reward defines the goal of the reinforcement learning problem. At eachtime step, the environment sends the RL agent a single number, a *reward*. It is the agent's sole objective to maximize this reward. In a biological system, we might think of rewards as analogous to pain and pleasure. The reward sent at any time depends on the agent's current action and the agent's current state. If my state is hungry and I choose the action of eating, I receive positive reward.

Value function Reward functions indicate what is good immediately, but value functions specify what is good in the long run. The value function is the total expected reward an agent is likely to accumulate in the future, starting from a given state. E.g. a state might always yield a low immediate reward, but is normally followed by a string of states that yield high reward. Or the reverse. Rewards are, in a sense, primary, whereas values, as predictions of rewards, are secondary. Without rewards there could be no value. Nevertheless, it is values with which we are most concerned when evaluating decisions. We seek actions that bring the highest value, not the highest reward, because they obtain the greatest amount of reward over the long run. Estimating values is not trivial, and efficiently and accurately estimating them is the core of RL.

Model of environment (optionally) Something that mimics the behaviour of the true envi- ronment, to allow inferences to be made about how the environment will behave. Given a state and action, the model might predict the resultant next state and next reward. They are used for *planning*, that is, deciding on a course of action by considering possible future situations before they are actually experienced.

Multi-arm Bandits

RL evaluates the actions taken rather than instructs correct actions like other forms of learning.

An n-Armed Bandit Problem

The problem

You are faced repeatedly with a choice of *n* actions.

After each choice, you receive a reward from a stationary probability distribution.

Objective is to maximize total reward over some time period, say 100 time steps.

Named after of slot machine (one-armed bandit problem), but *n* levers instead of 1.

Each action has an expected or mean reward based on its probability distribution. We shall call this the *value* of the action. We do not know these values with certainty.

Because of this uncertainty, there is always an exploration vs exploitation problem. We always have one action that we deem to be most valuable at any instant, but it is highly likely, at least initially, that there are actions we are yet to explore that are more valuable.

Action-Value Methods

The estimated action value is

$$Q_t(a) = \frac{R_1 + R_2 + \cdots + R_{N_t(a)}}{N_t(a)} \qquad (1)$$

The true value (mean reward) of an action is q, but the estimated value at the tth time-step is Q(a), given by Equation 1 (our estimate after N selections of an action yielding N rewards).

The greedy action selection method is

$$A_t = \operatorname*{argmax}_a Q_t(a) \qquad (2)$$

Simplest action selection rule is to select the action with the highest estimated value.

argmax$_a$ means the value of a for which Q_t is maximised.

ϵ-greedy methods are where the agent selects the greedy option most of the time, andselects a random action with probability ϵ.

Three algorithms are tried: one with e=0 (pure greedy), one with e=0.01 and anotherwith e=0.1

Greedy method gets stuck performing sub-optimal actions.

e=0.1 explores more and usually finds the optimal action earlier, but never selects it morethat 91% of the time.

e=0.01 method improves more slowly, but eventually performs better than the e=0.1method on both performance measures.

It is possible to reduce e over time to try to get the best of both high and low values.

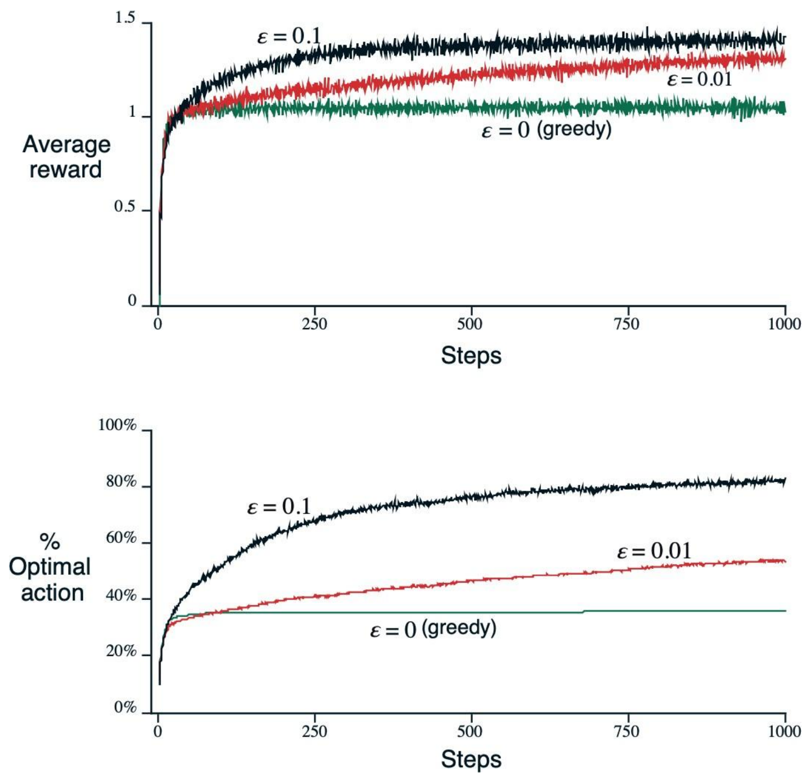


Figure 1: Average performance of g-greedy action-value methods on the 10-armed testbed. These data are averages over 2000 runs with different bandit problems. All methods used sample averages as their action-value estimates.

Incremental Implementation

The sample-average technique used to estimate action-values above has a problem: memory and computation requirements grow over time. This isn't necessary, we can devise an incremental solution instead:

$$\begin{aligned}
Q_{k+1} &= \frac{1}{k}\sum_{i}^{k} R_i \\
&= \frac{1}{k}\left(R_k + \sum_{i=1}^{k-1} R_i\right) \\
&= \vdots && (3) \\
&= Q_k + \frac{1}{k}\left[R_k - Q_k\right] && (4) \\
& && (5)
\end{aligned}$$

We are updating our estimate of Q_{k+1} by adding the discounted error between the rewardjust received and our estimate for that reward Q_k.

$$NewEstimate \leftarrow OldEstimate + StepSize\,[Target - OldEstimate] \qquad (6)$$

α is used to denote the stepsize (1/k) in the rest of this book.

Tracking a Nonstationary Problem

The averaging methods discussed above do not work if the bandit is changing over time. In such cases it makes sense to weight recent rewards higher than long-past ones. The popular way of doing this is by using a constant step-size parameter.

$$Q_{k+1} = Q_k + \alpha\,[R_k - Q_k] \qquad (7)$$

where the step-size parameter $\alpha \in (0, 1]$ is constant. This results in Q_{k+1} being a weightedaverage of the past rewards and the initial estimate Q_1:

$$\begin{aligned} Q_{k+1} &= Q_k + \alpha\,[R_k - Q_k] \\ &= \alpha R_k + (1-\alpha)Q_k \\ &= \alpha R_k + (1-\alpha)[\alpha R_{k-1} + (1-\alpha)Q_{k-1}] \\ &= \alpha R_k + (1-\alpha)\alpha R_{k-1} + (1-\alpha)^2 Q_{k-1} \\ &= \vdots \\ &= (1-\alpha)^k Q_1 + \sum_{i}^{k} \alpha(1-\alpha)^{k-i} R_i \qquad (8) \\ & \qquad (9) \end{aligned}$$

Because the weight given to each reward depends on how many rewards ago it was observed, we can see that more recent rewards are given more weight. Note the weights α sum to 1 here, ensuring it is indeed a weighted average where more weight is allocated to recent rewards.

In fact, the weight given to each reward decays exponentially into the past. This sometimes called an *exponential* or *recency-weighted* average.

Optimistic Initial Values

The methods discussed so far are dependent to some extent on the initial action-value estimate i.e. they are biased by their initial estimates.

For methods with constant α this bias is permanent.

In effect, the initial estimates become a set of parameters for the model that must be picked by the user.

In the above problem, by setting initial values to +5 rather than 0 we encourage exploration, even in the greedy case. The agent will almost always be disappointed with it's samples because they are less than the initial estimate and so will explore elsewhere until the values converge.

The above method of exploration is called *Optimistic Initial Values*.

Upper-confidence-bound Action Selection

ϵ-greedy action selection forces the agent to explore new actions, but it does so indiscriminately. It would be better to select among 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 uncertainty in those estimates. One way to do this is to select actions as:

$$A_t = \underset{a}{\operatorname{argmax}} \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right] \tag{10}$$

where c $>$ 0 controls the degree of exploration.

The square root term is a measure of the uncertainty in our estimate. It is proportional to t i.e. how many timesteps have passed and inversely proportional to $N_t(a)$ i.e. how many times that action has been visited. The more time has passed, and the less we have sampled an action, the higher our upper-confidence-bound.

As the timesteps increases, the denominator dominates the numerator as the ln term flattens.

Each time we select an action our uncertainty decreases because N is the denominator of this equation.

UCB will often perform better than e-greedy methods

Associative Search (Contextual Bandits)

Thus far we have been discussing the stationary *k*-armed bandit problem, where the value of each arm is unknown but nonetheless remains stationary. Now, we consider a problem where the task could change from step to step, but the value distributions of the arms in each task remain the same. This is called contextual bandits, and in the toy example we are usually given a hint that the task has changed e.g. the slot machine changes colour for each task. Now we want to learn the correct action to take in a particular setting, given the task colour observed. This is an intermediary between the stationary problem and the full reinforcement learning problem. See exercise below.

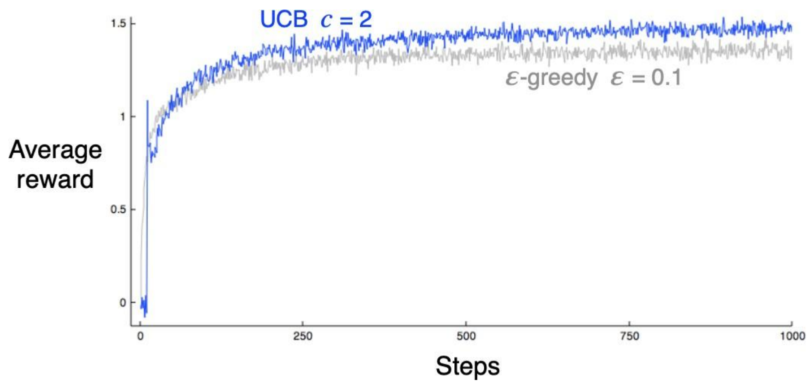


Figure 2: UCB performs better than e-greedy in the n-armed bandit problem

Summary

The value of an action can be summarized by $Q_t(a)$, the sample average return from an action

When selecting an action, it is preferable to maintain exploration, rather than only selecting the action we believe to be most valuable at any given timestep, to ensure we continue to improve our best estimate of the optimal action. We do so using ϵ)-greedy policies.

If our problem is non-stationary, rather than taking a standard average of every return received after an action, we can take a weighted average that gives higher value to more recently acquired rewards. We call this an *exponential* or *recency-weighted* average.

Optimistic initial values encourage lots of early exploration as our returns always decrease our estimate of Q_t meaning the greedy actions remain exploratory. Only useful for stationary problems.

ϵ-greedy policies can be adapted to give more value to actions that have been selected less- often, i.e. actions where we have higher uncertainty in their value, using *upper-confidence-bound* action selection.

Lastly, each of these techniques have varied performance on the n-armed bandit test de- pendent on their parametrisation. Their performance is plotted in Figure 3.

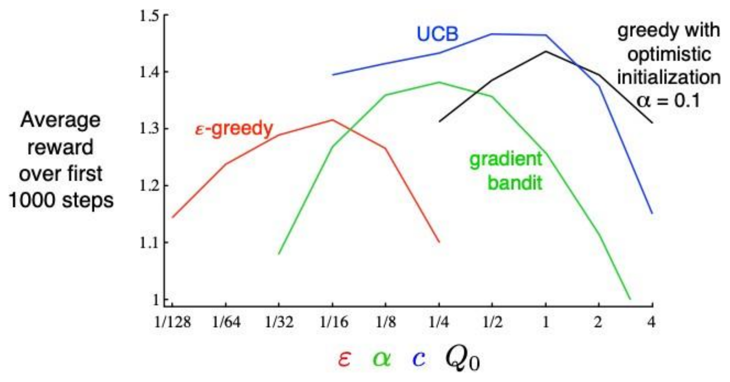


Figure 3: Performance of each of the bandit algorithms explored in this chapter

UNIT-II

Finite Markov Decision Processes

The Agent-Environment Interface

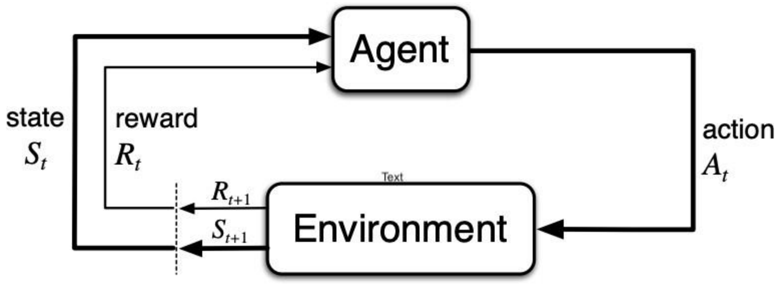


Figure 4: The agent-environment interface in reinforcement learning

At each timestep the agent implements a mapping from states to probabilities of selecting a possible action. The mapping is called the agents *policy*, denoted π, where $\pi(a\ s)$ is the probability of the agent selecting actions a in states.

In general, actions can be any decision we want to learn how to make, and states can be any interpretation of the world that might inform those actions.

The boundary between agent and environment is much closer to the agent than is first intuitive. E.g. if we are controlling a robot, the voltages or stresses in its structure are part of the environment, not the agent. Indeed reward signals are part of the environment,despite very possibly being produced by the agent e.g. dopamine.

Goals and Rewards

The *reward hypothesis*:

All we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward).

The reward signal is our way of communicating to the agent what we want to achieve nothow we want to achieve it.

Returns and Episodes

The return G_t is the sum of future rewards:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_t \tag{11}$$

This approach makes sense in applications that finish, or are periodic. That is, the agent- environment interaction breaks into *episodes*.

We call these systems *episodic tasks*. e.g playing a board game, trips through a maze etc.

Notation for state space in an episodic task varies from the conventional case ($s \in \mathrm{S}$)

to($s \in \mathrm{S}^+$)

The opposite, continuous applications are called *continuing tasks*.

For these tasks we use *discounted returns* to avoid a sum of returns going to infinity.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{12}$$

If the reward is a constant + 1 at each timestep, cumulative discounted reward G_t becomes:

$$G_t = \sum_{k=0}^{\infty} \gamma^k = \frac{1}{1-\gamma} \tag{13}$$

Discounting is a crucial topic in RL. It allows us to store a finite value of any state (summarized by its expected cumulative reward) for continuous tasks, where the non-discounted value would run to infinity.

Unified Notation for Episodic and Continuing Tasks

$$G_t = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1} \tag{14}$$

The Markov Property

A state signal that succeeds in retaining all relevant information about the past is *Markov*.Examples include:

A cannonball with known position, velocity and acceleration

All positions of chess pieces on a chess board.

In normal causal processes, we would think that our expectation of the state and reward atthe next timestep is a function of all previous states, rewards and actions, as follows:

$$Pr\{R_{t+1} = r, S_{t+1} = s'|S_0, A_0, R_1, \ldots, S_{t-1}, A_{t-1}, R_t, S_t, A_t\} \qquad (15)$$

If the state is Markov, however, then the state and reward right now completely characterizesthe history, and the above can be reduced to:

$$p(s', r|s, a) = Pr\{R_{r+1} = r, S_{t+1} = s'|S_t, A_t\} \qquad (16)$$

Even for non-Markov states, it is appropriate to think of all states as at least an approxi- mation of a Markov state.

Markov property is important in RL because decisions and values are assumed to be a function only of the current state.

Most real scenarios are unlikely to be Markov. In the example of controlling HVAC, the HVAC motor might heat up which affects cooling power and we may not be tracking that temperature. It is hard for a process to be Markov without sensing all possible variables.

Markov Decision Process (MDP)

Given any state and action s and a, the probability of each possible pair of next state andreward, s’, r is denoted:

p(s′, r|s, a) = Pr{Rr+1 = r, St+1 = s′|St, At} (17)

We can think of $p(s', r|s, a)$ as the dynamics of our MDP, often called the *transition function*–it defines how we move from state to state given actions.

Policies and Value Functions

Value functions are functions of states or functions of state-value pairs.

They estimate how good it is to be in a given state, or how good it is to perform a given action in a given state.

Given future rewards are dependent on future actions, value functions are defined with respect to particular policies as the value of a state depends on the action an agent takes in said state.

A *policy* is a mapping from states to probabilities of selecting each possible action.

RL methods specify how the agent's policy changes as a result of its experience.

For MDPs, we can define $v(\pi(s))$ formally as:

$$v_\pi(s) = \mathbb{E}_\pi \left[G_t | S_t = s\right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s\right] \tag{18}$$

i.e. the expected future rewards, given state S_t, and policy π. We call $v_\pi(s)$the *state value function for policy* π. Similarly, we can define the value of taking action *a* in state *s* under policy π as:

$$q_\pi(s, a) = \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a\right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a\right] \tag{19}$$

i.e. the expected value, taking action a in state s then following policy π.

We call q_π the *action-value function for policy* π

Both value functions are estimated from experience.

A fundamental property of value functions used throughout reinforcement learning and dynamic programming is that they satisfy recursive relationships similar to that which we have already established for the return. This recursive relationship is characterised by the *Bellman Equation*:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right] \tag{20}$$

This recursion looks from one state through to all possible next states given our policy andthe dynamics as suggested by 5:

Optimal Policies and Value Functions

A policy π' is defined as better than policy pi if its expected return is higher for all states.

There is always *at least* one policy that is better than or equal to all other policies - thisis the *optimal policy*.

Optimal policies are denoted π_*

Optimal state-value functions are denoted v_*

Optimal action-value functions are denoted q_*

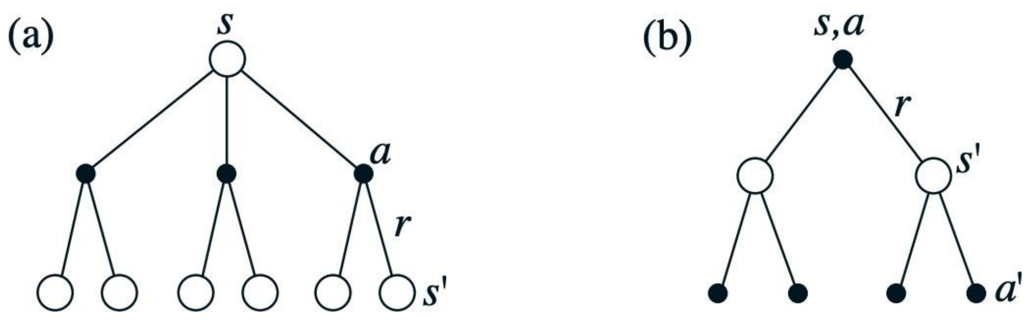


Figure 5: Backup diagrams for $v\pi$ and $q\pi$

We can write q_* in terms of v_*:

$$q_*(s, a) = \mathrm{E}\left[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a\right] \tag{21}$$

We can adapt the Bellman equation to achieve the Bellman optimality equation, which takestwo forms. Firstly for v_*:

$$v_*(s) = \max_{a \in \mathcal{A}(s)} \sum_{s', r} p(s', r|s, a) \left[r + \gamma v_*(s')\right] \tag{22}$$

and secondly for q_*:

$$q_*(s) = \sum_{s', r} p(s', r|s, a) \left[r + \gamma \max_{a'} q_*(s', a')\right] \tag{23}$$

Using v_* the optimal expected long term return is turned into a quantity that is immediately available for each state. Hence a one-step-ahead search, acting greedily, yield the optimal long-term actions.

Fully solving the Bellman optimality equations can be hugely expensive, especially if the number of states is huge, as is the case with most interesting problems.

Solving the Bellman optimality equation is akin to exhaustive search. We play out *every* possible scenario until the terminal state and collect their expected reward. Our policy then defines the action that maximises this expected reward.

In the continuous case the Bellman optimality equation is unsolvable as the recursion on the next state's value function would never end.

Optimality and Approximation

We must approximate because calculation of optimality is too expensive.

A nice way of doing this is allowing the agent to make sub-optimal decisions in scenariosit has low probability of encountering. This is a trade off for being optimal in situationsthat occur frequently.

Summary

We summarize our goal for the agent as a *reward* ; its objective is to maximise the cumulative sum of future rewards

For episodic tasks, returns terminate (and are backpropogated) when the episode ends. For the continuous control case, returns are discounted so they do not run to infinity.

A state signal that succeeds in retaining all relevant information about the past is *Markov*.

Markov Decision Processes (MDPs) are the mathematically idealised version of the RL problem. They have system dynamics:

$p(s', r|s, a) = Pr\{R_{r+1} = r, S_{t+1} = s'|S_t, A_t\}$

Policies are a (probabilistic) mapping from states to actions.

Value functions estimate how good it is for an agent to be in a state (v_π) or to take an action from a state (q_π). They are always defined w.r.t policies as the value of a state depends on the policy one takes in that state. Value functions are the *expected cumulative sum of future rewards* from a state or state-action pair.

Knowing our policy and system dynamics, we can define the state value function is defined by the Bellman equation:

$$v_\pi(s) \doteq \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right]$$

An optimal policy (π_*) is the policy that maximizes expected cumulative reward from all states. From the optimal policy we can derive the optimal value functions q_* and v_*.

Dynamic Programming

Dynamic Programming (DP) refers to the collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as an MDP. DP can rarely be used in practice because of their great cost, but are nonetheless important theoretically as all other approaches to computing the value function are, in effect, approximations of DP. DP algorithms are obtained by turning the Bellman equations into assignments, that is, into update rules for improving approximations of the desired value functions.

Policy Evaluation (Prediction)

We know from Chapter 3 that the value function can be represented as follows:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_\pi(s')\right] \tag{24}$$

If the dynamics are known perfectly, this becomes a system of |S| simultaneous linear equations in |S| unknowns, where the unknowns are $v_\pi(s)$, $s \in$ S. If we consider an iterative sequence of value function approximations v_0, v_1, v_2, . . ., with initial approximation v_0 chosen arbitrarily
e.g. $v_k(s) = 0 \ \forall s$ (ensuring terminal state = 0). We can update it using the Bellman equation:

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_k(s')\right] \tag{25}$$

Eventually this update will converge when $v_k = v_\pi$ after infinite sweeps of the state-space, the value function for our policy. This algorithm is called *iterative policy evaluation*. We call this update an *expected update* because it is based on the expectation over all possible next states, rather than a sample of reward/value from the next state. We think of the updates occurring through *sweeps* of state space.

Policy Improvement

We can obtain a value function for an arbitrary policy π as per the policy evaluation algorithm discussed above. We may then want to know if there is a policy π' that is better than our current policy. A way of evaluating this is by taking a new action a in state s that is not in our current policy, running our policy thereafter and seeing how the value function changes. Formally that looks like:

$$q_\pi(s,a) = \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_\pi(s')\right] \tag{26}$$

Note the mixing of action-value and state-value functions. If taking this new action in state s produces a value function that is greater than or equal to the previous value function for all states then we say the policy $\pi^{'}$ is an improvement over π:

$$v_{\pi}'(s) \geq v_{\pi} \forall s \in \mathrm{S} \tag{27}$$

This is known as the *policy improvement theorem*. Critically, the value function must be greater than the previous value function for all states. One way of choosing new actions for policy improvement is by acting greedily w.r.t the value function. Acting greedily will always produce a new policy $\pi^{'} \geq \pi$, but it is not necessarily the optimal policy immediately.

Policy Iteration

By flipping between policy evaluation and improvement we can achieve a sequence of monoton-ically increasing policies and value functions. The algorithm is roughly:

1. Evaluate policy π to obtain value function V_{π}
2. Improve policy π by acting greedily with respect to V_{π} to obtain new policy $\pi^{'}$
3. Evaluate new policy $\pi^{'}$ to obtain new value function V_{π}'
4. Repeat until new policy is no longer better than the old policy, at which point we haveobtained the optimal policy. (Only for finite MDPs)

This process can be illustrated as:

$$\pi_0 \xrightarrow{\mathrm{E}} v_{\pi_0} \xrightarrow{\mathrm{I}} \pi_1 \xrightarrow{\mathrm{E}} v_{\pi_1} \xrightarrow{\mathrm{I}} \pi_2 \xrightarrow{\mathrm{E}} \cdots \xrightarrow{\mathrm{I}} \pi_* \xrightarrow{\mathrm{E}} v_*,$$

Figure 6: Iterative policy evaluation and improvement

Value Iteration

Above, we discussed policy iteration which requires full policy evaluation at each iteration step, an often expensive process which (formally) requires infinite sweeps of

$\in \mathrm{S}$

the state space to approach the true value function. In value iteration, the policy evaluation is stopped after one visit to each *s* , or one *sweep* of the state space. Value iteration is achieved by turning the Bellman optimality equation into an update rule:

$$v_{k+1}(s) = \underset{a}{\operatorname{argmax}} \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_k(s')\right] \tag{28}$$

for all *s* . Value iteration effectively combines, in each of its sweeps, one sweep of policy evaluation and one sweep of policy improvement.

Asynchronous Dynamic Programming

Each of the above methods has required full sweeps of the state space to first evaluate a policy then improve it. Asynchronous dynamic programming does not require us to evaluate the entire state space each sweep. Instead, we can perform in place updates to our value function as they are experienced, focusing only on the most relevant states to our problem initially, then working to less relevant states elsewhere. This can mean that our agent can learn to act well more quickly and save optimality for later.

Generalised Policy Iteration

Generalised Policy Iteration is the process of letting policy evaluation and policy improvement interact, independent of granularity. That is to say, improvement/evaluation can be performed by doing complete sweeps of the state space, or it can be performed after every visit to a state (as is the case with value iteration). The level of granularity is independent of our final outcome: convergence to the optimal policy and optimal value function. This process can be illustrated as two convergening lines - Figure 7. We can see that policy improvement and policy evaluation work both in opposition and in cooperation - each time we act greedily we get further away from our true value function; and each time we evaluate our value function our policy is likely no longer greedy.

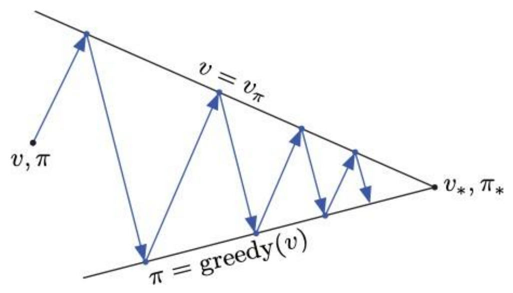


Figure 7: Generalised policy iteration leading to optimality

Summary

Policy evaluation is the iterative computation of a value function for a given policy. The value function is only truly evaluated in the limit.

Policy improvement is the computation of an improved policy given the value function for an existing policy.

By combining the above we achieve *Policy Iteration* and by performing each directly on the value function at each visit to a state we achieve *Value Iteration* - both of which can be used to obtain optimal value functions and policies given complete knowledge of a finite Markov Decision Process.

Generalised Policy Iteration is the process of performing policy evaluation and improve- ment regardless of granularity - leading to convergence to the optimal value functions.

DP methods operate in sweeps through the state space performing an *expected update* operation at each state. These updates are based on expected values at all subsequent states and their probabilities of occurring. In general this idea is known as *bootstrapping*, many RL methods bootstrap to update current beliefs from past beliefs.

UNIT-III

Monte Carlo Methods

If we do not have knowledge of the transition probabilities (model of the environment) then we must learn directly from experience. To do so, we use Monte Carlo methods. Monte carlo methods are most effective in episodic tasks where there is a terminal state and the value of the states visited en route to the terminal state can be updated based on the reward received at the terminal state. We use general policy iteration as outlined in chapter 4, but this time instead of *computing* the value function we *learn* it from samples. We first consider the prediction problem to obtain v_π and/or q_π for a fixed policy, then we look to improve using policy improvement, then we use it for control.

Monte Carlo Policy Prediction

Recall that the value of a state is the expected discounted future reward from that state. One way of estimating that value is by observing the rewards obtained after visiting the state, we would expect that in the limit this would converge toward the true value.

We can therefore run a policy in an environment for an episode. When the episode ends, we receive a reward and we assign that reward to each of the states visited en route to the terminal state.

Where DP algorithms perform one-step predictions to *every* possible next state; monte-carlo methods only sample one trajectory/episode. This can be summarised in a new backup diagram as follows:

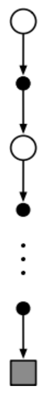


Figure 8: Monte carlo backup diagram for one episode

Importantly, monte carlo methods do not bootstrap in the same way DP methods do. They take the reward at the end of an episode, rather than estimated reward based onthe value of the next state.

Because of the lack of bootstrapping, this expense of estimating the value of one stateis independent of the number of states, unlike DP. A significant advantage, in additionto the other advantages of being able to learn from experience without a model or from simulated experience.

Monte Carlo Estimation of Action Values

With a model we only need to estimate the state value function *v* as, paired with our model, we can evaluate the rewards and next states for each of our actions and pick the best one.

With model free methods we need to estimate the state-action value function *q* as we must explicitly estimate the value of each action in order for the values to be useful in suggesting a policy. (If we only have the values of states, and don't know how states are linked through a model, then selecting the optimal action is impossible)

One serious complication arises when we do not visit every state, as can be the case if our policy is deterministic. If we do not visit states then we do not observe returns from these states and cannot estimate their value. We therefore need to *maintain* *exploration* of the state space. One way of doing so is stochastically selected a state-action pair to start an episode, giving every state-action pair a non-zero probability of being selected. In this case, we are said to be utilizing *exploring starts*.

Exploring starts falls down when learning from real experience because we cannot guarantee that we start in a new state-action pair in the real world.

An alternative approach is to use stochastic policies that have non-zero probability of selecting each state-action pair.

Monte Carlo Control

Much like we did with value iteration, we do not need to fully evaluate the value function for a given policy in monte carlo control. Instead we can merely *move* the value toward the correct value and then switch to policy improvement thereafter. It is natural to do this episodically i.e. evaluate the policy using one episode of experience, then act greedily w.r.t the previous value function to improve the policy in the next episode.

If we use a deterministic policy for control, we must use exploring starts to ensure sufficientexploration. This creates the *Monte Carlo ES* algorithm.

Monte Carlo Control without Exploring Starts

To avoid having to use exploring starts we can use either *on-policy* or *off-policy* methods. The only way to ensure we visit everything is to visit them directly.

On-policy methods attempt to improve or evaluate or improve the policy that is making decisions.

On-policy control methods are generally *soft* meaning that they assign non-zero probabilityto each possible action in a state e.g. e-greedy policies.

We take actions in the environment using e-greedy policy, after each episode we back propagate the rewards to obtain the value function for our e-greedy policy. Then we perform policy improvement by updating our policy to take the **new** greedy reward ineach state. Note: based on our new value function, the new greedy action may have changed in some states. Then we perform policy evaluation using our new e-greedy policy and repeat (as per generalised policy iteration).

The idea of on-policy Monte Carlo control is still that of GPI. We use first visit MC methods to estimate the action-value function i.e. to do policy evaluation, but we cannot then make improve our policy merely by acting greedily w.r.t our value function because that would prevent further exploration of non-greedy actions. We must maintain exploration and so improve the ϵ-greedy version of our policy. That is to say, when we find the greedy action (the action that maximises our reward for our given value function) we assign it probability $1 \ \epsilon + \frac{\epsilon}{\mathcal{A}(S_t)}$ of being selected so that the policy remains stochastic.

Note: doing the above will only find us the best policy amongst the ϵ-soft policies, which may not be the optimal policy for the environment, but it does allow us to remove the need for exploratory starts.

Off-policy Prediction via Importance Sampling

We face a dilemma when learning control: we want to find the optimal policy, but we can only find the optimal policy by acting suboptimally to explore sufficient actions. What we saw with on-policy learning was a compromise - it learns action values not for the optimal policy but for a near-optimal policy that still explores. An alternative is off-policy control where we have two policies: one used to generate the data (behaviour policy) and one that is learned for control (target policy). This is called *off policy learning*.

Off-policy learning methods are powerful and more general than on-policy methods (on- policy methods being a special case of off-policy where target and behaviour policies are the same). They can be used to learn from data generated by a conventional non-learning controller or from a human expert.

If we consider an example of finding target policy π using episodes collected through a behaviour policy b, we require that every action in π must also be taken, at least occasionally, by b i.e. $\pi(a\ s) > 0$ implies $b(a\ s) > 0$. This is called the assumption of ***coverage***.

Almost all off-policy methods utilize *importance sampling*, a general technique for estimating expected values under one distribution given samples from another. Given a starting state S_t, the probability of the subsequent state-action trajectory occuring under anypolicy π is:

$$\prod_{k=t}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k, A_k) \tag{29}$$

where p is the state transition probability function. We can then obtain the relative probability of the trajectory under the target and behaviour policies (the importance sampling ratio) as:

$$p_{t:T-1} = \frac{\prod_{k=t}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k, A_k)}{\prod_{k=t}^{T-1} b(A_k|S_k)p(S_{k+1}|S_k, A_k)} = \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)} \tag{30}$$

We see here that the state transition probability function helpfully cancels.

We want to estimate the expected returns under the target policy but we only have returnsfrom the behaviour policy. To address this we simply multiply expected returns from the behaviour policy by the importance sampling ratio to get the value function for our target policy.

$$\mathrm{E}\left[p_{t:T-1}G_t|S_t = s\right] = v_\pi(s) \tag{31}$$

Note: importance sampling ratios are only non-zero for episodes where the target-policy has non-zero probability of acting *exactly* like the behaviour policy b. So, if the behaviour policy takes 10 steps in an episode, each of these 10 steps have to have been *possible* by the target policy, else $\pi(a|s) = 0$ and $\rho_{t:T-1} = 0$.

Incremental Implementation

We perform monte carlo policy evaluation (prediction) incrementally in the same way as was done in Chapter 2 for the bandit problem. Generally incremental implementation follows this formula:

$$NewEstimate \leftarrow OldEstimate + StepSize\ [Observation - OldEstimate] \tag{32}$$

With on-policy monte carlo methods, this update is performed exactly after each episode for each visit to a state given the observed rewards, with off-policy methods the update is slightly more complex. With ordinary importance sampling, the step size is $1/n$ where n is the number of visits to that state, and so acts as an average of the scaled returns. For weighted importance sampling, we have to form a weighted average of the returns which requires us to keep track of the weights. If the weight takes the form $W_i = \rho_{t:T(t)-1}$ then our update rule is:

$$V_{n+1} = V_n + \frac{W_n}{C_n}\left[G_n - V_n\right] \tag{33}$$

where,

$$C_{n+1} = C_n + W_{n+1} \tag{34}$$

with $C_0 = 0$. This allows us to keep tracking of the corrected weighted average term for each update as they are made. Note here that the weighted average gives more weight to updates based on common trajectories from b in π that we have some more often.

Off-policy Monte Carlo Control

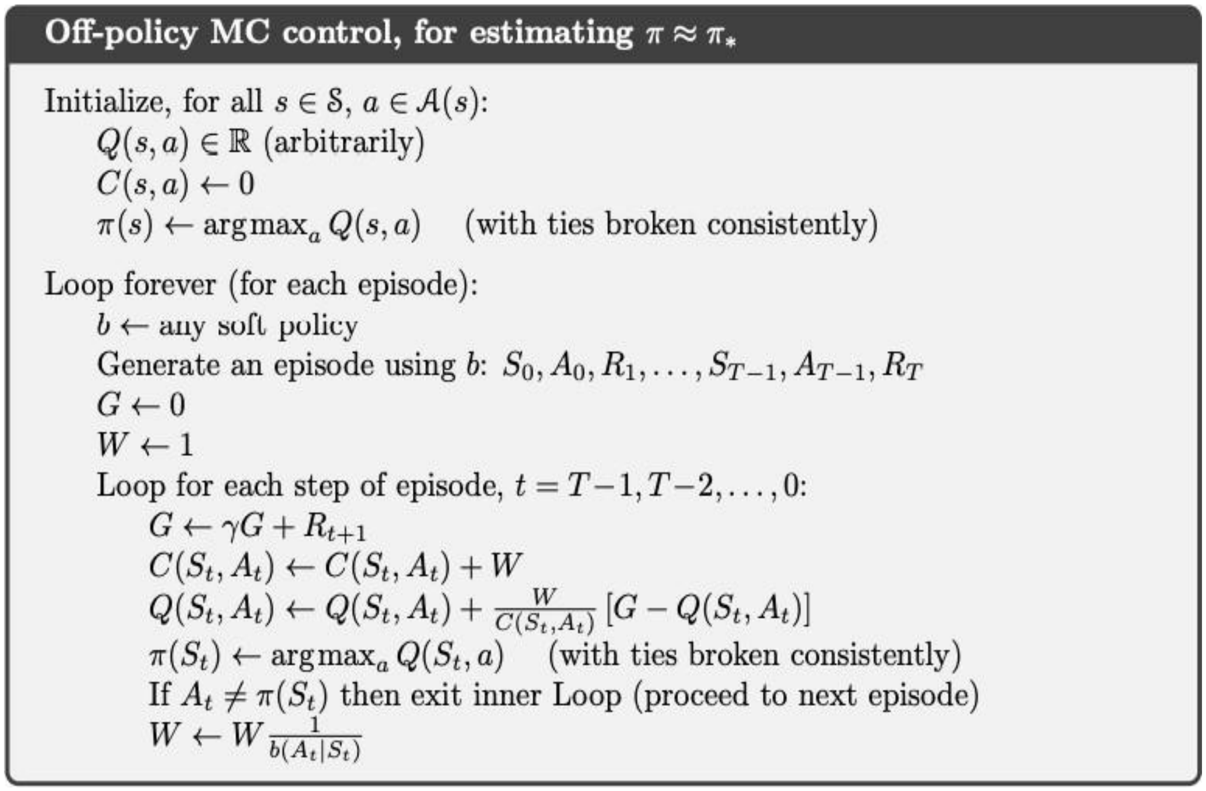
Using incremental implementation (updates to the value function) and importance sampling we can now discuss *off-policy monte carlo control* –the algorithm for obtaining optimal policy π_* by using rewards obtained through behaviour policy b. This works in much the same way as in previous sections; b must be ϵ-soft to ensure the entire state space is explored in the limit; updates are only made to our estimate for q_π, Q, if the sequence of states an actions produced by b could have been produced by π. This algorithm is also based on GPI: we update our estimate of Q using Equation 33, then update π by acting greedily w.r.t to our value function. If this policy improvement changes our policy such that the trajectory we are in from b no longer obeys our policy, then we exit the episode and start again. The full algorithm is shown in 9.

Summary

In the absence of a model of the environment, monte carlo methods allow us to evaluate and improve our value function based on *experience*

We roll-out trajectories to terminal states, and back-propagate the rewards to the states visited en-route in several ways

Monte carlo methods use GPI (see chapter 4) in much the same way dynamic programming does. We evaluate our policy, then improve by acting greedily w.r.t our new value function until we converge on the optimal value function and policy.



Off-policy MC control, for estimating $\pi \approx \pi_*$

```
Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
    $Q(s,a) \in \mathbb{R}$ (arbitrarily)
    $C(s,a) \leftarrow 0$
    $\pi(s) \leftarrow \operatorname{argmax}_a Q(s,a)$    (with ties broken consistently)

Loop forever (for each episode):
    $b \leftarrow$ any soft policy
    Generate an episode using $b$: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    $W \leftarrow 1$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        $C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$
        $\pi(S_t) \leftarrow \operatorname{argmax}_a Q(S_t, a)$    (with ties broken consistently)
        If $A_t \neq \pi(S_t)$ then exit inner Loop (proceed to next episode)
        $W \leftarrow W \frac{1}{b(A_t|S_t)}$
```

Figure 9: Off-policy monte carlo control

Differences with dynamic programming methods: 1) They do not require a model of the environment as they learn directly from experience, and 2) They do not bootstrap i.e. the value function estimates are an average of all real returns accumulated after visiting the state.

Maintaining sufficient exploration is crucial with monte carlo methods; if our policy is deterministic we will likely not explore the full states space during our roll-outs. To deal with this we have several options: 1) Start every episode randomly with uniform probability such that we make sure we start at each possible state–called *exploring starts*, unrealistic in the real world as we can't make a robot restart from all possible states.Or 2) Use ϵ-soft policies that have a non-zero probability of selecting all possible states.The downside of doing this is that we will converge on the optimal ϵ-soft, which is not necessarily the optimal policy for the environment, because it needs to learn how account for its own randomness. This is the price we pay for exploration.

Monte carlo methods can either be *on-policy* or *off-policy*.

On-policy methods use the same policy to collect data as is evaluated and improved. This suffers the downsides outlined above.

Off-policy methods have two policies, one that collects the data called the *behaviour policy* b and the other which we look to improve called the target policy π. We find trajectories from the behaviour policy that line up with our target policy, that is to say, that couldhave been produced by our target policy. This process only works if the behaviour policy has non-zero probability of selecting each of the actions in the target policy, aka *coverage* of the target policy. The agent explores, but learns a deterministic optimal policy offline that may be unrelated to the behaviour policy used to collect the experience.

Based on rewards observed by running the behaviour policy, we update our value functionusing *importance sampling,* which measures, if effect, how likely the observed behaviour would have been given our target policy. For example, the target policy may take 1 of 4 actions with equal probability in each state. If we observe two timesteps from our beaviour policy, then our probability of selecting the actions taken by the behaviour policy would be 0.25×0.25.

We weight each return using the *importance sampling ratio*, a measure of how likely we were to produce the roll-out using the target policy compared to how likely we were to produce the roll-out using the behaviour policy.

Importance sampling can be *ordinary* i.e. an average of the returns observed from a state or *weighted* where trajectories viewed more often, or with a higher importance sampling ratio give the value update more weight.

UNIT-IV

Temporal-Difference Learning

TD learning is novel to RL. It is a hybrid that lies between monte carlo and dynamic programming methods. As with those, TD learning uses GPI; control (policy improvement) works in much the same way, it is in the prediction of the value function (policy evaluation) where the distinctions lie.

TD Prediction

Monte carlo methods wait until the end of the episode before back-propagating the return to the states visited en-route. The simplest TD method (TD(0)) does not wait until the end of the episode, in fact, it updates its estimate of the value function $V(s)$ based on the instantaneous reward as follows:

$$V(S_t) \leftarrow V(S_t) + \alpha\,[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{35}$$

Note here, that TD(0) *bootstraps* in much the same way DP methods do i.e. it bases it's estimate of $V(s)$ on the value of the next state $V(s')$. TD methods therefore take both sampling from monte carlo, and bootstrapping from DP. The TD(0) backup diagram is shown in Figure 10.

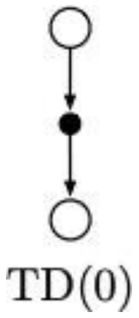


Figure 10: Backup diagram for TD(0)

The quantity in the brackets of equation 35 is known as the *TD error*. Formally this is expressed as:

$$\delta_t = \boldsymbol{R}_{t+1} + \gamma V(S_{t+1}) - V(S_t) \tag{36}$$

In Figure 11 we see the differences between monte carlo and TD updates to the value function. In the case of monte carlo, we would need to wait until we arrived home before we could update our predictions for journey length. With TD learning, we can adjust these estimates on the fly based on the error in our predictions en-route. We see here that the adjustment we make changes at each timestep depending on the difference between predictions, that is to say, the update is proportional to the *temporal differences* between our predictions across timesteps.

Advantages of TD Prediction Methods

TD methods generally converge faster than MC methods, although this has not been formally proven.

TD methods do converge on the value function with a sufficiently small step size parameter, or with a decreasing stepsize.

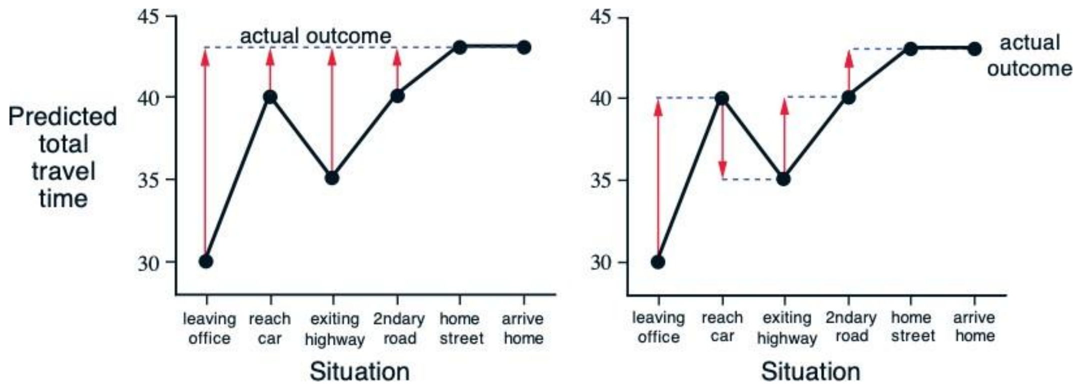


Figure 11: Monte carlo updates (left) and TD-updates (right) to the value function associated with a daily commute

They are extremely useful for continuing tasks that cannot be broken down in episodes as required by MC methods.

Optimality of TD(0)

If we only have a finite number of episodes or training data to learn from we can update our value function in *batches*. We repeatedly play the same data, and update our value function by the sum of each of the increments for each state at the end of the batch.

TD batching and Monte Carlo batching converge often on two different estimates of the value function. The monte carlo batch estimate can only judge based on observed rewards from a state, but TD batching can make estimates based on later states using bootstrapping. Example (*You are the predictor*) on pg 127 explains this excellently. The monte carlo estimate will converge on the correct estimate of the value function produced by the training data, but TD methods will generalise better to future rewards because they preserve the transition from state- to-state in the TD update, and thus bootstrap better from values of other states.

In general batch MC methods always minimise the mean squared error or the training set whereas TD(0) finds the estimates that would be exactly correct for the maximumlikelihood model of the Markov process. This is called the *certainty equivalence estimate* because it is equivalent to assuming that the estimate of the underlying process was known with certainty rather than being approximated.

Sarsa: On-policy TD Control

We'll now look at using TD prediction methods for control. We'll follow, as before, the framework of Generalised Policy Iteration (GPI), only this time using TD methods for predicting the value function. For on-policy control, we wish to learn the state-action value function for our policy $q_\pi(s, a)$ and all states s and actions a. We amend the TD update formula provided in Equation 35 to account for state-action values:

$$Q(S_t, A_t) \longleftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right] \tag{37}$$

This produces a quintuple of events: $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ giving rise to the name SARSA. As with other on-policy control algorithms, we update our policy to be greedy w.r.t our ever-changing value function.

Q-learning: Off-policy TD Control

Q-learning, much like SARSA, makes 1-step updates to the value function, but does so in subtly different ways. SARSA is on-policy, meaning it learns the optimal version of the policy used for collecting data i.e. an ϵ-greedy policy to ensure the state-space is covered. This limits its performance as it needs to account for the stochasticity of exploration with probability ϵ. Q- learning, on the other hand, is off-policy and directly predicts the optimal value function q_* whilst using an ϵ-greedy policy for exploration. Updates to the state-action value function are performed as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right] \tag{38}$$

We observe that the max action selection at next state S_{t+1} means we are no longer following an ϵ-greedy policy for our updates. Q-learning has been shown to converge to q_* with probability 1.

Expected Sarsa

Instead of updating our value function with the value maximising action at S_{t+1} (as is thecase with Q-learning) or with the action prescribes by our ϵ-greedy policy (as is the case with SARSA), we could make updates based on the *expected value* of Q at S_{t+1}. This is the premise of expected sarsa. Doing so reduces the variance induced by selecting random actions according to an ϵ-greedy policy. It's update is described by:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \mathbb{E}_\pi Q(S_{t+1}, A_{t+1}|S_{t+1}) - Q(S_t, A_t)\right] \tag{39}$$

$$\leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t)\right] \tag{40}$$

$$\tag{41}$$

We can adapt expected SARSA to be off-policy by making our target policy π greedy, in which case expected SARSA becomes Q-learning. It is therefore seen as a generalisation of Q-learning that reliably improves over SARSA.

Maximization Bias and Double Learning

Many of the algorithms discussed so far in these notes use a maximisation operator to select actions - either ϵ-greedy or greedy action selection. Doing so means we implicitly favour positive numbers. If the true values of state action pairs are all zero i.e. $Q(S, A) = 0$ s, a, then, at some stage in learning, we are likely to have distributed value estimates around 0. Our maximisation operator will select the postive value estimates, despite the latent values being 0. Thus we bias positive values, a so-called *maximisation bias*.

We can counter this by learning two estimates of the value function Q_1 and Q_2. We use one to select actions from a state e.g. $A^* = \text{argmax}_a\ Q_1(a)$ and then use the other to provide an estimate of its value $Q_2(A^*) = Q_2(\text{argmax}_a\ Q_1(a))$. This estimate will be unbiased in the sense that $\mathrm{E}\left[Q_2(A^*)\right] = q(A^*)$.

Games, Afterstates, and Other Special Cases

Much of the theory in this book revolves around action-value functions, where the value of an action in a given state is quantified **before** the action is taken. In some settings, it is expedient to quantify value functions after an action, as many actions may lead to the same state afterward. This is the case for the game tic-tac-toe, as described in the book. In certain circumstances, it can be beneficial to amend value functions to accommodate afterstates if this property is shown.

Summary

TD methods update value functions en-route to episode termination, rather than waiting to the end of the episode like Monte Carlo methods. Updates to the value function are made in proportion to the *temporal differences* between value function estimates.

TD methods bootstrap off of value function estimates elsewhere in the state space, and consequently generalise to new data better than MC methods, which cannot make predic- tions outside of the observed training data.

TD methods continue to follow the framework of Generalised Policy Iteration (GPI) as discussed in previous chapters.

SARSA is the eminent on-policy TD method, but is limited in that in can only ever learn the optimal ϵ-soft behaviour policy.

Q-learning is the eminent off-policy TD method, that will learn the optimal policy π_* with probability 1.

Expected SARSA makes updates based on the expected value of the next state given a policy, reducing the variance induced by an ϵ-soft behaviour policy, performing better than SARSA given the same experience.

The postive bias of action selection via maximisation can be mitigated by double learning methods.

n-step Bootstrapping

In the previous chapter we discussed one-step TD prediction, and in the chapter previous to that we discussed monte carlo methods where predictions are made based on returns at the end of the episode. In this chapter we will discuss an approach between these two, where predictions are made after *n*-steps in the future. Doing so is often more effective than either of the two previously presented approaches.

n-step TD Prediction

The spectrum of *n*-step TD methods is summarised by Figure 12. Recall from the previous chapter that our one-step return used for TD(0) was:

$$G_{t:t+1} = R_{t+1} + \gamma V_t(S_{t+1}) \tag{42}$$

we can generalise this to the *n*-step case as follows:

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n}) \tag{43}$$

for all n, t such that $n \geq 1$ and $0 \leq t \leq T - n$. All *n*-step returns can be considered approxi- mations to the full return, truncated after n steps and then corrected for the remaining missing terms by $V_{t+n-1}(S_{t+n})$. If the n-step return extends to or beyond termination then all the missing terms are taken as zero.

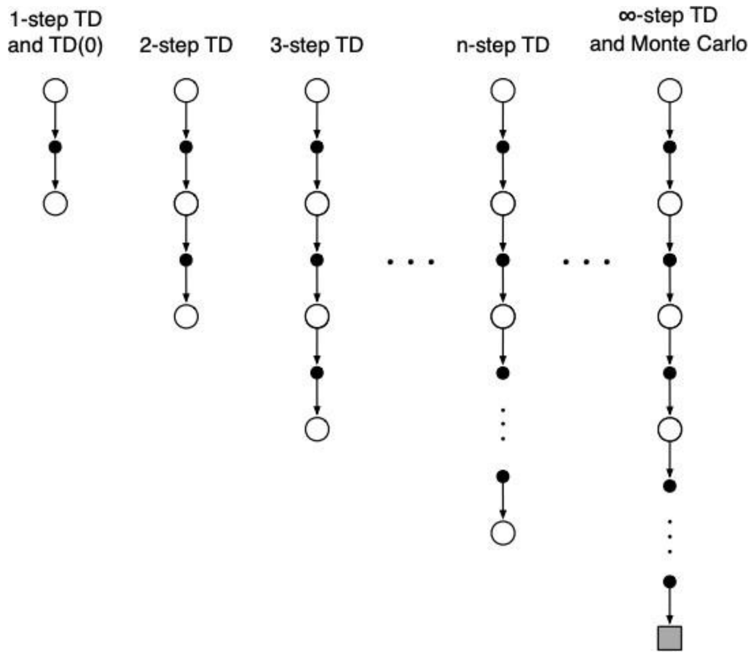


Figure 12: Backup diagram for TD(0)

The n-step return uses the value function V_{t+n-1} to correct for missing rewards beyond R_{t+n}. An important property of n-step returns is that their expectation is guaranteed to be a better estimate of v_π that V_{t+n-1} in a worst-state sense i.e.:

$$\max_s \left| \mathbb{E}_\pi \left[G_{t:t+n} | S_t = s \right] - v_\pi(s) \right| \leq \gamma^n \max_s \left| V_{t+n-1}(s) - v_\pi(s) \right| \tag{44}$$

This is called the error reduction property of n-step returns. Because of this, one can show formally that all n-step TD methods converge to t he correct predictions under appropriate conditions.

n-step Sarsa

We can extend n-step value function prediction to a control algorithm by incorporating the ideas of the Sarsa. Now, instead of making value function updates based on rewards received to the $t + n^{th}$ state, we select update our state-action value function Q by making updates up to the $t + n^{th}$ state-action pair. The family of n-step Sarsa backup diagrams is shown in Figure 13. The *n*-step Sarsa update is:

$$Q_{t+n}(S_t, A_t) = Q_{t+n-1}(S_t, A_t) + \alpha \left[G_{t:t+n} - Q_{t+n-1}(S_t, A_t)\right] \tag{45}$$

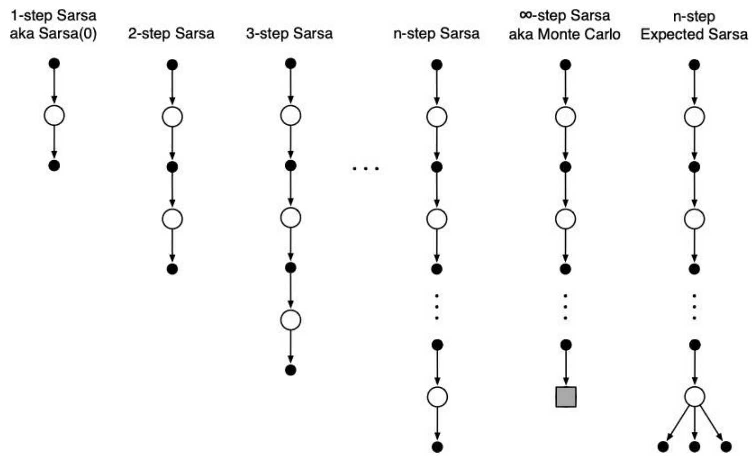


Figure 13: Backup diagrams for family of n-step Sarsa algorithms

An example of why *n*-step sarsa speeds up policy learning is given in Figure 14 Expected Sarsa remains an important version of Sarsa because of the variance-reducing expectation. The algorithm is described as in equation 45, but this time the expected return

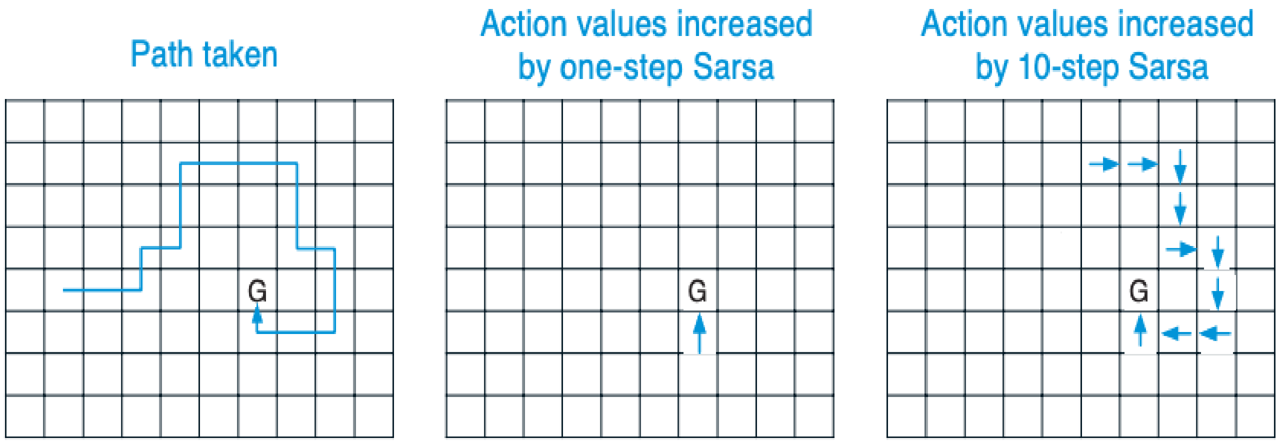


Figure 14: Gridworld example of the speedup of policy learning due to the use of n-step methods. The first panel shows the path taken by an agent in a single episode, ending at a location of high reward, marked by the G. In this example the values were all initially 0, and all rewards were zero except for a positive reward at G. The arrows in the other two panels show which action values were strengthened as a result of this path by one-step and n-step Sarsa methods. The one-step method strengthens only the last action of the sequence of actions that led to the high reward, whereas the n-step method strengthens the last n actions of the sequence, so that much more is learned from the one episode.

includes the expected value of the final value function, i.e.:

$$G_{t:t+n} = R_{t+1} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \bar{V}_{t+n-1}(S_{t+n}) \tag{46}$$

where $\bar{V}_{t+n-1}(s)$ is the *expected approximate value* of state *s*, using the estimated action values at time *t*, under the target policy:

$$\bar{V}_t(s) \doteq \sum_a \pi(a|s) Q_t(s,a), \quad \text{for all } s \in \mathcal{S} \tag{47}$$

Expected approximate values are used in developing many of the action-value method dis-cussed hereafter.

n-step Off-policy Learning

We can extend *n*-step learning to the off-policy case where we have a behaviour policy *b* that creates the data and a target policy π that we wish to update. As previously discussed, when we make updates in this way, we must weight the updates using the importance sampling ratio- equation 30. We can create a generalised form of the *n*-step algorithm, that works for both on-policy and off-policy cases:

$$Q_{t+n}(S_t, A_t) = Q_{t+n-1}(S_t, A_t) + \alpha \rho_{t+1:t+n} \left[G_{t:t+n} - Q_{t+n-1}(S_t, A_t) \right] \tag{48}$$

Per-decision Methods with Control Variates

The off-policy methods described above may not learn as efficiently as they could. If we consider the case where the behaviour policy does not match the target policy for one of the *n* steps used in the update, the update will go to 0 and the value function estimate will not change. Instead we can use a more sophisticated approach, that has a different definiton of the *n*-step horizon:

$$G_{t:h} \doteq \rho_t(R_{t+1} + \gamma G_{t+1:h}) + (1-\rho_t)V_{h-1}(S_t) \tag{49}$$

.

In this approach, if ρ_t is zero, then instead of the target being zero and causing the estimate to shrink, the target is the same as the estimate and causes no change. Because the expected value of ρ_t is 1, the expected value of the second term in equation 49, called the *control variate* has expected value of 0, and so does not change the update in expectation.

Off-policy Learning Without Importance Sampling: The *n*-step Tree Backup Algorithm

Here we will discuss off-policy learning without the need for importance sampling, conceptualised by the *n*-step Tree Backup Algorithm–Figure 15. In this tree backup update, the target now includes all rewards *plus* the estimated values of the dangling action nodes hanging off the sides, at all levels. It is an update from the *entire tree* of estimated action values. For each node in the tree backup diagram, we the estimated values of the non-selected actions are weighted by their probability of being selected under our policy $\pi(A_t\ S_t)$. The value of the selected action does not contribute at all at this stage, instead its probability of being selected weights the instantaneous reward of the next state *and* each of the non-selected actions at the next state, which too are weighted by their probabilities of occurring as described previously. Formally, the one-step return is as follows:

$$G_{t:t+1} \doteq R_{t+1} + \gamma \sum_a \pi(a|S_{t+1})Q_t(S_{t+1}, a) \tag{50}$$

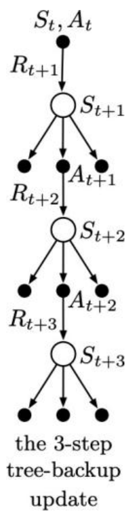


Figure 15: The 3-step tree-backup update

the two-step backup return (for $t < T - 1$) is described recursively as:

$$G_{t:t+2} \doteq R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1}) Q_t(S_{t+1}, a) + \gamma \pi(A_{t+1}|S_{t+1}) G_{t+1:t+2} \tag{51}$$

in the general case this becomes:

$$G_{t:t+n} \doteq R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1}) Q_{t+n-1}(S_{t+1}, a) + \gamma \pi(A_{t+1}|S_{t+1}) G_{t+1:t+n} \tag{52}$$

These returns are then used in the normal action-value update for *n*-step sarsa, as describedby equation 45.

A Unifying Algorithm: *n*-step Q(σ)

We'd like an algorithm that generalises the differences between n-step sarsa, as discussed at the beginning of this chapter, and n-step tree backup as discussed at the end of this chapter. To do so, we create a new algorithm parameter σ [0, 1)] that acts as a linear interpolator between the two extreme cases. This allows us to do a full, expected tree back-up in some cases, anda straight sarsa update on others, or work in the continuous region between. The family of algorithms explored in this chapter are summarised in figure 16:

Summary

n-step TD methods lie between one-step TD methods as described in the previous chapter and full Monte Carlo backups as described in the chapter two previous. They typically perform better than either of these extremes.

n-step sarsa is an on-policy algorithm that updates state-action values based on rewards obtaining n timesteps into the future following our policy π.

n-step off policy learning corrects for the failures of on-policy learning by weighting updates by the importance sampling ratio ρ_t as discussed in chapter 6.

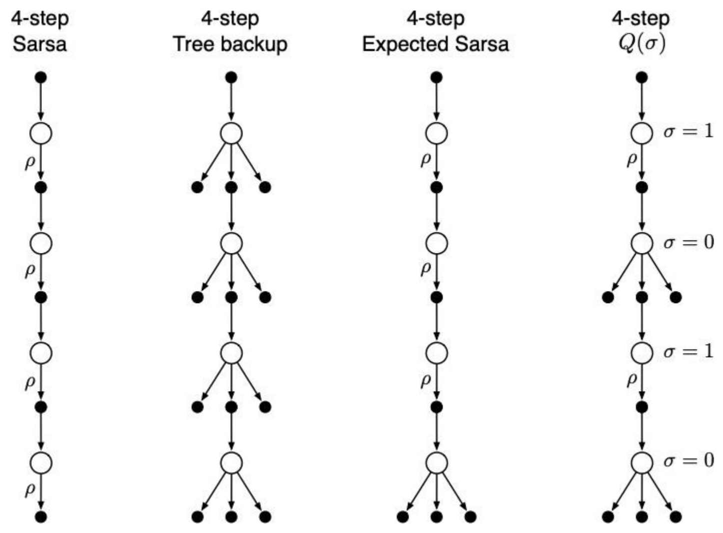


Figure 16: The backup diagrams of the three kinds of n-step action-value updates considered so far in this chapter (4-step case) plus the backup diagram of a fourth kind of update that unifies them all. The label 'ρ' indicates half transitions on which importance sampling is required in the off-policy case. The fourth kind of update unifies all the others by choosing o a state-by-state basis whether to sample ($\sigma t = 1$) or not ($\sigma t = 0$).

Variance in the off-policy *n*-step updates can be reduced using *per-decision methods with control variates* that stop the value update being 0 if the behaviour policy *b* does not match the target policy π for any of the *n* timesteps.

Finally, these methods are all more complex (expensive), and require additional memory, than one-step TD methods discussed previous.

We can do without importance sampling if our update is based on an expectation over each of the state-action values visited along our *n*-step trajectory. This is called the *n-step tree backup algorithm*.

The continuum between all of these discussed methods can be traversed using the *n*-step Q(σ) algorithm that characterised at which time steps we take full expectations and which we use only the sarsa update.

Planning and Learning with Tabular Methods

RL methods can be described as either *model-based* or *model-free*. Model-based methods rely on *planning* as their primary component, while model-free methods rely on *learning*. Despite their differences, both still use value functions and both make backups to state values based on future returns or estimates. This chapter will provide a unifying framework for model-free and model-based RL methods.

Models and Planning

Models are anything the agent can use to predict the outcome of its actions.

Models can be either *distribution models* or *sample models*. The former is a probability distribution over all possible next states whereas the latter produces one of the possible next states sampled from the probability distribution.

Distribution models are stronger than sample models because they can always be used to create samples, but sample models are easier to create in practice.

Models are used to *simulate* the environment thus producing *simulated experience*. Dis- tribution models could produce every possible episode, weighted by their probability of occurring, whereas sample models can only produce one episode.

The word *planning* refers to any computational process that takes a model as input andproduces or improves a policy for interacting with the modelled environment.

The common structure of updating our policy by utilising our model is given in Figure 17.

We can use models in place of the real environment to perform model-free learning safely i.e. using simulated experience rather than real experience.

Dyna: Integrated Planning, Acting, and Learning

Instead of planning all possible future states permutation, we may want to plan over a small number of future timesteps. When we do this we will likely want to update both our policy and model *online*. The canonical algorithm for doing so is **Dyna-Q**.

Planning agents can use the experience they collect to do one of two things: 1) improve the model (also called *model-learning*) and 2) improve the value function and policy (also called *direct reinforcement learning (direct RL)*). These options are summarised in 18. Sometimes model-learning is called *indirect RL* because improving our model, improves our policy by proxy.

Dyna-Q agents conduct direct RL, planning, model-learning and acting simultaneously. Planning and direct learning tend to use exactly the same machinery as each other e.g. *n*- step sarsa, and thus are closely linked. Figure 19 shows the generalised Dyna architecture.

Planning (in Dyna Q) is the process of running our Q updates (from Q-learning) using our model. We select random states and actions previously taken, find the reward and next state using our model and make the Q update. We do this *n* times for each timestep taken in the real environment, where *n* is a hyper parameter. For each unit of real experience collected, we are updated our value function *n* simultaneously.

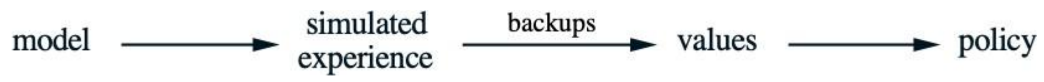


Figure 17: The framework for updating our policy using our model of the environment

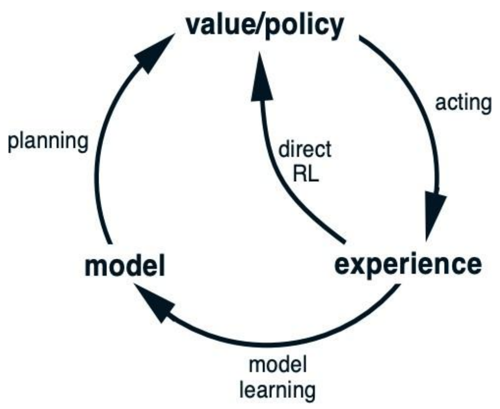


Figure 18: Direct RL and model-learning, either achieved using data collected from experience

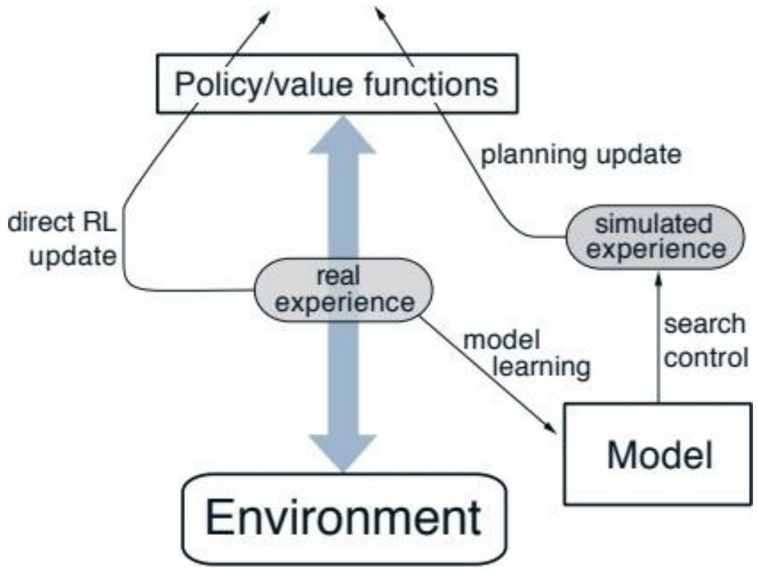


Figure 19: Generalised Dyna architecture

When the Model is Wrong

Often our model will be wrong. This can happen initially because the environment is stochastic and we haven't collected enough samples to evaluate this stochasticity accurately. Otherwise, it can happen because we've used function approximation (e.g. neural networks) to predict the model, which generally arrive with non-zero error. When the model is incorrect the planning process is likely to compute a suboptimal policy. Often the suboptimal policy quickly leads to the discovery and correction of modelling error, as when the agent takes actions prescribed by the planned policy in the real environment, it quickly finds the expected rewards are incorrect or do not exist.

Issues arise when the environment changes to become *better* or more favourable at some stage during learning. In these cases the optimal policy/route to goal remains unchanged, but there is an even better policy available that the agent may never access because it has no reason to doubt its previously learned optimal policy. To address this, another algorithm is proposed called Dyna-Q+. Here, the agent keeps track of the number of timesteps elapsed τ since a state-action pair had been selected, and, if sufficient time has elapsed, it is presumed that the dynamics of the environment from that state have changed. The modelled rewards for each state-action pair now take the form $r + k \tau$ for some small k.

Prioritized Sweeping

Until now, updates made to the value function during planning have been made arbitrarily, that is to say, state-actions have been sampled uniformly from past experience regardless of how likely we are to visit those state-actions. It would be more useful to prioritize updates to the value function that would be most effected by the reward we have recently received. This general idea is termed *backward focusing* of planning computations. If we prioritize the updates according to a measure of urgency and perform them in order of priority, we call this *prioritized sweeping*. The prioritization queue is assembled based on the expected update to a state-value pair from a new reward. We then loop through this queue, and for each state-value pair, find the state-value pairs we would predict lead to it, and calculated *their* expected update and add them to the queue. The additions to the queue are only made if the expected update is larger than some value θ. In this way, we only plan based on state-value pairs whose value updates are non-trivial. This technique is regularly 5-10 times faster than Dyna-Q.

Expected vs. Sample Updates

This book began by discussing dynamic programming; a way of conducting policy evaluation and improvement given a distribution model of the environment. Then we discussed sampling methods like: Monte Carlo, temporal-difference methods and n-step bootstrapping to estimate value functions in the absence of a model. Implicit here, was the idea that a distribution model than can be used to compute expectations is better than sampling which often leads to higher variance through sampling error. But is this true? To properly assess the relative merits of expected and sample updates for planning we must control for their different computational requirements.

In practice, the computation required by update operations is usually dominated by the number of state-action pairs at which Q is evaluated. For a particular starting pair, s, a, let b be the *branching factor* (i.e. the number of possible next states, s' for which $\hat{p}(s\ s, a) > 0$. Then an expected update of this pair requires roughly b times as much computation as a sample update. If we have sufficient time to complete an expected update, it is usually better than that of b sample updates because of the absence of sampling error. But if there is insufficient time, sampling is always better because some update is better than no update. The key question is therefore: given a unit of computational power, is it better to perform one expected update,

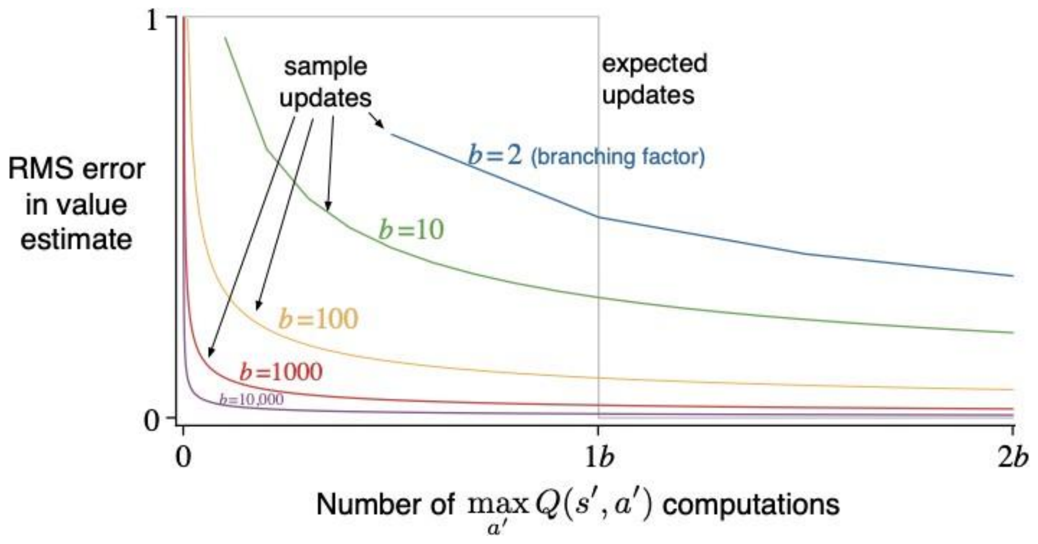


Figure 20: Comparison of efficiency of expected and sample updates

or *b* sample updates? An answer to this question is provide in Figure 20. For large values of *b* sampling has huge efficiency gains over expected updates. In reality this effect would be more pronounced as the values of the successor states would be estimates that are themselves updated. By causing estimates to be more accurate sooner, sample updates will have a second advantage in that the values backed up from the successor states will be more accurate.

Trajectory Sampling

One could update every state-action pair, or state value, by performing full sweeps of the state space using dynamic programming. As discussed on several occasions this is computationally inefficient, given many of the states may have small probability of occurring. Instead we could sample from state-action space according to some distribution; uniformly in the case of Dyna Q, or according to an on-policy distribution that we observed following the current policy. We can do so by interacting with the model, simulating explicit individual trajectories and updating state or state-action pairs encountered along the way. We call this *trajectory sampling*.

A question arises around whether on-policy updates, that follow the model's transition dis- tribution, are better than updates made with uniform probability across the state-space. It seems for problems with large state-spaces, and small branching factors, it is advantageous to use on-policy distribution sampling. In the long run, however, focusing on the on-policy distri- bution may hurt because our estimates for these values are already well defined and we are not seeing unusual parts of the state space.

Real-time Dynamic Programming

Real-time dynamic programming (RTDP) is an on-policy trajectory-sampling version of the value iteration algorithm of dynamic programming (DP). Because the algorithm follows the on- policy distribution, our value iteration updates are more efficient that normal DP because we implicitly ignore irrelevant states to our policy in our updates. RTDP is guaranteed to find a policy that is optimal on the relevant states without visiting every state infinitely often, or even without visiting some states at all, under certain conditions. These are:

The initial value of every goal state is zero

There exists at least one policy that guarantees that a goal state will be reached withprobability one from any start state.

All rewards for transitions from non-goal states are strictly negative, and

All the initial values are equal to, or greater than, their optimal values (which can besatisfied by setting initial values to 0.)

Tasks with these properties are usually called *stochastic optimal path problems*. RTDP can find optimal policies for these tasks with approximately 50% of the computation required for traditional sweep-based value iteration i.e. dynamic programming.

Planning at Decision Time

The techniques discussed thus far have involved planning in between action-selection, we call this *background planning*. Another way to do planning is at the time of action-selection i.e. to run trajectories or simulations in our model, and decide the optimal action to select based on the outcome of these trajectories. We call this *decision-time planning*.

So we now have two ways of thinking about planning:

1. Using simulated experience to gradually improve our policy or value function, or
2. Using simulated experience to select an action for the current state

Usually the results of this kind of planning are not used to update our value function or policy directly, and are generally discarded after action selection. This is fine for large state spaces, but if the state spaces are smaller than it can be expedient to keep these values and update our value function, such that we combine the two types of planning outlined above.

Decision-time planning is most useful in applications where fast response is not required, like chess. If low-latency actions are required, it's generally better to do planning in the background.

Heuristic Search

The canonical decision time planning method is *heuristic search*. For each state encountered, heuristic search dedicates computational resource to searching a large tree of possible continua- tions, or state permutations thereafter. The search evaluates leaf nodes at the end of the search and backs up the values to the state-action nodes for the current states. The value maximising action from the current state is found and then selected. The values are usually discarded. This kind of planning is effective because it focuses only on pertinent next states and actions, and focuses resource on obtaining the next best one-step action. The backup diagram for heuristic search is shown in Figure 21.

Rollout Algorithms

Rollout algorithms apply Monte Carlo control to simulated trajectories that all begin at the current environment state. They estimate action values for a given policy by

/ ≥

averaging the returns from many simulations in a model using the current policy. These algorithms, like heuristic search, make immediate use of these action value estimates then discard them, they do not attempt to estimate q_* like Monte Carlo policy iteration as discussed in chapter 5. They take advantage of the mathematical properties of policy iteration discussed in chapter 4, that is, π' ¿ π if both are identical except that $\pi'(s) = a = \pi(s)$ and $q_\pi(s, a)$ $v_\pi(s)$.

The limited factor with these types of algorithms is always computational resource, which itself is dependent on the size of the state-space. However there are ways around this, including

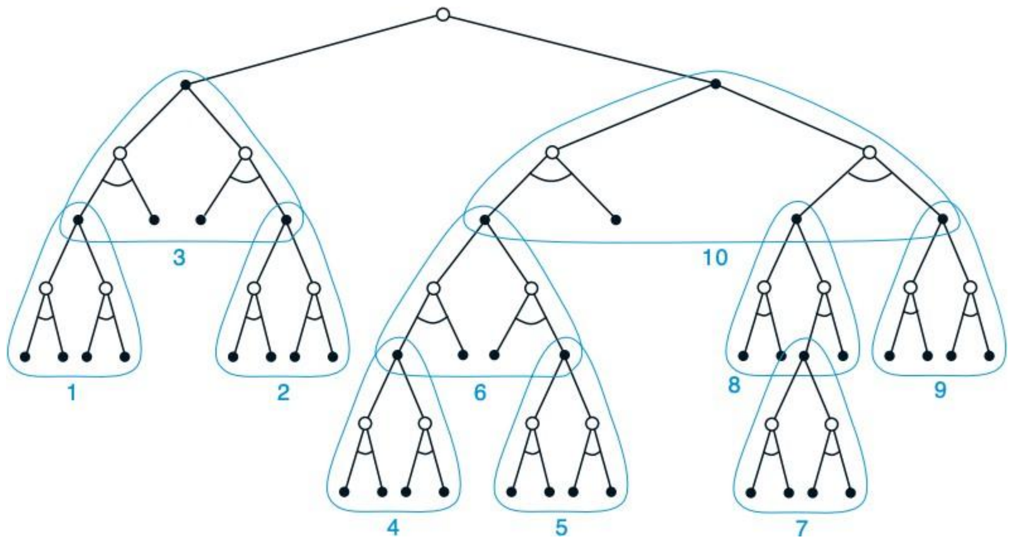


Figure 21: Heuristic search can be implemented as a sequence of one-step updates (shown here outlined in blue) backing up values from the leaf nodes toward the root. The ordering shown here is for a selective depth-first search

pruning the rollouts to only involve the most probable next states, or to truncate simulated trajectories so they do not run to the end of the episode.

Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a rollout algorithm as described above, but with some modifications. Instead of rolling out trajectories randomly, MCTS first selects a small set of actions according to its *tree policy*, then sometimes expands this selection to one or more actions, and then performs a full rollout to end of episode using a rollout policy. The value found upon termination of this trajectory is then backed up to the current state. This process is repeated for as long as time/computational constraints allow, then the agent selects an action usually based on which action produced best future estimated reward. MCTS is illustrated in Figure 22

Summary

Planning algorithms use models. These models can be either *distribution models* that provide a probability distribution over next states (as required to compute expectations for dynamic programming), or can be *sample models* that provide a sample of future states drawn from the underlying distribution that we are yet to access, or cannot access.

Algorithms used for both learning and planning are often identical, the difference being that *learning* involves updating our value function from real experience, whereas *planning* involves updating our value function based on simulated experience from our model.

Models can often be wrong, and the best way to ensure they stay up to date in continual maintain them through true experience. Sometimes we will obtain an optimal policy using a model that was once correct, but has since become redundant. We can attempt to keep an accurate model of our environment by taking actions in the environment that have not

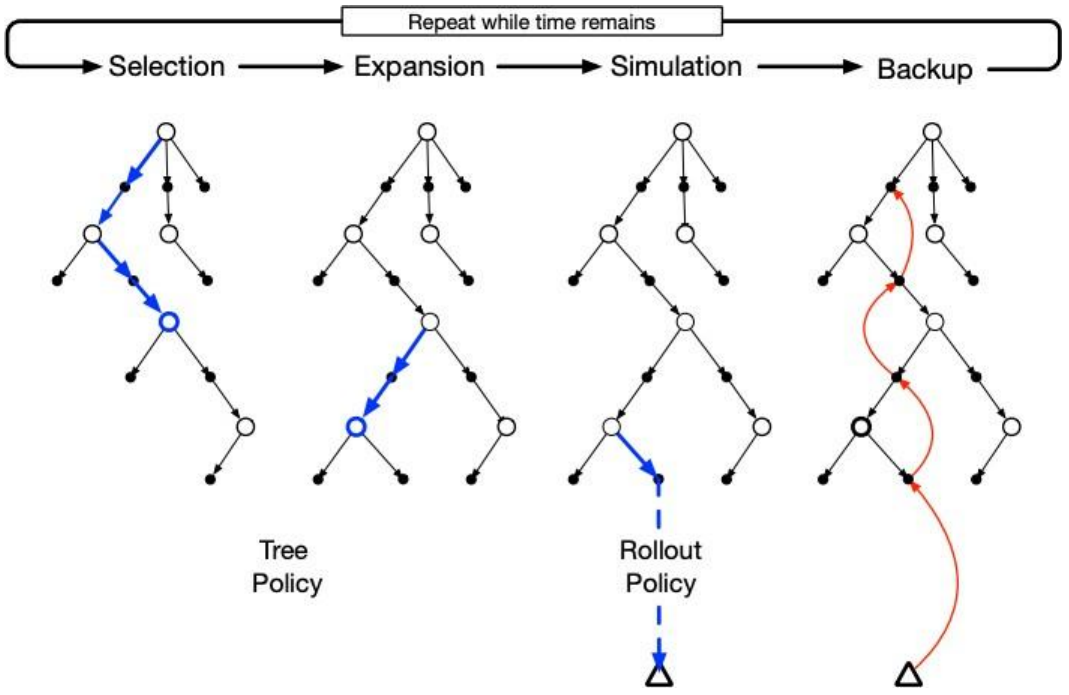


Figure 22: Monte Carlo Tree Search. When the environment changes to a new state, MCTS executes as many iterations as possible before an action needs to be selected, incrementally building a tree whose root node represents the current state. Each iteration consists of the four operations Selection, Expansion (though possibly skipped on some iterations), Simulation, and Backup, as explained in the text and illustrated by the bold arrows in the trees.

be attempted for some time to test the truth of our model. This technique is exhibited by Dyna Q+.

Sample updates are often more efficient than expected updates for large state-spaces with large branching factors (a measure of the expected number of possible next states from an action).

Trajectory sampling in our model can be used for planning. This can be useful when conducted on-policy, because we prioritise making updates to our value function that are particularly relevant to our policy.

Planning can be conducted pre-action selection, to update our value function and inform future action-selection, or it can be done at the time of action-selection. The former is called *background planning* and the latter is called *decision time planning*.

Examples of decision time planning include Heuristic Search (where value maximising actions are taken in a simulation of various next states, and backed up to the root node), Rollout Algorithms (where Monte Carlo rollouts are performed over many simulations of episodes and the value maximising action returned), or MCTS (a special case of rollout algorithms that effectively combines heuristic search over some small number of actions and then performs Monte Carlo rollouts thereafter).

Decision time planning can be effective for tasks where low-latency action selection is not required. They are fundamentally limited by the time constraints on action-selection and on the available computational resource.

Summary of Part I

Each of the methods presented this far in this textbook can be summarised on the continuum illustrated in Figure 23.

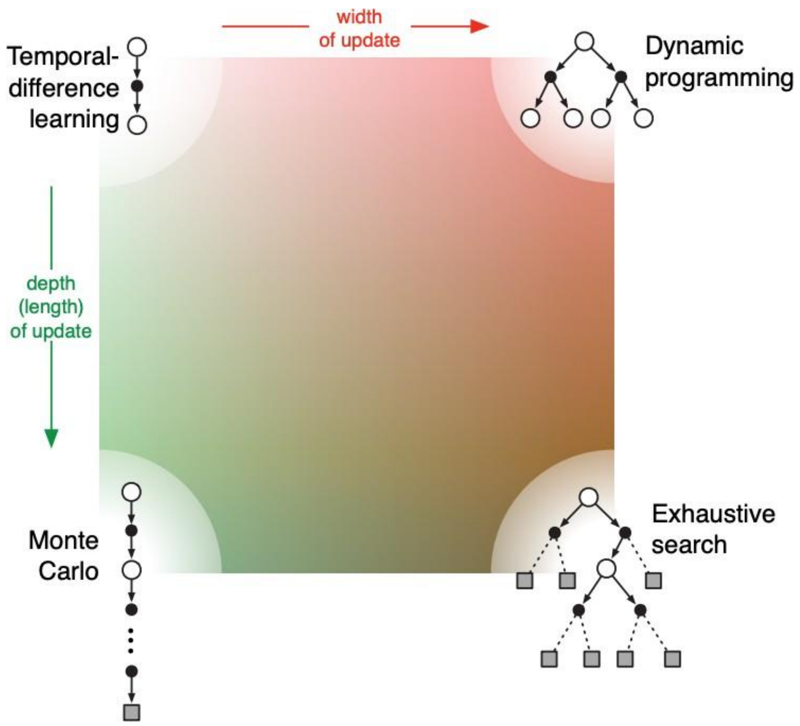


Figure 23: A slice through the space of reinforcement learning methods, highlighting the two of the most important dimensions explored in Part I of this book: the depth and width of the updates.

UNIT-V

On-policy Prediction with Approximation

Value-function Approximation

Until now we have made updates to our value function in the tabular setting, we can, however, store our value function using some kind of function approximating technique with parameters **w** R^d. Most function approximation techniques are examples of *supervised learning* that require parameters **w** to be found using training examples. In our case, we pass the function approximators our update to the value function online i.e. $s > u$ where u is the update ot the value function we'd like to make at state s.

The Prediction Objective ($\bar{VE}$)

Given we have far less parameters **w** than states s we cannot feasible approximate the value function perfectly. Indeed any updates we make the weights based on one state update will invariably lead to increased error at other states. We must therefore define which states we care about getting correct most, which we do with a state distribution $\mu(s) \geq 0, \sum_s \mu(s) = 1.$ We can weight the error in prediction between our approximate value function $\hat{v}(s, \mathbf{w})$ and the true value function $v_\pi(s)$ by this distribution to obtain our objective function, the *mean square value error*:

$$\bar{VE}(\mathbf{w}) \doteq \sum_{s \in \mathcal{S}} \mu(s) \left[v_\pi(s) - \hat{v}(s, \mathbf{w}) \right]^2 \qquad (53)$$

Often $\mu(s)$ is chosen to be the fraction of time spent in s, called the *on-policy distribution* in on-policy training.

Stochastic-gradient and Semi-gradient Methods

Our function approximator can be parametrised by a weight vector with a fixed number of real valued components, $\mathbf{w} = (w_1, w_2, \ldots, w_d)^T$. In *stochastic gradient descent*, we update the weight vector at each timestep by moving it in the direction that minimises the error most quickly forthe example shown:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \frac{1}{2}\alpha\nabla\left[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)\right]^2 \tag{54}$$

$$= \mathbf{w}_t + \alpha\left[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)\right]\nabla\hat{v}(S_t, \mathbf{w}_t) \tag{55}$$

where the second equation is obtained via the chain rule. Here, $\nabla f(\mathbf{w})$ for any scalar expression $f(\mathbf{w})$ that is a function of a vector (here $\mathbf{w}$), denotes the column vector of partial derivatives of the expression with respect to the components of the vector:

$$\nabla f(\mathbf{w}) \doteq \left(\frac{\partial f(\mathbf{w})}{\partial w_1}, \frac{\partial f(\mathbf{w})}{\partial w_2}, \cdots, \frac{\partial f(\mathbf{w})}{\partial w_d}\right)^T . \tag{56}$$

Often we will not know the *true* value function that we are trying to approximate, and instead we will be using a bootstrapped value or some other approximation. In this case we replace $v_\pi(s)$ with U_t which yields the following generalised SGD method for state-value prediction:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[U_t - \hat{v}(S_t,))\right] \nabla \hat{v}(S_t,) \tag{57}$$

If U_t is an unbiased estimate, that is, if $\mathrm{E}[U_t\ S_t |= s] = v_\pi(s)$ then $\mathbf{w}_t$ is guaranteed to converge to the local optimum. An example of an unbiased estimator is the monte carlo estimate for state s_t, unfortunately methods that use bootstrapping are biased in that their target is not independent of the the weights $\mathbf{w}_t$. We therefore call these methods *semi-gradient methods*.

Linear Methods

.

Corresponding to every state s, there is a real-valued vector $\mathbf{x}(s) = (x_1(s), x_2(s), \ldots, x_d(s))$ with the same number of components as $\mathbf{w}$. Linear methods approximate the state-value function by the inner product between $\mathbf{w}$ and $\mathbf{x}(s)$:

$$\hat{v}(s, \mathbf{w}) \doteq \mathbf{w}^T \mathbf{x}(s) \doteq \sum_{i=1}^{d} w_i x_i(s) \tag{58}$$

$\mathbf{x}(s)$ is called a *feature vector* representing state s. For linear methods, features are *basis functions* because they form a linear basis for the set of approximate functions. For linear methods, the gradient of the approximate value function w.r.t $\mathbf{w}$ in this case is:

$$\hat{v}(s, \mathbf{w})) = \mathbf{x}(s) \tag{59}$$

Thus the general SGD update reduces to a simple form:

.

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha\,[U_t - \hat{v}(S_t, \mathbf{w})]\,\mathbf{x}(S_t) \tag{60}$$

For the continuing case, the update at each time *t* is:

$$\begin{aligned}\mathbf{w}_{t+1} &\doteq \mathbf{w}_t + \alpha\left(R_{t+1} + \gamma \mathbf{w}_t^T \mathbf{x}_{t+1} - \mathbf{w}_t^T \mathbf{x}_t\right)\mathbf{x}_t & (61)\\ &= \mathbf{w}_t + \alpha\left(R_{t+1}\mathbf{x}_t + \mathbf{x}_t(\mathbf{x}_t - \gamma \mathbf{x}_{t+1}^T)\mathbf{w}_t\right) & (62)\\ & & (63)\end{aligned}$$

Once the system has reached steady state, for any given $\mathbf{w}_t$, the expected next weight vectorcan be written as

$$\mathrm{E}[\mathbf{w}_{t+1}|\mathbf{w}_t] = \mathbf{w}_t + \alpha(\mathbf{b} - \mathbf{A}\mathbf{w}_t) \tag{64}$$

where

.

It's therefore clear that the system must converge toward the weight vector $\mathbf{w}_{TD}$ at which:

$$\mathbf{b}\text{-}\mathbf{A}\mathbf{w}_{TD} = 0 \quad (66)$$

$$b = \mathbf{A}\mathbf{w}_{TD} \quad (67)$$

$$\mathbf{w}_{TD} \doteq \mathbf{A}^{-}1\mathbf{b} \quad (68)$$

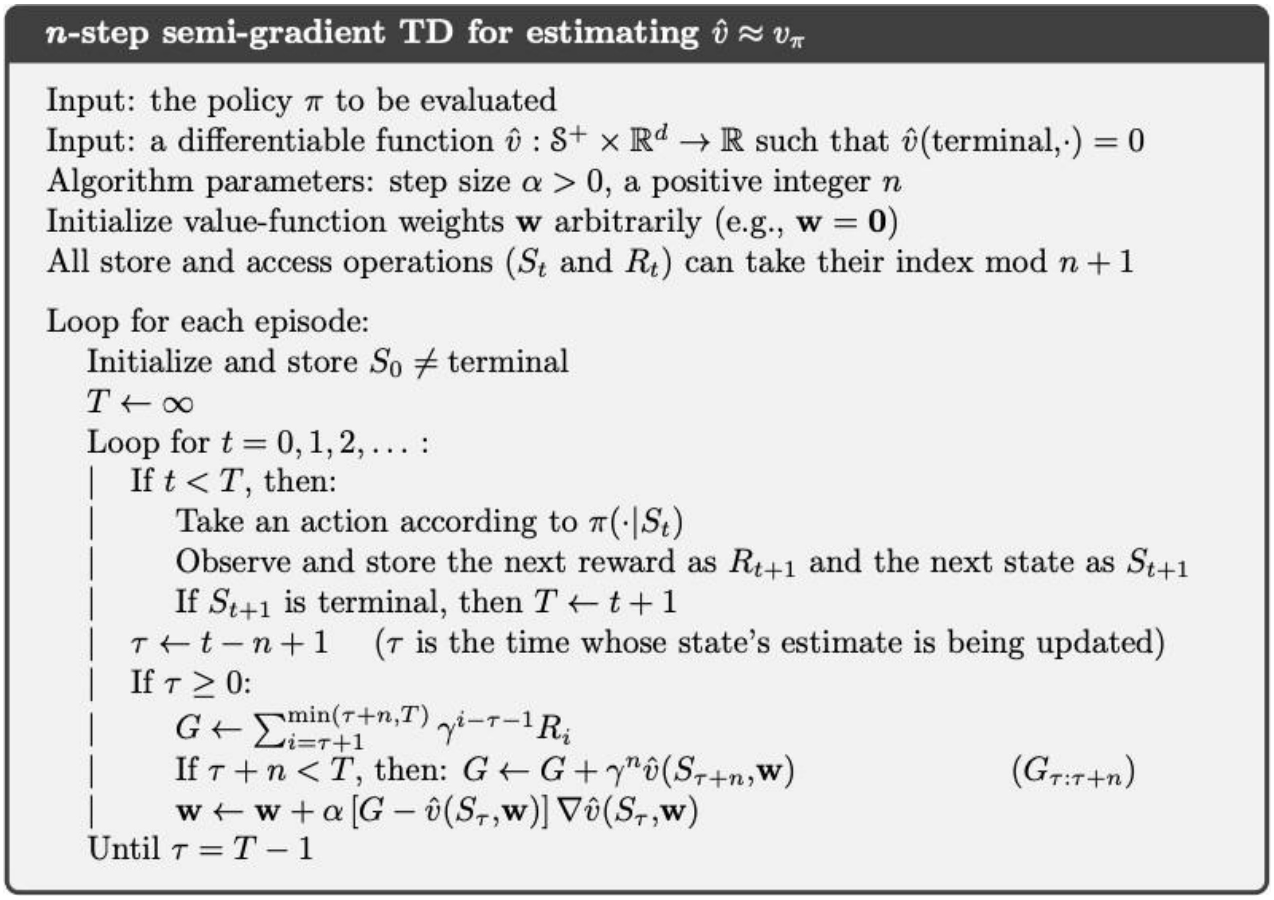
This quantity is called the *TD fixed point*. The *n*-step semi-gradient TD algorithm forestimating $\hat{v} \approx v_\pi$ is given in Figure 24.

Feature Construction for Linear Models

Constructing features that are appropriate for the RL task is critical for providing the agent useful prior knowledge about the domain. In this section we will discuss different ways of constructing features.

Polynomials

We may want to design features with higher complexity than can be captured with linear methods. If we think of the example of parametrising a state s by two dimensions $s_1 \in$ R and $s_2 \in$ R, we could represent this state as $\mathbf{x}(s) = (s_1, s_2)^T$. This, however, does not allow us to represent interactions between the two dimensions if they do indeed interact, and it would also mean that the value of our state would have to be 0 if both dimensions were zero. We

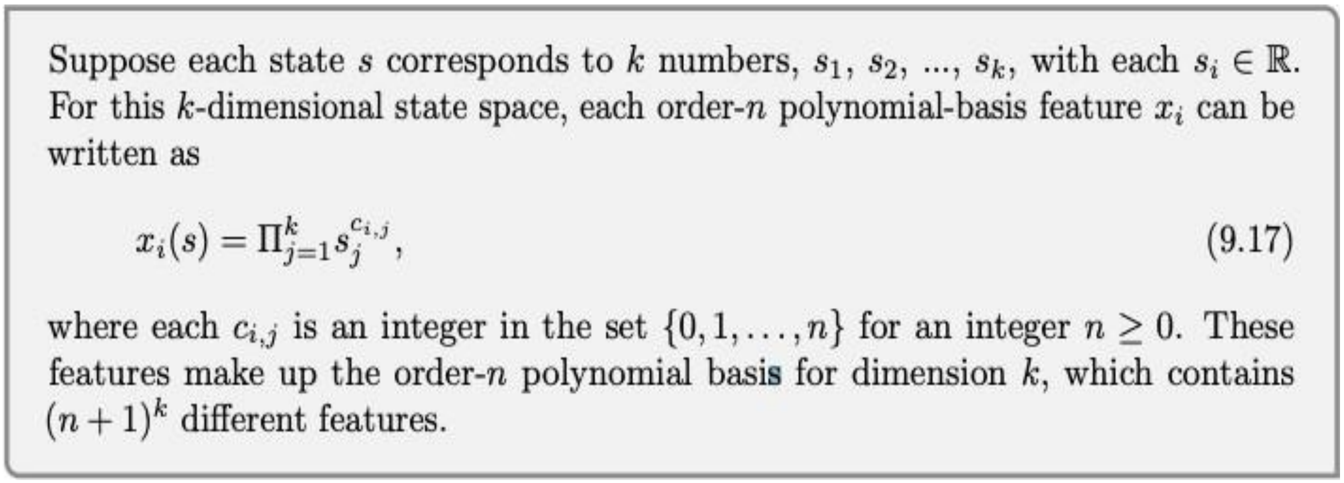


n-step semi-gradient TD for estimating $\hat{v} \approx v_\pi$

```
Input: the policy $\pi$ to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Algorithm parameters: step size $\alpha > 0$, a positive integer $n$
Initialize value-function weights $\mathbf{w}$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
All store and access operations ($S_t$ and $R_t$) can take their index mod $n+1$

Loop for each episode:
    Initialize and store $S_0 \neq$ terminal
    $T \leftarrow \infty$
    Loop for $t = 0, 1, 2, \ldots$ :
    |   If $t < T$, then:
    |       Take an action according to $\pi(\cdot|S_t)$
    |       Observe and store the next reward as $R_{t+1}$ and the next state as $S_{t+1}$
    |       If $S_{t+1}$ is terminal, then $T \leftarrow t+1$
    |   $\tau \leftarrow t - n + 1$    ($\tau$ is the time whose state's estimate is being updated)
    |   If $\tau \geq 0$:
    |       $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
    |       If $\tau + n < T$, then: $G \leftarrow G + \gamma^n \hat{v}(S_{\tau+n},\mathbf{w})$            $(G_{\tau:\tau+n})$
    |       $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[G - \hat{v}(S_\tau,\mathbf{w})\right] \nabla \hat{v}(S_\tau,\mathbf{w})$
    Until $\tau = T - 1$
```

Figure 24: Pseudocode for n-step semi-gradient TD for estimating $\hat{v} \approx v_\pi$



Suppose each state s corresponds to k numbers, s_1, s_2, ..., s_k, with each $s_i \in \mathbb{R}$. For this k-dimensional state space, each order-n polynomial-basis feature x_i can be written as

$$x_i(s) = \Pi_{j=1}^{k} s_j^{c_{i,j}}, \qquad (9.17)$$

where each $c_{i,j}$ is an integer in the set $\{0, 1, \ldots, n\}$ for an integer $n \geq 0$. These features make up the order-n polynomial basis for dimension k, which contains $(n+1)^k$ different features.

Figure 25: Generalised form of polynomial basis

can instead use a four-dimensional feature vector like: $\mathbf{x}(s) = (1, s_1, s_2, s_1 s_2)^T$. This allows us to capture any combination of these dimensions in polynomial space. The generalised form of polynomials is shown in Figure 25.

Note that $c_{i,j}$ here is a matrix with i rows corresponding to i features and j columns corre- sponding to j dimensions.

Fourier Basis

Fourier series express periodic functions as weighted sums of sine and cosine basis functions (as features) of different frequencies, with a function f being periodic if $f(x) = f(x + \tau)$ for all x and some period τ . The one-dimensional, order-n Fourier cosine basis consists of the $n + 1$ features:

$$x_i(s) = cos(i\pi s), \quad s \in [0, 1] \tag{69}$$

for i = 0, . . . n.

Coarse Coding

Consider a two dimensional state space represented by the set $s = \{s_1, s_2\}$. We can lay the state space out in two dimensions, and map circles to the state space as in Figure 26. If one state s falls within multiple overlapping circles, then another state covered by these circles s' will be updated with the same value update. How we choose to define these circles (other indeed other shapes) defines the degree of generalisation across states. In effect, we are saying the states that lay close to each other in state space should be updated generally in proportion to one other.

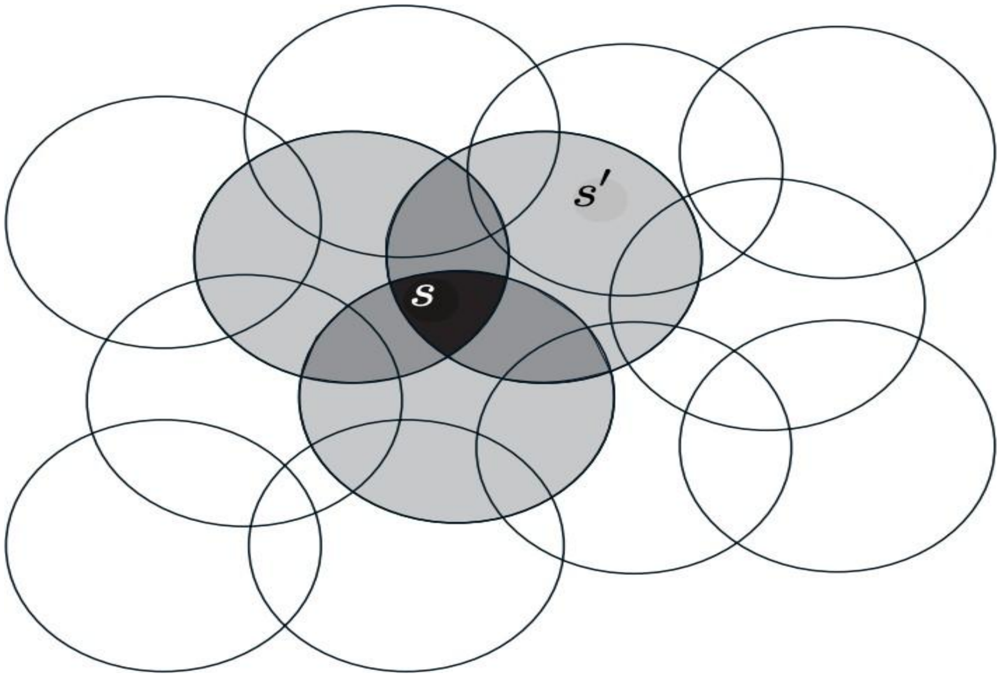


*Figure 26: Coarse coding. Generalization from state s to state s*0 *depends on the number of their features whose receptive fields (in this case, circles) overlap. These states have one feature in common, so there will be slight generalization between them.*

Tile Coding

Tile coding is similar to course coding, but now uses sets of overlapping, uniformly or asym- metrically distributed, tiles - Figure 27. Tiles do not need to be squares, they can be irregular shapes, horizontal/vertical lines or log lines. Each affects the mode of generalisation in function approximation.

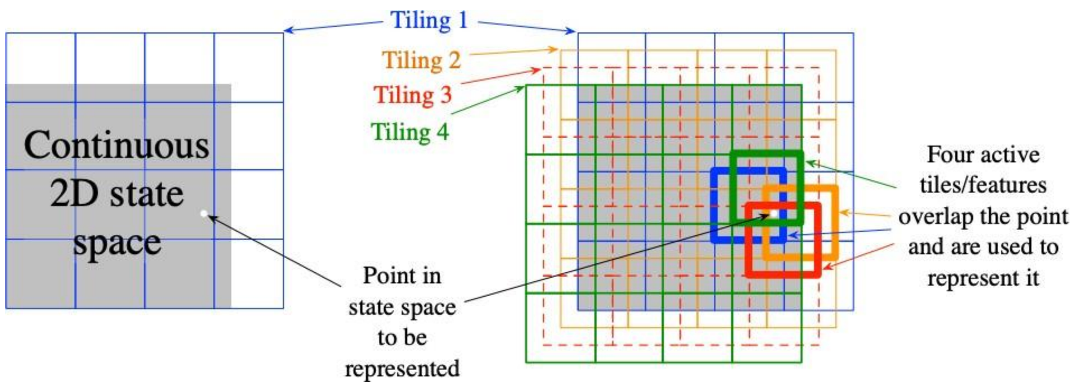

Tiling 1
Tiling 2
Tiling 3
Tiling 4
Continuous 2D state space
Point in state space to be represented
Four active tiles/features overlap the point and are used to represent it

Figure 27: Multiple, overlapping grid-tilings on a limited two-dimensional space. These tilings are offset from one another by a uniform amount in each dimension.

Radial Basis Functions

Radial Basis Functions (RBFs) are the natural extension of course coding to continuous-valued domains. Rather than a feature being either off or on (0 or 1), it can take any value in the interval [0,1]. Typical RBFs use Gaussian's to parametrise the distribution of the feature, which has centre state c_i and width σ_i. The feature value therefore takes the form:

$$x_i(s) \doteq exp\left(-\frac{||s - c_i||^2}{2\sigma_i^2}\right) \tag{70}$$

Figure 28 shows the overlapping distribution of RBF features.

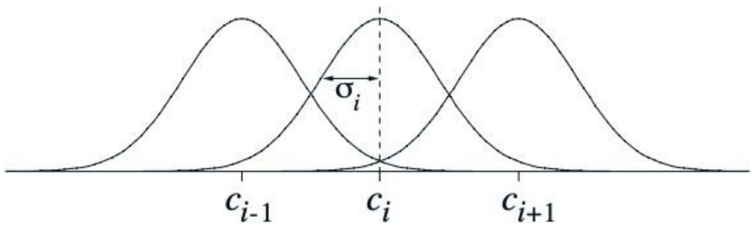


Figure 28: One-dimensional radial basis functions.

Selecting Step-Size Parameters Manually

Can we say in general how best to select α in our function approximator? A good rule of thumbfor setting the step-size parameter of linear SGD methods is:

$$\alpha \doteq (\tau \mathbb{E}[\mathbf{x}^T \mathbf{x}])^{-1} \tag{71}$$

where **x** is a random feature vector chosen from the same distribution as input vectors will be in the SGD, and τ is the number of experiences within which you would like to learn.

Nonlinear Function Approximation: Artificial Neural Networks

Artificial Neural Networks (ANNs) have proven to be some of the best non-linear function approximators. We know plenty about them already, so here is some general points:

ANNs overfit on training data, a problem mitigated by dropout.

Batch normalization involves normalising the output of hidden layers before passing them to the next layer, and performing this process on mini-batches of data. This improves the learning rates of the algorithms.

Least-Squares TD

Memory-based Function Approximation

So far we have been parametrising functions that approximate our value function, and these parameters are updated as we see more data. These are examples of *parametric* models. An alternative approach is to store examples of transitions or rewards from one state to the next, and querying them when we arrive in a state, similar to the approach used in Dyna – Chapter 8. These are *nonparametric* models or *memory-based* function approximators, with examples including nearest neighbours, or locally weighted regressions, with the weights being proportional to the distance between current state and queried state.

Because these methods are *local* approximations rather than *global*, they allow us to some- what overcome the curse of dimensionality, and only approximate the value function where it is required, rather than across the entire state space. For example, if we are working with a state space with *k* dimensions, a tabular method , storing a global approximation requires memory exponential in *k*, whereas storing examples in a memory-based method requires only memory proportional to *k*, or linear in the number of examples *n*.

Kernel-based Function Approximation

In the previous subsection, we discussed weighted average regression, where the weight is pro- portional to some measure of distance. The function that assigns the weights in often calledthe *kernel function* or simply a *kernel* and takes the form $k : \mathcal{S} \times \mathcal{S} \to \mathbb{R}$ and is summarised by $k(s, s')$. Viewed slightly differently, $k(s, s')$ is a measure of the strength of generalization from s' to s.*Kernel regression* is the memory-based method that computes a kernel weighted average of the targets of *all* examples stored in memory, assigning the result to the query state. If D is the set of stored examples, and $g(s')$ denotes the target for state s' in a stored example, then the value function is

$$\hat{v}(s, \mathcal{D}) = \sum_{s' \in \mathcal{D}} k(s, s')g(s') \tag{76}$$

The common kernel is the Gaussian radial basis function (RBF) as defined earlier and in the Gaussian Process literature. Kernel based methods allow us to operate in the high-dimensional feature space, only using stored examples, and without needing a complex, parametrised model. This is the so-called *kernel trick* and is the basis of many modern machine learning methods.

Looking Deeper at On-policy Learning: Interest and Emphasis

Until now we have treated all encountered states equally, but it is likely that we will actually be more interested in some states that others. To deal with this, we introduce two new concepts called *interest* and *emphasis*. Interest is the degree to which we value an accurate estimate of the value of a given state, generally $I_t \in [0, 1]$ and emphasis in a scalar that multiplies the learning update and thus emphasises or de-emphasises the learning done at time t. The emphasis is determined recursively from the interest by

$$M_t = I_t + \gamma^n M_{t-n} \tag{77}$$

Summary

Generalization of RL systems is critical to be applicable to artificial intelligence and large engineering problems. Generalization is achieved by approximating the value function using various techniques.

We defined the the prediction objective $\overline{VE}(\mathbf{w})$, called the *mean squared value error*, which provides a rational basis for comparing different function approximators.

Most techniques use stochastic gradient descent (SGD) to find the set of weight parameters than minimise $\overline{VE}(\mathbf{w})$

Linear methods can be proven to converge to the globally optimum weight vector under certain conditions.

Features, used to build linear methods, can take many forms, including: polynomials, fouriers, coarse codings, tile codings, and RBFs.

Artificial NNs have been proven to be potentially the most effective nonlinear function approximator.

Instead of using *parametric* function approximators, *non-parametric* models can be used that predict values based on the distance of a current state to past observed next states for which we have some stored values, or state representations.

Interest and *Emphasis* allow us to focus the function approximation on parts of the state space where we would value more accurate estimates.

On-policy Control with Approximation

Episodic Semi-gradient Control

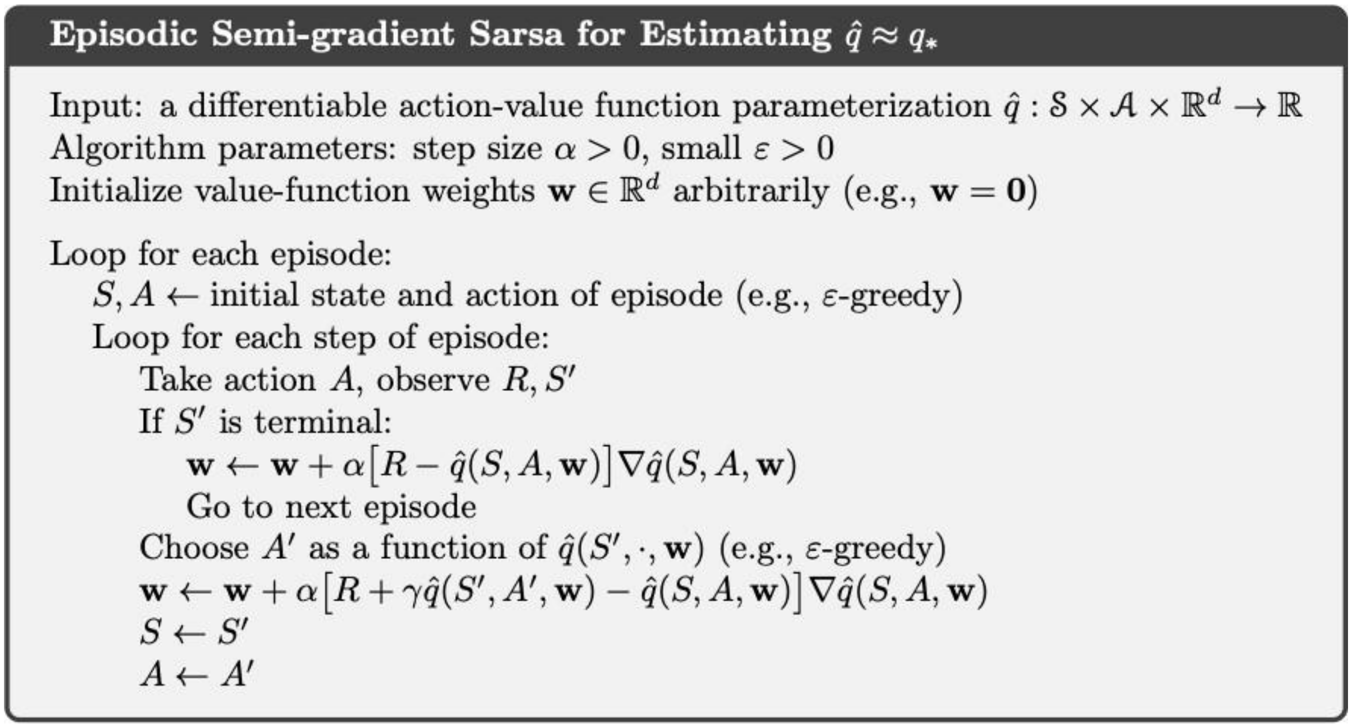
In the previous chapter, we used *semi-gradient methods,* namely semi-gradient TD(0), to approx- imate the state value function $\hat{v}_\pi$. Now, for the control setting, we want to instead approximate the state-action value function $\hat{q}_\pi$ so we can take greedy actions w.r.t it. The general gradient descent update for the action-value prediction is

.

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha\,[U_t - \hat{q}(S_t, A_t, \mathbf{w}_t)]\,\nabla\hat{q}(S_t, A_t, \mathbf{w}_t) \tag{78}$$

We can then use our function approximation to select actions greedily i.e. $A^*_{t+1} = \text{argmax}_a\, \hat{q}(S_t, a, \mathbf{w}_t)$.

In the on-policy case (this chapter) this policy used for this selection must be the same as the one used for estimation (i.e. to obtain U_t). The algorithm for episode semi-gradient sarsa is shown in Figure **??**



```
Episodic Semi-gradient Sarsa for Estimating $\hat{q} \approx q_*$

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  $S, A \leftarrow$ initial state and action of episode (e.g., $\varepsilon$-greedy)
  Loop for each step of episode:
    Take action $A$, observe $R, S'$
    If $S'$ is terminal:
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
      Go to next episode
    Choose $A'$ as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., $\varepsilon$-greedy)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
    $S \leftarrow S'$
    $A \leftarrow A'$
```

Figure 29: Episodic semi-gradient sarsa for estimating $\hat{q} \approx q_*$

Semi-gradient *n*-step Sarsa

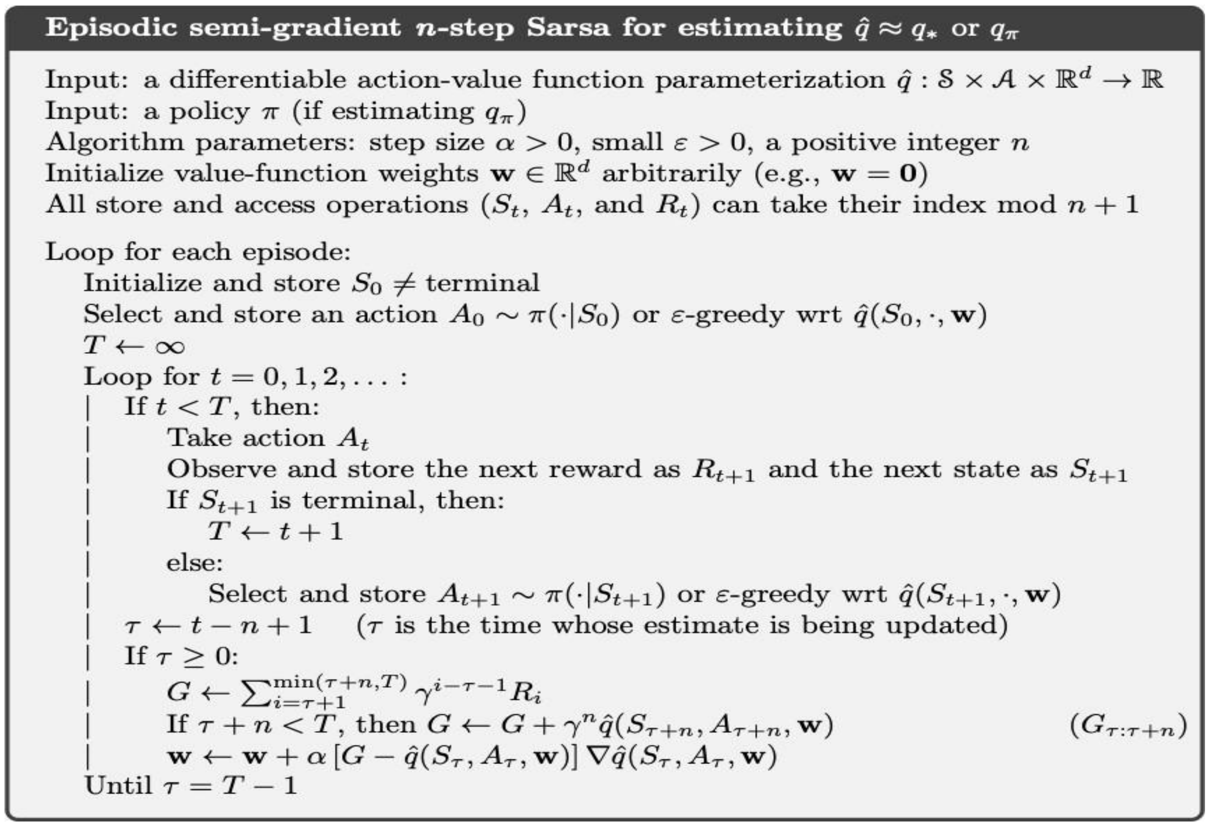
The *n*-step semi-gradient sarsa algorithm follows logically from the previous section. The *n*-stepreturn becomes

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} + R_{t+n} + \gamma \hat{q}(S_{t+n}, A_{t+n}, \mathbf{w}_{t+n-1}) \tag{79}$$

The n-step update becomes

$$\mathbf{w}_{t+n} \doteq \mathbf{w}_{t+n-1} + \alpha \left[G_{t:t+n} - \hat{q}(S_t, A_t, \mathbf{w}_{t+n-1})\right] \nabla \hat{q}(S_t, A_t, \mathbf{w}_{t+n-1}) \tag{80}$$

The algorithm is provided in Figure 30. Generally, an intermediate level of bootstrappingshows optimal performance.



Episodic semi-gradient n-step Sarsa for estimating $\hat{q} \approx q_*$ or q_π

```
Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Input: a policy $\pi$ (if estimating $q_\pi$)
Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$, a positive integer $n$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
All store and access operations ($S_t$, $A_t$, and $R_t$) can take their index mod $n+1$

Loop for each episode:
    Initialize and store $S_0 \neq$ terminal
    Select and store an action $A_0 \sim \pi(\cdot|S_0)$ or $\varepsilon$-greedy wrt $\hat{q}(S_0, \cdot, \mathbf{w})$
    $T \leftarrow \infty$
    Loop for $t = 0, 1, 2, \dots$ :
    |   If $t < T$, then:
    |       Take action $A_t$
    |       Observe and store the next reward as $R_{t+1}$ and the next state as $S_{t+1}$
    |       If $S_{t+1}$ is terminal, then:
    |           $T \leftarrow t + 1$
    |       else:
    |           Select and store $A_{t+1} \sim \pi(\cdot|S_{t+1})$ or $\varepsilon$-greedy wrt $\hat{q}(S_{t+1}, \cdot, \mathbf{w})$
    |   $\tau \leftarrow t - n + 1$    ($\tau$ is the time whose estimate is being updated)
    |   If $\tau \geq 0$:
    |       $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n, T)} \gamma^{i-\tau-1} R_i$
    |       If $\tau + n < T$, then $G \leftarrow G + \gamma^n \hat{q}(S_{\tau+n}, A_{\tau+n}, \mathbf{w})$        $(G_{\tau:\tau+n})$
    |       $\mathbf{w} \leftarrow \mathbf{w} + \alpha [G - \hat{q}(S_\tau, A_\tau, \mathbf{w})] \nabla \hat{q}(S_\tau, A_\tau, \mathbf{w})$
    Until $\tau = T - 1$
```

Figure 30: Episodic semi-gradient n-step sarsa for estimating $q\hat{} \approx q*$

Average Reward: A New Problem Setting for Continuing Tasks

Until now we have considered two types of rewards: episodic and discounted (for continuing tasks). The average reward is a third task with no discounting, such that future rewards are valued just the same as present rewards. In the average-reward setting, the quality of a policy π is defined as the average rate of reward, or simply *average reward*, while following that policy, which we denote as $r(\pi)$

$$r(\pi) \doteq \lim_{h\to\infty} \frac{1}{h} \sum_{t=1}^{h} \mathbb{E}\left[R_t | S_0, A_{0:t-1}\ \pi\right] \tag{81}$$

$$= \lim_{t\to\infty} \mathbb{E}\left[R_t | S_0, A_{0:t-1}\ \pi\right] \tag{82}$$

$$= \sum_s \mu_\pi(s) \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) r \tag{83}$$

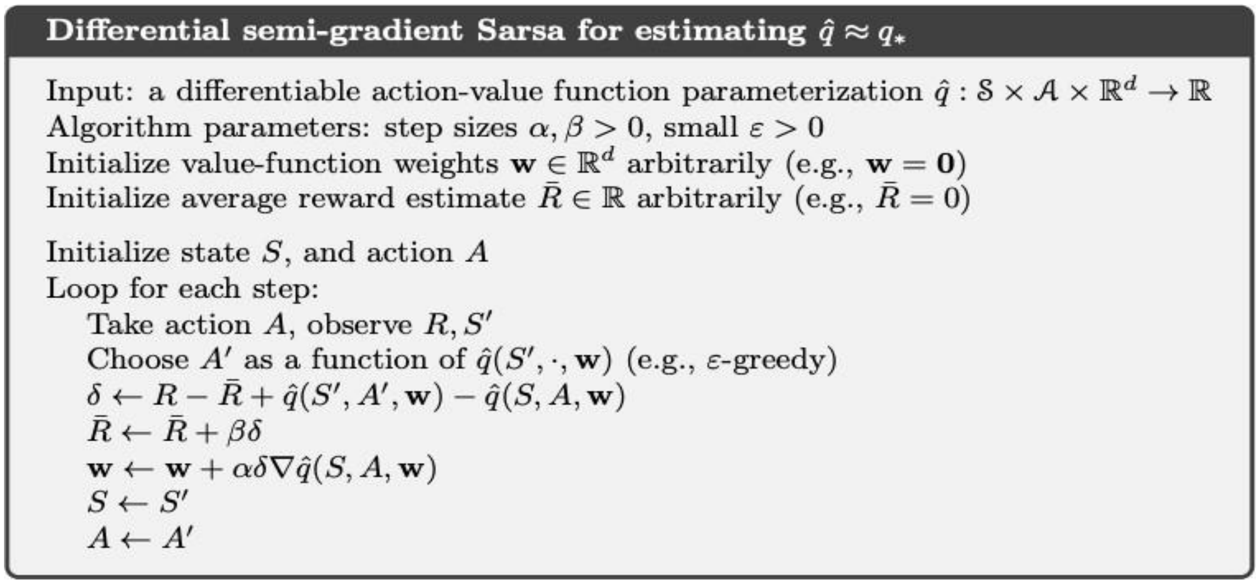
The last equation holds if $\mu_\pi(s)$ exists and is independent of S_0, in other words if the MDP is *ergodic* i.e. the long run expectation of being in a state depends only on the policy and the MDP transition probabilities.
We can order policies based on their average reward per timestep–$r(\pi)$– and all policies that attain the maximal reward rate are considered optimal.
In the average-reward setting, returns are defined in terms of differences between rewards and the average reward

$$G_t \doteq R_{t+1} - r(\pi) + Rt + 2 - r(\pi) + Rt + 3 - r(\pi) + \cdots \tag{84}$$

known as the *differential return* with corresponding value functions known as *differential value functions*. This can be used to create new TD errors, the action-value error being

$$\delta_t \doteq R_{t+1} - \bar{R}_t + \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t) \tag{85}$$

where $\bar{R}_t$ is an estimate at time t of the average reward $r(\pi)$. This error can be used to create a differential semi-gradient sarsa algorithm as shown in Figure 31



Differential semi-gradient Sarsa for estimating $\hat{q} \approx q_*$

```
Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameters: step sizes $\alpha, \beta > 0$, small $\varepsilon > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
Initialize average reward estimate $\bar{R} \in \mathbb{R}$ arbitrarily (e.g., $\bar{R} = 0$)

Initialize state $S$, and action $A$
Loop for each step:
    Take action $A$, observe $R, S'$
    Choose $A'$ as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., $\varepsilon$-greedy)
    $\delta \leftarrow R - \bar{R} + \hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})$
    $\bar{R} \leftarrow \bar{R} + \beta\delta$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\nabla\hat{q}(S, A, \mathbf{w})$
    $S \leftarrow S'$
    $A \leftarrow A'$
```

Figure 31: Differential semi-gradient sarsa for estimating $\hat{q} \approx q_*$

Deprecating the Discounted Setting

Should we use the discounted reward or the average reward in continuous settings? It turns out there is no benefit to using discounting in the continuous setting, as, given a large enough sequences of rewards (infinite in the limit) we end up multiplying every reward by the same sequence of discounts $1 + \gamma + \gamma^2 + \gamma^3 + \gamma^4 + \; = \frac{1}{1-\gamma}$.

The root cause of difficulties with the discounted control setting is that with function ap- proximation we lose the policy improvement theorem from chapter 4. We can no longer say that if we change the policy to improve the discounted value of one state we are guaranteed to have improved the overall policy - errors in our function approximation mean we could have a detrimental effect on our value function elsewhere.

Differential Semi-gradient *n*-step Sarsa

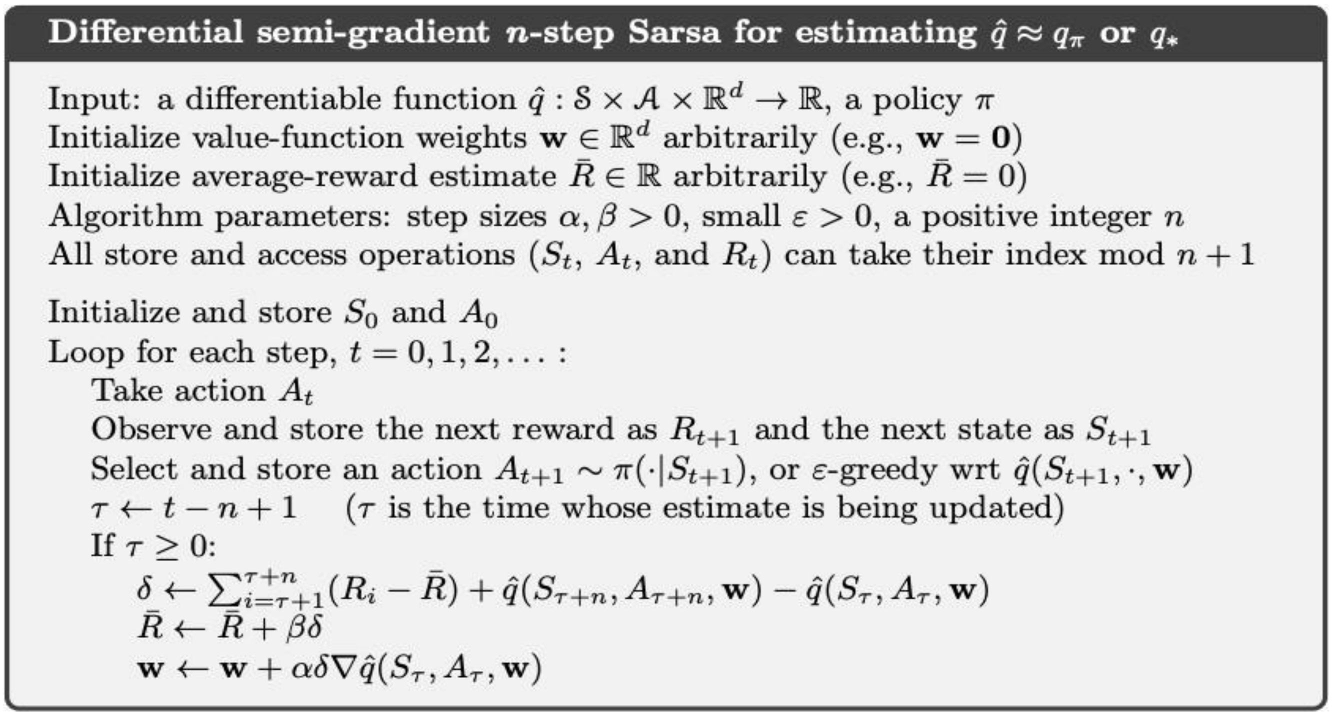
For the *n*-step generalisation of Sarsa, we just need an expression for the *n*-step differential return with function approximation:

$$G_{t:t+n} \doteq R_{t+1} - \bar{R}_{t+n-1} + \cdots + R_{t+n} - \bar{R}_{t+n-1} - \hat{q}(S_{t+n}, A_{t+n}, \mathbf{w}_{t+n-1}) \tag{86}$$

The pseudocode is given in Figure 32.

Summary

In the episodic case, control with function approximation works the same as before, we find the approximate value function and act greedily.



Differential semi-gradient n-step Sarsa for estimating $\hat{q} \approx q_\pi$ or q_*

```
Input: a differentiable function $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$, a policy $\pi$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
Initialize average-reward estimate $\bar{R} \in \mathbb{R}$ arbitrarily (e.g., $\bar{R} = 0$)
Algorithm parameters: step sizes $\alpha, \beta > 0$, small $\varepsilon > 0$, a positive integer $n$
All store and access operations ($S_t$, $A_t$, and $R_t$) can take their index mod $n+1$

Initialize and store $S_0$ and $A_0$
Loop for each step, $t = 0, 1, 2, \ldots$ :
    Take action $A_t$
    Observe and store the next reward as $R_{t+1}$ and the next state as $S_{t+1}$
    Select and store an action $A_{t+1} \sim \pi(\cdot|S_{t+1})$, or $\varepsilon$-greedy wrt $\hat{q}(S_{t+1}, \cdot, \mathbf{w})$
    $\tau \leftarrow t - n + 1$    ($\tau$ is the time whose estimate is being updated)
    If $\tau \geq 0$:
        $\delta \leftarrow \sum_{i=\tau+1}^{\tau+n}(R_i - \bar{R}) + \hat{q}(S_{\tau+n}, A_{\tau+n}, \mathbf{w}) - \hat{q}(S_\tau, A_\tau, \mathbf{w})$
        $\bar{R} \leftarrow \bar{R} + \beta\delta$
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\nabla\hat{q}(S_\tau, A_\tau, \mathbf{w})$
```

Figure 32: Differential semi-gradient n-step sarsa for estimating $\hat{q} \approx q_*$

In the continuing task setting we need to introduce a new reward–the average reward $r(\pi)$–which represents our expectation over the reward space.

Our updates are now governed by the *differential reward* which is the difference between our observed reward and current best estimate for the average reward.

Discounting does not matter/work in the presence of approximations, as in this case, most policies cannot be represented by a value function. Instead, policies need to be ranked and the average reward $r(\pi)$ provides an effective way to do this.

Eligibility Traces

Eligibility traces are an alternative way of performing *n*-step TD prediction or control.

n-step methods can be seen as *forward view* algorithms that look at future rewards seen from some state and update the state value accordingly. Eligibility traces are *backward view* methods that update previous state values based on how *responsible* we think they were for the reward we have just obtained.

They are more computationally efficient than *n*-step methods because they don't have to store the last *n* feature vectors, but instead store one *trace* vector.

The λ-return

In Chapter 7 we derived the *n*-step return $G_{t:t+n}$ and used it to make updates to the state value. What if instead of using one *n*-step return, we took a weighted average of every *n*-step return? This is the λ-return and is called a *compound update*, rather than a *component update* in the *n*-step case. We could, for example, take the mean of the 2-step and 4-step updates, the backup diagram for which is shown in Figure 33.

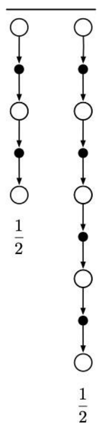


Figure 33: Backup diagram for compound update for 2-step and 4-step returns

The TD(λ) algorithm generalises this approach such that each *n*-step update is weighted proportionally to λ^{n-1} (where $\lambda \in [0, 1]$). The λ*-return* in the state-based form is given by

$$G_t = (1 - \lambda)$$

and the backup diagram for the algorithm is given in Figure 34, the decaying weighting factoris given in Figure 35.

Intuitively this algorithm makes sense. We give more responsibility for our rewards to re- cently selected actions and less to older selected actions, but nonetheless take into account all previously taken actions. Crucially, this algorithm is forward view, the episode is completed then we return to the first step and update based on the weighted average of all *n*-step returns received thereafter, cycling through each state.

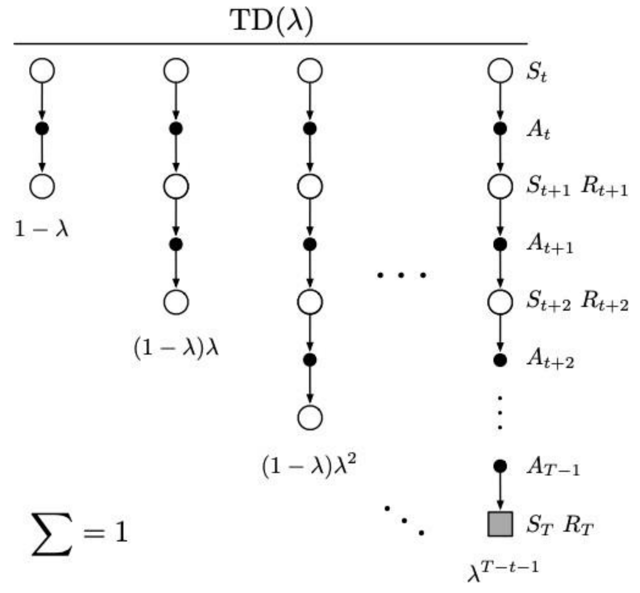


Figure 34: The backup digram for TD(λ). If λ = 0, then the overall update reduces to its first component, the one-step TD update, whereas if λ = 1, then the overall update reduces to its last component, the Monte Carlo update.

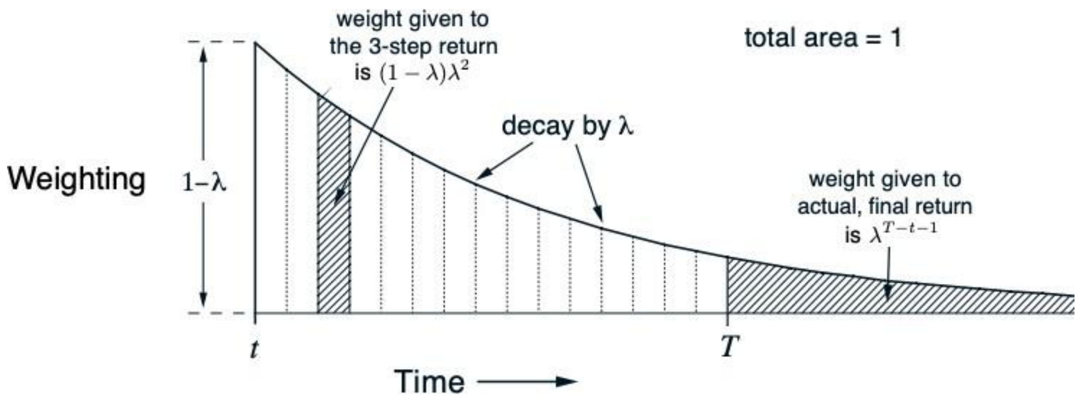

weight given to the 3-step return is $(1-\lambda)\lambda^2$
total area = 1
decay by λ
Weighting
1−λ
weight given to actual, final return is λ^{T-t-1}
t
T
Time

Figure 35: Weighting given in the λ-return to each of the n-step returns.

Semi-gradient TD(λ) for estimating $\hat{v} \approx v_\pi$

```
Input: the policy $\pi$ to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Algorithm parameters: step size $\alpha > 0$, trace decay rate $\lambda \in [0, 1]$
Initialize value-function weights $\mathbf{w}$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
    Initialize $S$
    $\mathbf{z} \leftarrow \mathbf{0}$                                        (a $d$-dimensional vector)
    Loop for each step of episode:
    |   Choose $A \sim \pi(\cdot|S)$
    |   Take action $A$, observe $R, S'$
    |   $\mathbf{z} \leftarrow \gamma\lambda\mathbf{z} + \nabla\hat{v}(S,\mathbf{w})$
    |   $\delta \leftarrow R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$
    |   $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\mathbf{z}$
    |   $S \leftarrow S'$
    until $S'$ is terminal
```

Figure 36: Pseudocode for semi-gradient TD(λ)

TD(λ)

In TD(λ) the eligibility trace is first initialized to 0, and then incremented at each step by thevalue gradient and fades according to λ

.

–

here the value gradient is effectively a proxy for the state visited i.e. bump up the eligibility at these weights because a state corresponding to these weights has been visited recently. We know the TD error for state-value prediction is

.

$$\delta_t = R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t) \tag{90}$$

and the weight vector in TD(λ) is updated in proportion to the TD error and the eligibility trace

.

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha\delta_t \mathbf{z}_t \tag{91}$$

The algorithm for semi-gradient TD(λ) is given in Figure 36

To understand TD(λ) better lets consider the extreme cases of $\lambda = 0$ and $\lambda = 1$. In the former, the algorithm becomes one step returns as per TD(0) discussed in chapter 6. In the later we get Monte Carlo returns where every single previous state is updated with the return, but discounted by γ. (Note: this gives insight as to why γ is still retained in equation 97). The advantage of this kind of Monte Carlo return, as compared with the normal MC return discussed in chapter 5, is that it can be performed *online* rather than waiting for the end of the episode before backpropogating the returns.

n-step Truncated λ-return Methods

The off-line λ-return algorithm, which uses the λ-return (Equation 87), is an important ideal but of little use in the episodic case (because we have to wait until the end of the episode before we make any updates), nor the continuing case (because it cannot be calculated for arbitrarily large n). Instead we can use the truncated λ-return which halts the return calculation at some horizon h:

$$G^{\lambda}_{t:h} \doteq (1-\lambda)\sum_{n=1}^{h-t-1} \lambda^{n-1} G_{t:t+n} + \lambda^{h-t-1} G_{t:h}, \quad 0 \le t < h \le T \tag{92}$$

$$h = 1: \quad \mathbf{w}_1^1 \doteq \mathbf{w}_0^1 + \alpha \left[G_{0:1}^{\lambda} - \hat{v}(S_0, \mathbf{w}_0^1) \right] \nabla \hat{v}(S_0, \mathbf{w}_0^1),$$

$$h = 2: \quad \mathbf{w}_1^2 \doteq \mathbf{w}_0^2 + \alpha \left[G_{0:2}^{\lambda} - \hat{v}(S_0, \mathbf{w}_0^2) \right] \nabla \hat{v}(S_0, \mathbf{w}_0^2),$$
$$\mathbf{w}_2^2 \doteq \mathbf{w}_1^2 + \alpha \left[G_{1:2}^{\lambda} - \hat{v}(S_1, \mathbf{w}_1^2) \right] \nabla \hat{v}(S_1, \mathbf{w}_1^2),$$

$$h = 3: \quad \mathbf{w}_1^3 \doteq \mathbf{w}_0^3 + \alpha \left[G_{0:3}^{\lambda} - \hat{v}(S_0, \mathbf{w}_0^3) \right] \nabla \hat{v}(S_0, \mathbf{w}_0^3),$$
$$\mathbf{w}_2^3 \doteq \mathbf{w}_1^3 + \alpha \left[G_{1:3}^{\lambda} - \hat{v}(S_1, \mathbf{w}_1^3) \right] \nabla \hat{v}(S_1, \mathbf{w}_1^3),$$
$$\mathbf{w}_3^3 \doteq \mathbf{w}_2^3 + \alpha \left[G_{2:3}^{\lambda} - \hat{v}(S_2, \mathbf{w}_2^3) \right] \nabla \hat{v}(S_2, \mathbf{w}_2^3).$$

The general form for the update is

$$\mathbf{w}_{t+1}^h \doteq \mathbf{w}_t^h + \alpha \left[G_{t:h}^{\lambda} - \hat{v}(S_t, \mathbf{w}_t^h) \right] \nabla \hat{v}(S_t, \mathbf{w}_t^h), \quad 0 \leq t < h \leq T.$$

Figure 37: Online λ-return updates for the first three steps of an episode

Redoing Updates: Online λ-return Algorithm

The problem with the algorithm described in the last section is the choice of truncation pa- rameter *n*. If it is too small, then we do not get the benefit of the offline λ-return algorithm that uses all *n* step updates from an episode, and if it is too large then we cannot learn quickly online. We can, in principle, get the best of both using the *online λ-return algorithm*, but we pay for this with computational complexity.

For every step we take in an episode, we recalculate the λ-return for the whole episode, where our current step acts as horizon *h* in equation 92. We are then making increasingly accurate updates to states as we move through the episode, but the number of calculations required at each step scales with *n*. The updates are given in Figure 37.

In general, online λ-return performs better than off λ-return because it has had more, in- formative updates the weight vector used for bootstrapping throughout the episode. Note: this is a forward view algorithm, we keep updating state values by looking at a stream of future rewards as they are updated

True Online TD(λ)

We can produce a backward view version of the online λ-return algorithm presented inthe last section that uses eligibility traces, called True Online TD(λ).

Because of a trick with the weight matrix, where we only need to keep the last weight vector from all the updates at the last time step, the computational complexity of TrueOnline TD(λ) matches TD(λ).

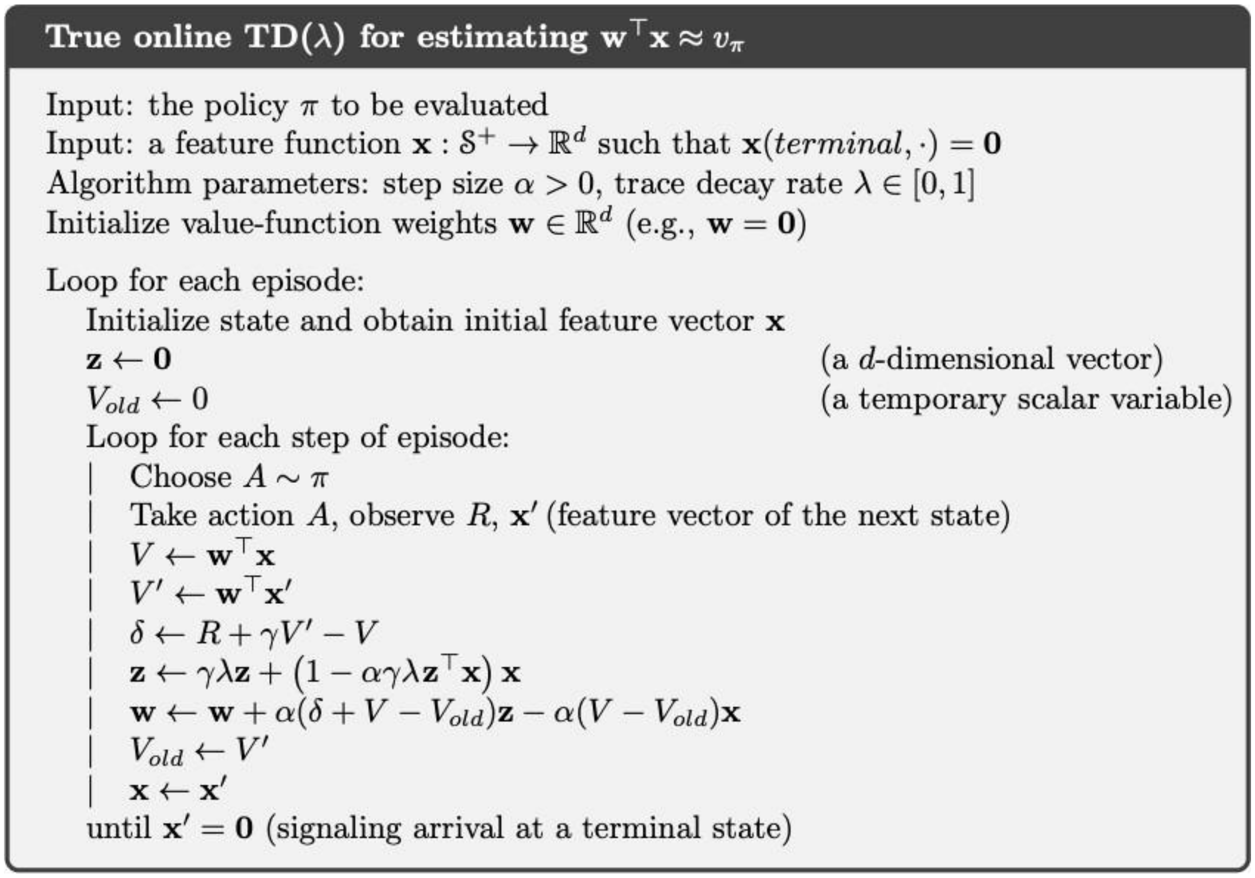
The pseudocode for True Online TD(λ) is given in Figure 38.

SARSA(λ)

We can extend the ideas discussed in this chapter (i.e. eligibility traces) to the action-value, forward-view case readily, producing the SARSA(λ) algorithm. The full list of equations required for SARSA(λ) are detailed below. First we need the action-value form of the *n*-step return

$$G_{t:t+n} \doteq R_{t+1} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \hat{q}(S_{t+n}, A_{t+n}, \mathbf{w}_{t+n-1}), \quad t+n < T, \qquad (93)$$

with $G_{t:t+n} \doteq G_t$ if $t+n \geq T$. The action-value form of the offline λ-return algorithm simply use $\hat{q}$ rather than $\hat{v}$:



True online TD(λ) for estimating $\mathbf{w}^\top\mathbf{x} \approx v_\pi$

```
Input: the policy $\pi$ to be evaluated
Input: a feature function $\mathbf{x} : \mathcal{S}^+ \to \mathbb{R}^d$ such that $\mathbf{x}(terminal, \cdot) = \mathbf{0}$
Algorithm parameters: step size $\alpha > 0$, trace decay rate $\lambda \in [0,1]$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
    Initialize state and obtain initial feature vector $\mathbf{x}$
    $\mathbf{z} \leftarrow \mathbf{0}$                                        (a $d$-dimensional vector)
    $V_{old} \leftarrow 0$                                      (a temporary scalar variable)
    Loop for each step of episode:
    |   Choose $A \sim \pi$
    |   Take action $A$, observe $R$, $\mathbf{x}'$ (feature vector of the next state)
    |   $V \leftarrow \mathbf{w}^\top\mathbf{x}$
    |   $V' \leftarrow \mathbf{w}^\top\mathbf{x}'$
    |   $\delta \leftarrow R + \gamma V' - V$
    |   $\mathbf{z} \leftarrow \gamma\lambda\mathbf{z} + \left(1 - \alpha\gamma\lambda\mathbf{z}^\top\mathbf{x}\right)\mathbf{x}$
    |   $\mathbf{w} \leftarrow \mathbf{w} + \alpha(\delta + V - V_{old})\mathbf{z} - \alpha(V - V_{old})\mathbf{x}$
    |   $V_{old} \leftarrow V'$
    |   $\mathbf{x} \leftarrow \mathbf{x}'$
    until $\mathbf{x}' = \mathbf{0}$ (signaling arrival at a terminal state)
```

Figure 38: True Online TD(λ)for estimating w T x ≈ vπ

$$\mathbf{w}_{t+n} \doteq \mathbf{w}_{t+n-1} + \alpha\left[G_t^\lambda - \hat{q}(S_t, A_t, \mathbf{w}_{t+n-1})\right]\nabla\hat{q}(S_t, A_t, \mathbf{w}_{t+n-1}), \quad t = 0, \ldots, T-1 \tag{94}$$

where $G_t^\lambda \doteq G_{t:\infty}^\lambda$. The algorithm has the same update rule as given earlier for TD(λ)

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\delta_t\mathbf{z}_t \tag{95}$$

except it uses the action-value form of the TD error:

$$\delta_t \doteq R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t), \tag{96}$$

and the action-value form of the eligibility trace:

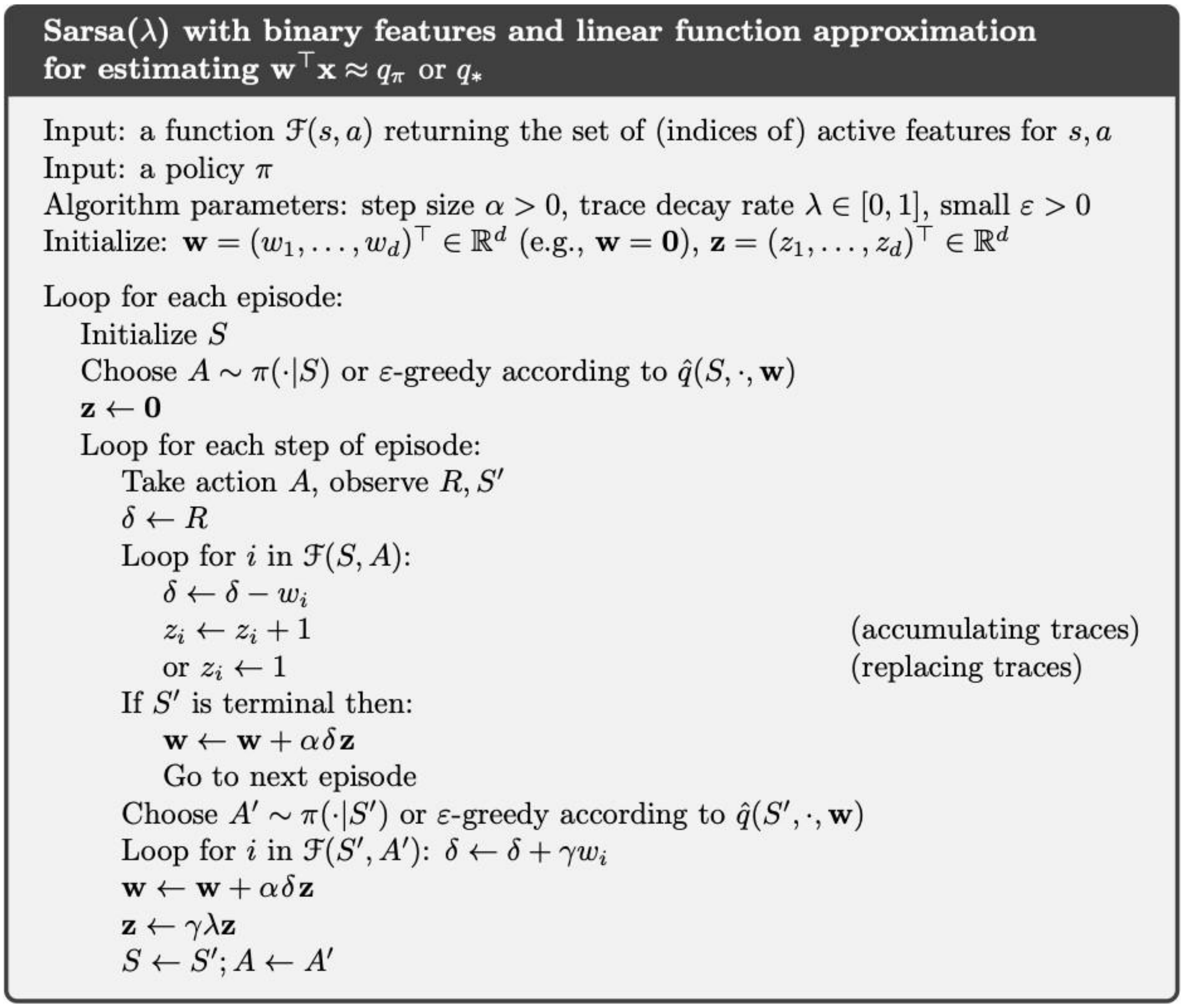
$$\mathbf{z}_{-1} \doteq \mathbf{0} \tag{97}$$

$$\mathbf{z}_t \doteq \gamma\lambda\mathbf{z}_{t-1} + \nabla\hat{q}(S_t, A_t, \mathbf{w}_t), \quad 0 \le t \le T. \tag{98}$$

Complete pseudocode for the algorithm is given in Figure 39.

A compelling visual description of the different SARSA algorithms reviewed thus far is given in Figure 40. The SARSA(λ) updates make the most intuitive sense. It is the actions closest to the final reward where we can be most confident in their value, actions prior to the goal may have had a direct effect in producing the reward, but the distance between action and target means there is uncertainty in how much that action affected the final outcome. Consequently, updating the values of these actions more conservatively appears rational. SARSA(λ) proves the most efficient and learning optimal policies in these settings for this reason.

To explain clearly the intuition behind this algorithm we refer back to Figure 35. If we imagine ourselves in the first state and observe the weight we give to all forthcoming updates, we see that we give the bulk of the weight to the initial rewards we receive (0 in the gridworld) and the least weight to the final return at time T . Equally, if we are in the final state prior to termination, we give most weight to the next-step rewards (1 in the gridworld). This creates the fading reward traces shown in Figure 39.



Sarsa(λ) with binary features and linear function approximation for estimating $\mathbf{w}^\top \mathbf{x} \approx q_\pi$ or q_*

```
Input: a function $\mathcal{F}(s, a)$ returning the set of (indices of) active features for $s, a$
Input: a policy $\pi$
Algorithm parameters: step size $\alpha > 0$, trace decay rate $\lambda \in [0, 1]$, small $\varepsilon > 0$
Initialize: $\mathbf{w} = (w_1, \dots, w_d)^\top \in \mathbb{R}^d$ (e.g., $\mathbf{w} = \mathbf{0}$), $\mathbf{z} = (z_1, \dots, z_d)^\top \in \mathbb{R}^d$

Loop for each episode:
    Initialize $S$
    Choose $A \sim \pi(\cdot|S)$ or $\varepsilon$-greedy according to $\hat{q}(S, \cdot, \mathbf{w})$
    $\mathbf{z} \leftarrow \mathbf{0}$
    Loop for each step of episode:
        Take action $A$, observe $R, S'$
        $\delta \leftarrow R$
        Loop for $i$ in $\mathcal{F}(S, A)$:
            $\delta \leftarrow \delta - w_i$
            $z_i \leftarrow z_i + 1$                                   (accumulating traces)
            or $z_i \leftarrow 1$                                      (replacing traces)
        If $S'$ is terminal then:
            $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\mathbf{z}$
            Go to next episode
        Choose $A' \sim \pi(\cdot|S')$ or $\varepsilon$-greedy according to $\hat{q}(S', \cdot, \mathbf{w})$
        Loop for $i$ in $\mathcal{F}(S', A')$: $\delta \leftarrow \delta + \gamma w_i$
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\mathbf{z}$
        $\mathbf{z} \leftarrow \gamma\lambda\mathbf{z}$
        $S \leftarrow S'$; $A \leftarrow A'$
```

Figure 39: SARSA(λ) with binary features and linear function approximation for estimating $w^T \approx q\pi or q*$

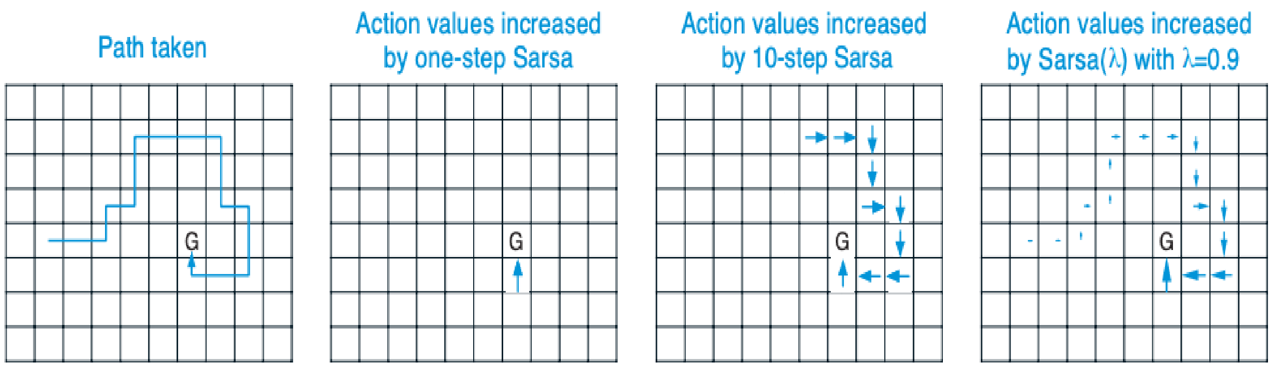

Path taken
Action values increased by one-step Sarsa
Action values increased by 10-step Sarsa
Action values increased by Sarsa(λ) with λ=0.9
G

Figure 40: 4 SARSA algorithms tackling the gridworld task

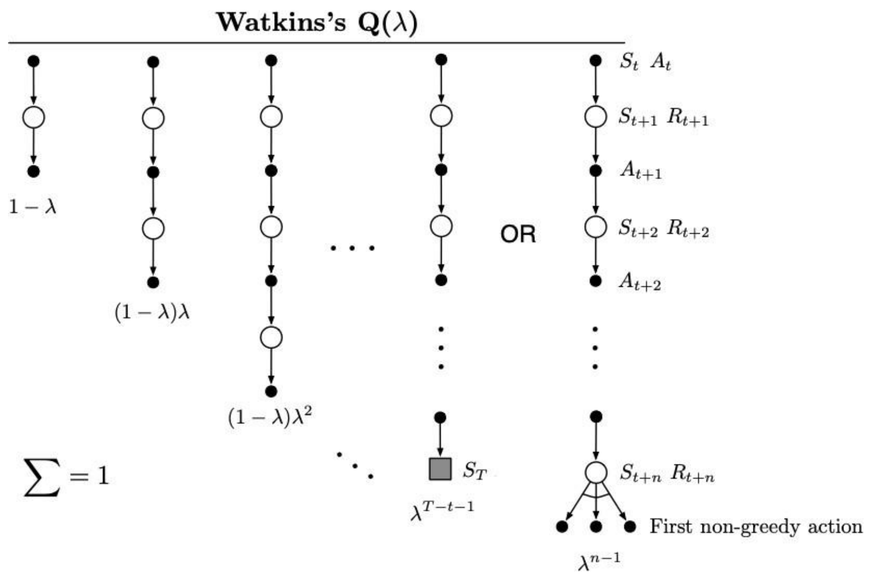


Figure 41: The backup diagram for Watkins's Q(λ). The series of component updates ends either with the end of the episode or with the first non-greedy action, whichever comes first.

Variable λ and γ

We can amend λ, the eligibility trace parameter, and γ, the discounting parameter, to vary with state rather than be constants throughout an episode. That is, λ and γ are now state-dependent and denoted λ_t and γ_t. This creates a further layer of generalisation not yet seen. The returnis now defined more generally as

$$
\begin{aligned}
G_t &\doteq R_{t+1} + \gamma_{t+1} G_{t+1} && (99)\\
&= R_{t+1} + \gamma_{t+1} R_{t+2} + \gamma_{t+1}\gamma_{t+2} R_{t+3} + \cdots && (100)\\
&= \sum_{k=t}^{\infty} \left(\prod_{i=t+1}^{k} \gamma_i \right) R_{k+1} && (101)\\
& && (102)
\end{aligned}
$$

Off-policy Traces with Control Variates

To generalise to the off-policy case we need to bring back ideas from importance sampling. This section, and associated derivations, are long, but eventually we arrive at a new eligibility trace

$$\mathbf{z}_t \doteq \gamma_t \lambda_t \rho_t \mathbf{z}_{t-1} + \nabla \hat{q}(S_t, A_t, \mathbf{w}_t), \tag{103}$$

where $\rho_t = \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$ is the single-step importance sampling ratio.

Watkins's Q(λ) to Tree-Backup(λ)

Q-learning and Tree-backups can be extended to the eligibility trace setting too, and are called Watkins's Q(λ) and Tree-backup(λ) respectively. Their backup diagrams are shown in Figures 41 and 42. In Q(λ) we continue to include weighted *n*-step rewards whilst greedy actions are selected, but when a non-greedy action is selected on-policy the cumulative reward terminates with an expectation at that state.

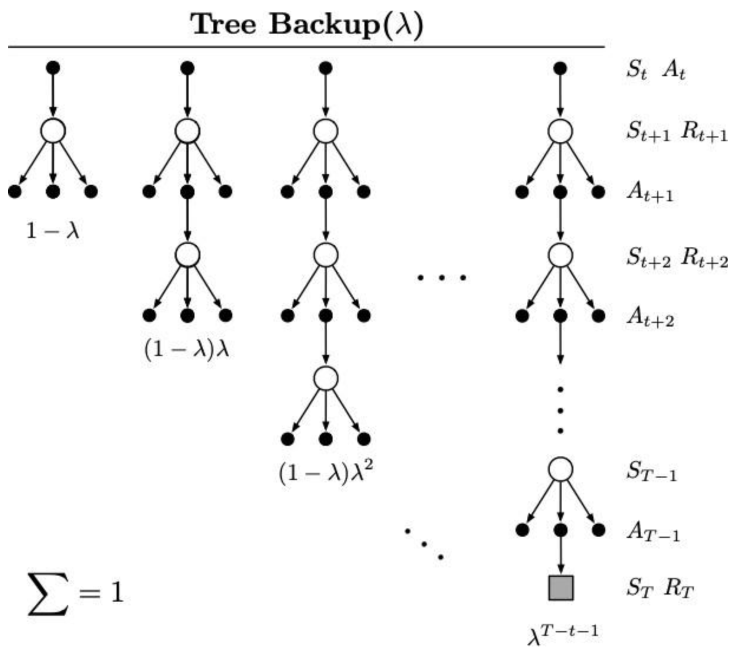


Figure 42: The backup diagram for the λ version of the Tree Backup algorithm.

Implementation Issues

In the tabular setting, it would appear that every state-value instance would need to be updated at every timestep. This is a problem for implementation on traditional serial computers. Fortu- nately, for typical values of λ and γ, almost all state have eligibility traces near zero. In practice, therefore, we only need to update the states for which the update is some margin bigger than zero, decreasing the complexity of the system. Function approximation, like ANNs, reduce the complexity of eligibility-based algorithms too.

Summary

Eligibility traces allow us to walk the continuum between one-step methods and Monte Carlo in a similar way to the *n*-step methods discussed in chapter 7. They are, however, more general, and learn value functions more quickly.

Monte Carlo methods are good in non-Markov environments, because their rewards are unbiased by bootstrapping. Eligibility traces have the same property. They are mostuseful for environments that have long-delayed rewards and are non-Markov.

Eligibility traces use significantly more computation than *n*-step methods, but in return they offer significantly faster learning, particularly when rewards are delayed by many steps. Thus they can be particularly effective in **low data settings**, and not effective when data can be acquired cheaply via a simulator. When data is cheap, the challenge is processing that data as quickly as possible, and typically, that means using one-step methods like Q-learning.

λ-return is forward view and TD(λ) is backward view.

Policy Gradient Methods

Instead of learning action-value estimates, and deriving a policy thereafter, we will now param- eterize a policy directly depending on its performance. Our policy is now $\pi(a\ s, \boldsymbol{\theta})$ representing the probability of taking action a in state s given parameter vector $\theta \in \mathrm{R}^{d'}$. *Actor-critic* meth- ods learn both a parameterised policy (the actor) and a parameterised value function estimate (the critic)

The policy parameter update is therefore

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \nabla \boldsymbol{J}(\hat{\boldsymbol{\theta}}_t) \tag{104}$$

where $\boldsymbol{J}(\hat{\boldsymbol{\theta}}_t)$ is a stochastic estimate whose expectation approximates the gradient of the performance measure with respect to its argument $\boldsymbol{\theta}_t$.

Policy Approximation and its Advantages

The policy can be parameterised in any way as long as $\pi(a|s, \boldsymbol{\theta})$ is differentiable.

If the action-space is discrete and not too large, then a natural and common kind of parameterisation is to form numerical preferences $h(s, a, \boldsymbol{\theta}) \in \mathrm{R}$ for each state-action pair. The actions with the highest preferences in each state are given the highest probabilities of being selected, for example, according to an exponential soft-max distribution:

$$\pi(a|s, \boldsymbol{\theta}) \doteq \frac{e^{h(s,a,\boldsymbol{\theta})}}{\sum_b e^{h(s,b,\boldsymbol{\theta})}} \tag{105}$$

We call this kind of policy parameterisation *soft-max in action preferences*. An advantage of parameterising policies according to soft-max action preferences is that the approximate policy can approach a determinisitc policy, whereas with ϵ-greedy action selection over action values there is always an ϵ probability of selecting a random action.

A second advantage of soft-max action preferences is that we can now select actions with arbitrary probabilities, rather than the greedy action selected most of the time and all other actions given probability of selection $\epsilon/|A|$. This is useful in environments with imperfect information, where it is useful to act stochastically e.g. when bluffing in pokerit is useful to do so randomly to unnerve an opponent.

Often the most important reason for using a policy-gradient method is that we can inject prior knowledge about the system in the RL agent.

The Policy Gradient Theorem

We define performance in the episodic case as

.

$$J(\boldsymbol{\theta}) = v_{\pi\theta}(s_0), \tag{106}$$

where $v_{\pi\theta}(s_0)$ is the true value function for $\pi_{\boldsymbol{\theta}}$, the policy determined by $\boldsymbol{\theta}$. And s_0 is some non-random start state. The policy gradient theorem then establishes that

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\boldsymbol{\theta}) \tag{107}$$

where μ is the on-policy distribution under π as discussed previously i.e. the theorem is a sum over states weighted by how often the states occur under the target policy π. We can use this to approximate gradient ascent without access to the derivative of the state distribution.

REINFORCE: Monte Carlo Policy Gradient

REINFORCE is a method for updating our policy parameters based on episodic returns from states G_t. We update the parameters in the direction of actions that yielded highest rewards. Recall from Equation 107 that the policy gradient theorem is a weighted sum of over states which is the same as an expectation over states:

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\boldsymbol{\theta}) \tag{108}$$

$$= \mathbb{E}_\pi \left[\sum_a q_\pi(S_t,a) \nabla \pi(a,S_t,\boldsymbol{\theta}) \right] \tag{109}$$

We can arrive at REINFORCE by replacing the sum over the random variable's possible values by an expectation under π, and then sampling the expectation. We multiply and divide the summed terms by $\pi(a|S_t,\boldsymbol{\theta})$:

$$\nabla J(\boldsymbol{\theta}) \propto \mathbb{E}_\pi \left[\sum_a \pi(a|S_t,\boldsymbol{\theta}) q_\pi(S_t,a) \frac{\nabla \pi(a,S_t,\boldsymbol{\theta})}{\pi(a,S_t,\boldsymbol{\theta})} \right] \tag{110}$$

$$= \mathbb{E}_\pi \left[q_\pi(S_t,A_t) \frac{\nabla \pi(A_t,S_t,\boldsymbol{\theta})}{\pi(A_t,S_t,\boldsymbol{\theta})} \right] \quad \text{(replacing } a \text{ by the sample } A_t \sim \pi) \tag{111}$$

$$= \mathbb{E}_\pi \left[G_t \frac{\nabla \pi(A_t,S_t,\boldsymbol{\theta})}{\pi(A_t,S_t,\boldsymbol{\theta})} \right] \quad \text{(because } \mathbb{E}_\pi [G_t|S_t,A_t] = q_\pi(S_t,A_t)) \tag{112}$$

$$\tag{113}$$

The last expression is what we need: a quantity that can be sampled on each time step whose expectation is proportional to the gradient. We now arrive at the REINFORCE update

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha G_t \frac{\nabla \pi(A_t,S_t,\boldsymbol{\theta})}{\pi(A_t,S_t,\boldsymbol{\theta})} \tag{114}$$

This update makes intuitive sense. The gradient term represents the direction in parameter space that most increases the probability of taking that same action again in the future. In effect, the gradient distil the policy approximation down to the weights that were most responsible for the action taken, from there we can affect these weights based on how much we valued the action. If the action yielded high reward G_t, then we boost the weights in proportion to the reward, and if this action has low probability of being taken under the current policy (the denominator $\pi(A_t, S_t, \boldsymbol{\theta})$) then the weights are further boosted. Alternatively, if the action is taken often, and yields low reward, we make little adjustment to the weights, lowering their impact on the policy approximation in the long run as other weights receive boosts.

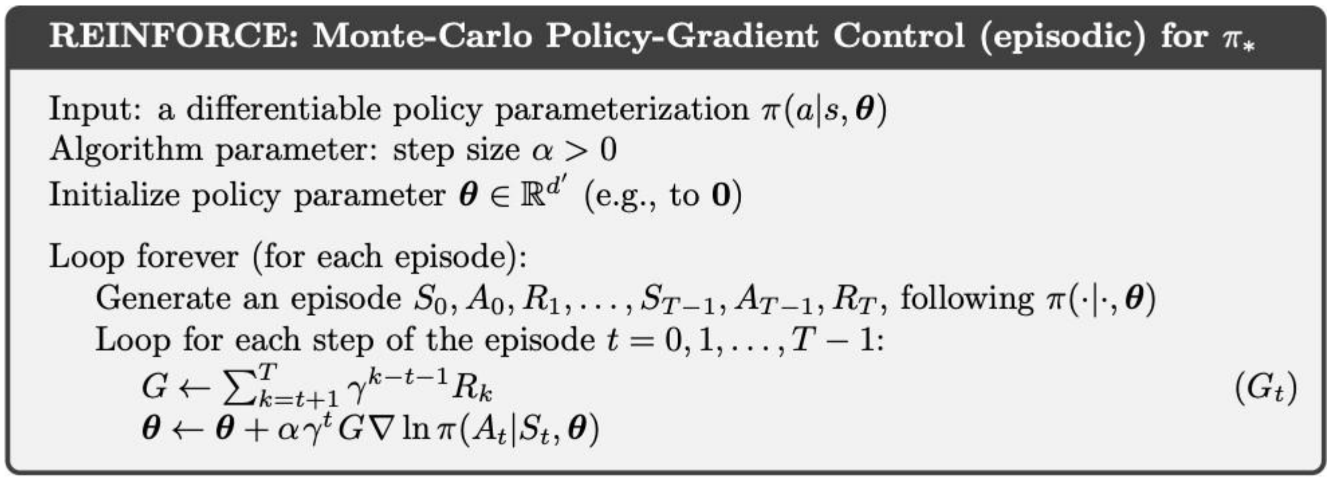
Note this is a monte carlo algorithm, using complete returns as updates, meaning it is only well defined in the episodic case, and can have high variance causing slow learning. The pseudocode for REINFORCE is given in Figure 43.

REINFORCE with Baseline

The policy gradient theorem can be generalized to include a comparison of the action value to an arbitrary *baseline* $b(s)$:

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a \left(q_\pi(s,a) - b(s)\right) \nabla \pi(a|s,\boldsymbol{\theta}) \tag{115}$$

We do this because it can help us reduce the variance of our results, which speeds learning. By comparing our observed result with some state-dependent baseline, we get a feel for how



REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

```
Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
    Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
    Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
        $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$                    $(G_t)$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$
```

Figure 43: Pseudocode for REINFORCE: monte carlo policy-gradient control (episodic) for $\pi*$

different the observed value is from what we expected. Thus, we arrive at a lower variance REINFORCE update as

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \left(G_t - b(S_t)\right) \frac{\nabla \pi(A_t, S_t, \boldsymbol{\theta})}{\pi(A_t, S_t, \boldsymbol{\theta})} \tag{116}$$

The natural baseline is an estimate of the state value, $\hat{v}(S_t, \mathbf{w})$, which we can learn as discussed in previous chapters. This method significantly speeds up learning as shown in Figure44.

Actor-Critic Methods

The REINFORCE method uses the state-value function as a baseline for comparing true return from a state with what we expected the return to be. Because this comparison is made prior to action selection, we cannot use it to directly evaluate actions. In actor- critic methods, the state-value function is applied also to the *second* state of the transition, which, when discounted and added to the one-step reward, constitutes the one-step return

$G_{t:t+1}$.

When the state-value function is used in this way it is called the *critic* and the policy is the *actor*.

One-step actor-critic methods replace the full return of REINFORCE with the one-step return (and use a learned state-value function as the baseline) as follows:

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \left(G_{t:t+1} - \hat{v}(S_t, \mathbf{w})\right) \frac{\nabla \pi(A_t, S_t, \boldsymbol{\theta})}{\pi(A_t, S_t, \boldsymbol{\theta})} \tag{117}$$

$$= \boldsymbol{\theta}_t + \alpha \left(R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w})\right) \frac{\nabla \pi(A_t, S_t, \boldsymbol{\theta})}{\pi(A_t, S_t, \boldsymbol{\theta})} \tag{118}$$

$$= \boldsymbol{\theta}_t + \alpha \delta_t \frac{\nabla \pi(A_t, S_t, \boldsymbol{\theta})}{\pi(A_t, S_t, \boldsymbol{\theta})} \tag{119}$$

The pseudocode for one-step actor-critic is given in Figure 45, and it can be extended to include eligibility traces from Chapter 11 as shown in Figure 46.

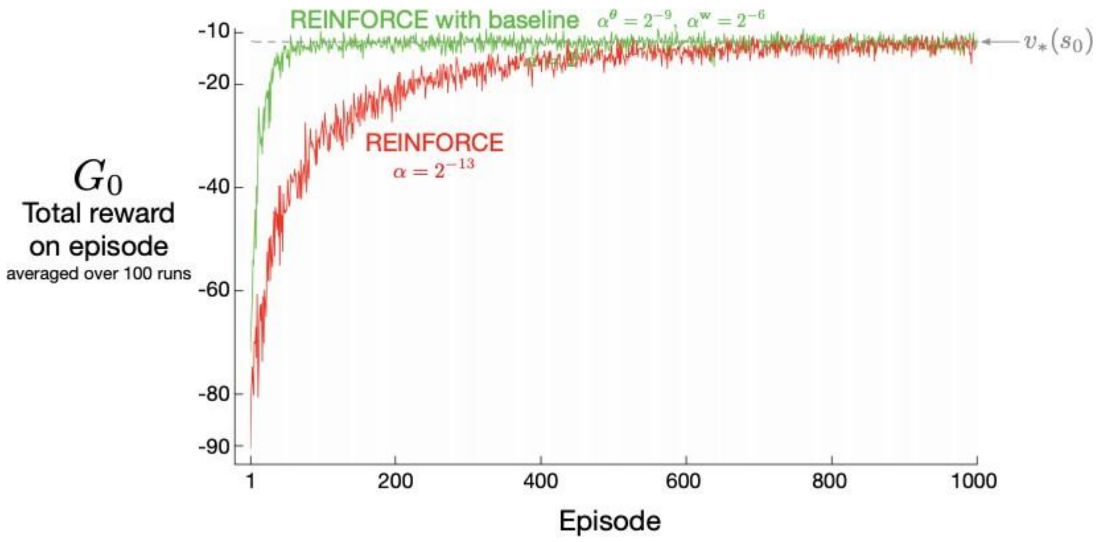
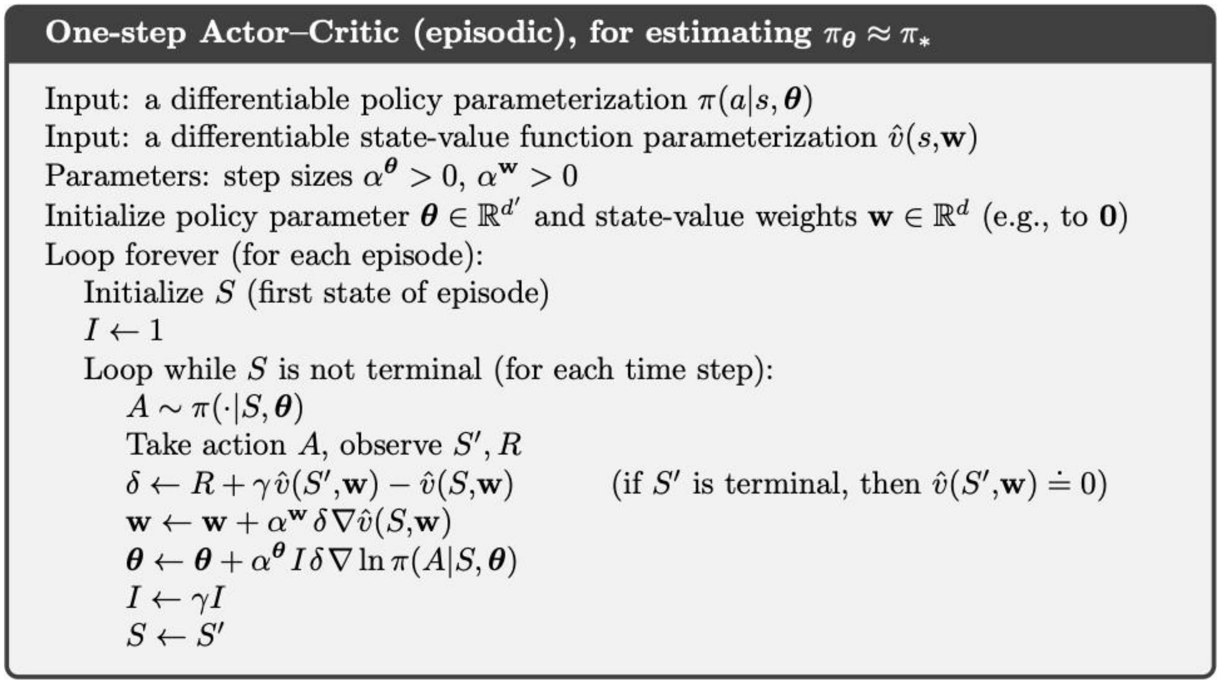


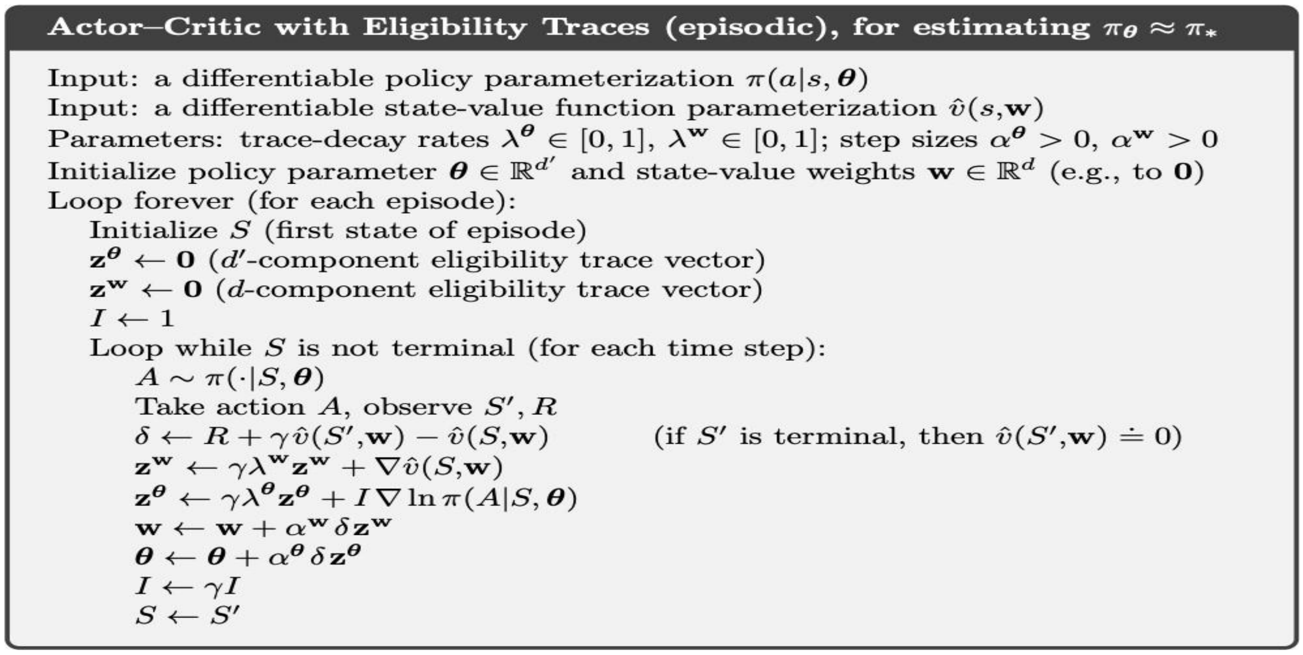
Figure 44: Adding a baseline to REINFORCE can make it learn much faster, as illustrated here on the short-corridor gridworld (Example 13.1). The step size used here for plain REINFORCE is that at which it performs best.



```
One-step Actor–Critic (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)
Loop forever (for each episode):
    Initialize $S$ (first state of episode)
    $I \leftarrow 1$
    Loop while $S$ is not terminal (for each time step):
        $A \sim \pi(\cdot|S, \boldsymbol{\theta})$
        Take action $A$, observe $S', R$
        $\delta \leftarrow R + \gamma \hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$          (if $S'$ is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S,\mathbf{w})$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla \ln \pi(A|S, \boldsymbol{\theta})$
        $I \leftarrow \gamma I$
        $S \leftarrow S'$
```

Figure 45: Pseudocode for one-step actor-critic



```
Actor–Critic with Eligibility Traces (episodic), for estimating $\pi_{\boldsymbol\theta} \approx \pi_*$

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol\theta)$
Input: a differentiable state-value function parameterization $\hat v(s,\mathbf{w})$
Parameters: trace-decay rates $\lambda^{\boldsymbol\theta} \in [0,1]$, $\lambda^{\mathbf{w}} \in [0,1]$; step sizes $\alpha^{\boldsymbol\theta} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol\theta \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)
Loop forever (for each episode):
   Initialize $S$ (first state of episode)
   $\mathbf{z}^{\boldsymbol\theta} \leftarrow \mathbf{0}$ ($d'$-component eligibility trace vector)
   $\mathbf{z}^{\mathbf{w}} \leftarrow \mathbf{0}$ ($d$-component eligibility trace vector)
   $I \leftarrow 1$
   Loop while $S$ is not terminal (for each time step):
      $A \sim \pi(\cdot|S,\boldsymbol\theta)$
      Take action $A$, observe $S', R$
      $\delta \leftarrow R + \gamma\hat v(S',\mathbf{w}) - \hat v(S,\mathbf{w})$        (if $S'$ is terminal, then $\hat v(S',\mathbf{w}) \doteq 0$)
      $\mathbf{z}^{\mathbf{w}} \leftarrow \gamma\lambda^{\mathbf{w}}\mathbf{z}^{\mathbf{w}} + \nabla\hat v(S,\mathbf{w})$
      $\mathbf{z}^{\boldsymbol\theta} \leftarrow \gamma\lambda^{\boldsymbol\theta}\mathbf{z}^{\boldsymbol\theta} + I\nabla\ln\pi(A|S,\boldsymbol\theta)$
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}}\delta\mathbf{z}^{\mathbf{w}}$
      $\boldsymbol\theta \leftarrow \boldsymbol\theta + \alpha^{\boldsymbol\theta}\delta\mathbf{z}^{\boldsymbol\theta}$
      $I \leftarrow \gamma I$
      $S \leftarrow S'$
```

Figure 46: Pseudocode for one-step actor-critic with eligibility traces

Policy Gradient for Continuing Problems

As discussed in Chapter 10, for continuing problems without episode boundaries we need todefine the performance in terms of average rate of reward per time step:

$$
\begin{aligned}
J(\boldsymbol\theta) &\doteq r(\pi) && (120)\\
&= \lim_{t\to\infty} \mathbb{E}[R_t|S_0, A_{0:t-1} \sim \pi] && (121)\\
&= \sum_s \mu(s) \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)r && (122)\\
& && (123)
\end{aligned}
$$

We need to remember that in the continuing case, we define values $v_\pi(s) \doteq \mathbb{E}[G_t, S_t = s]$ and $q_\pi(s,a) \doteq \mathbb{E}[G_t, S_t = s, A_t = a]$ w.r.t the differential return:

$$G_t \doteq R_{t+1} - r(\pi) + R_{t+2} - r(\pi) + R_{t+3} - r(\pi) + \cdots \tag{124}$$

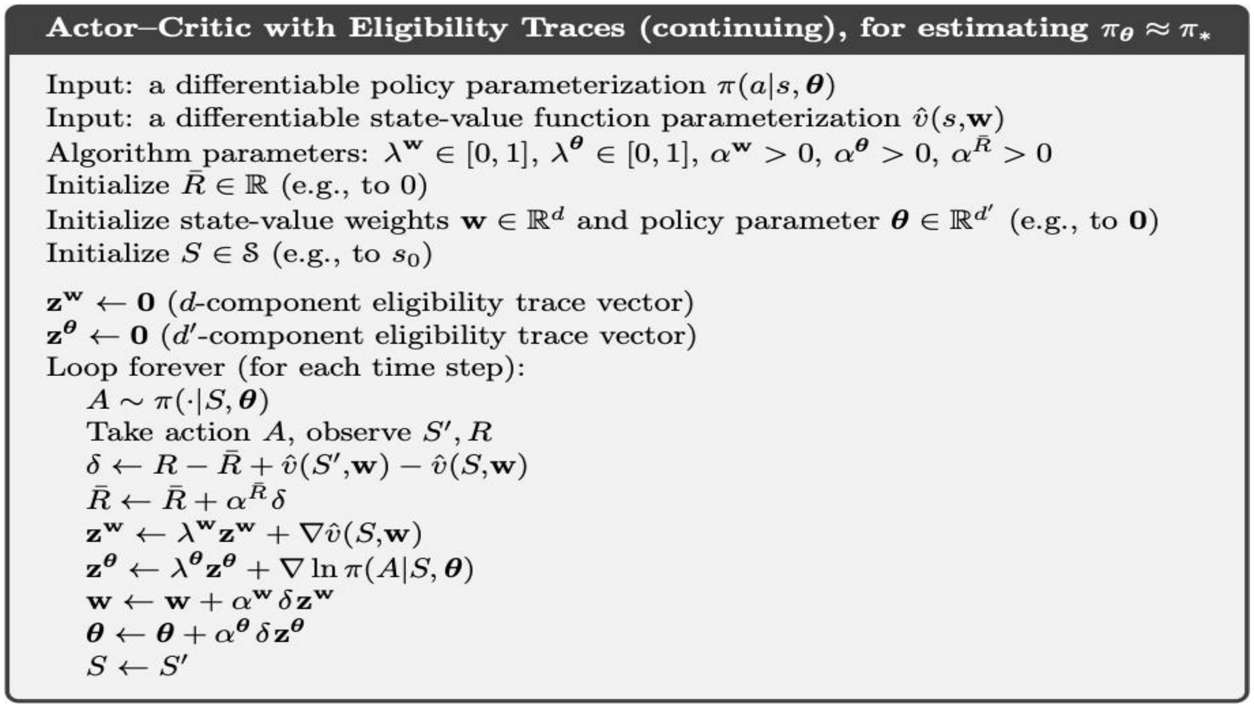
Pseudocode for the continuing case is given in Figure 47.

Policy Parameterisation for Continuous Actions

For state-spaces with huge numbers of actions, or continuous action spaces with infinite numbers of actions, we do not need to learn the probability of selecting each of these many actions, we can instead learn the parameters of the distribution over actions given a state. The probability density function for the normal distribution is conventionally written

$$p(x) \doteq \frac{1}{\sigma\sqrt{2\pi}} exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \tag{125}$$

where μ and σ are the mean and standard deviation of the normal distribution. Note $p(x)$ is the density of the probability, *not* the probability itself. As such it can be greater than 1; it is



Actor–Critic with Eligibility Traces (continuing), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

```
Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Algorithm parameters: $\lambda^{\mathbf{w}} \in [0,1]$, $\lambda^{\boldsymbol{\theta}} \in [0,1]$, $\alpha^{\mathbf{w}} > 0$, $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\bar{R}} > 0$
Initialize $\bar{R} \in \mathbb{R}$ (e.g., to 0)
Initialize state-value weights $\mathbf{w} \in \mathbb{R}^d$ and policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)
Initialize $S \in \mathcal{S}$ (e.g., to $s_0$)

$\mathbf{z}^{\mathbf{w}} \leftarrow \mathbf{0}$ ($d$-component eligibility trace vector)
$\mathbf{z}^{\boldsymbol{\theta}} \leftarrow \mathbf{0}$ ($d'$-component eligibility trace vector)
Loop forever (for each time step):
    $A \sim \pi(\cdot|S,\boldsymbol{\theta})$
    Take action $A$, observe $S', R$
    $\delta \leftarrow R - \bar{R} + \hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$
    $\bar{R} \leftarrow \bar{R} + \alpha^{\bar{R}}\delta$
    $\mathbf{z}^{\mathbf{w}} \leftarrow \lambda^{\mathbf{w}}\mathbf{z}^{\mathbf{w}} + \nabla\hat{v}(S,\mathbf{w})$
    $\mathbf{z}^{\boldsymbol{\theta}} \leftarrow \lambda^{\boldsymbol{\theta}}\mathbf{z}^{\boldsymbol{\theta}} + \nabla \ln \pi(A|S,\boldsymbol{\theta})$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}}\delta\mathbf{z}^{\mathbf{w}}$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}}\delta\mathbf{z}^{\boldsymbol{\theta}}$
    $S \leftarrow S'$
```

Figure 47: Pseudocode for one-step actor-critic with eligibility traces, continuing case

the area under the graph that must sum to 1. One can take the integral between two values of *x* to arrive at the probability of *x* falling within that range. The policy parameterisation therefore becomes:

$$\pi(a|s,\boldsymbol{\theta}) \doteq \frac{1}{\sigma(s,\boldsymbol{\theta})\sqrt{2\pi}} exp\left(-\frac{(x-\mu(s,\boldsymbol{\theta}))^2}{2\sigma(s,\boldsymbol{\theta})^2}\right) \tag{126}$$

where $\mu : \mathcal{S} \times \mathbb{R}^{d'} \to \mathbb{R}$ and $\sigma : \mathcal{S} \times \mathbb{R}^{d'} \to \mathbb{R}^+$ i.e. they are both matrices of with dimensions equal to the number of states times the number of dimensions of the feature vector defining the policy. To from the policy approximator we need to split the policy's parameter vector into two parts, $\boldsymbol{\theta} = [\boldsymbol{\theta}_\mu, \boldsymbol{\theta}_\sigma]^T$. The mean can be approximated as a linear function, but the standard deviation must always be positive and is better approximated as the exponential of a linear function. Thus

$$\mu(s,\boldsymbol{\theta}) \doteq \boldsymbol{\theta}_\mu^T \mathbf{x}_\mu(s) \quad \text{and} \quad \sigma(s,\boldsymbol{\theta}) \doteq exp\left(\boldsymbol{\theta}_\sigma^T \mathbf{x}_\sigma(s)\right) \tag{127}$$

Summary

Prior to this chapter, actions had been selected by consulting an action-value function that informed us of how valuable actions were in given states at reaching our goal. Here, we instead parameterised policies directly, updating our approximations by reviewing the rewards received after taking some actions.

There are several advantages to doing this:

They can learn specific probabilities of taking actions (rather than sharing|A¢/)probability with non-greedy actions as per ϵ-greedy methods)

They can learn appropriate levels of exploration and then approach deterministic policies in the long run

They can naturally handle continuous action spaces by learning distributions over actions given a state

The policy gradient theorem provides a theoretical foundation for all policy gradient meth- ods and gives an exact formula for how performance is affected by the policy parameter that does not involve derivatives of the state distribution.

The REINFORCE method is the policy gradient theorem in action using monte carlo returns

A baseline (usually our current estimate for the state-value) can be used in conjunction with REINFORCE to reduce the variance of the output and speed learning.

Actor-critic methods assess the policy's action selection by comparing its one-step reward with the value function at the next timestep. This introduces bias into the actor's gradient estimates, but is often desirable for the same reason that bootstrapping TD methods are often superior to Monte Carlo methods (substantially reduced variance).