Study Notes on Reinforcement Learning: An Introduction (2nd Ed.)

Independent personal study notes accompanying Sutton & Barto (2018)

Chizoba Obasi

November 2026

Table of contents

Preface **1**
Coverage . . . 1
Downloads . . . 1
How to Read These Notes . . . 1
Key Notation . . . 2
Reference . . . 4

I. Tabular Solution Methods 5

1. Introduction **6**
1.1. Reinforcement Learning . . . 6
1.2. Examples . . . 7
1.3. Elements of Reinforcement Learning . . . 7
1.4. Limitations and Scope . . . 8
1.5. An Extended Example: Tic-Tac-Toe . . . 9
1.6. Summary . . . 10
1.7. Early History of Reinforcement Learning . . . 10

2. Multi-Armed Bandits **13**
2.1. The k-armed Bandit Problem . . . 13
2.2. Action-value Methods . . . 13
2.3. The 10-armed Testbed . . . 14
2.4. Incremental Implementation . . . 14
2.5. Tracking a Nonstationary Problem . . . 14
2.6. Optimistic Initial Values . . . 16
2.7. Upper-Confidence-Bound (UCB) Action Selection . . . 16
2.8. Gradient Bandit Algorithms . . . 17
2.9. Associative Search (Contextual Bandits) . . . 19
2.10. Summary . . . 19

3. Finite Markov Decision Processes **21**
3.1. The Agent-Environment Interface . . . 21
3.2. Goals & Rewards . . . 22
3.3. Returns & Episodes . . . 22
3.4. Unified Notation for Episodic & Continuing Tasks . . . 23
3.5. Policies, Value Functions & Bellman Equations . . . 24
3.6. Optimal Policies & Optimal Value Functions . . . 26
3.7. Optimality & Approximation . . . 29
3.8. Summary . . . 29
3.9. Glossary . . . 30

4. Dynamic Programming **31**
4.1. Policy Evaluation (Prediction) . . . 31
4.2. Policy Improvement . . . 32
4.3. Policy Iteration . . . 33
4.4. Value Iteration . . . 33
4.5. Asynchronous Dynamic Programming . . . 33
4.6. Generalized Policy Iteration (GPI) . . . 33
4.7. Efficiency of Dynamic Programming . . . 34
4.8. Summary . . . 34

5. Monte Carlo Methods **35**
5.1. Monte Carlo Prediction . . . 35
5.2. Monte Carlo Estimation of Action Values . . . 36
5.3. Monte Carlo Control . . . 36
5.4. Monte Carlo Control Without Exploring Starts . . . 37
5.5. Off-Policy Prediction via Importance Sampling . . . 38
5.6. Incremental Implementation . . . 40
5.7. Off-Policy Monte Carlo Control . . . 40
5.8. Discounting-aware Importance Sampling . . . 40
5.9. Per-decision Importance Sampling . . . 41
5.10. Summary . . . 42

6. Temporal-Difference Learning **43**
6.1. TD Prediction . . . 43
6.2. Advantages of TD Prediction Methods . . . 45
6.3. Optimality of TD(0) . . . 45
6.4. Sarsa: On-policy TD Control . . . 46
6.5. Q-learning: Off-Policy TD Control . . . 48
6.6. Expected Sarsa . . . 49
6.7. Maximization Bias & Double Learning . . . 50
6.8. Games, Afterstates & Other Special Cases . . . 51
6.9. Summary . . . 51

7. n-step Bootstrapping **52**
7.1. n-step TD Prediction, TD(λ) . . . 52
7.2. n-step Sarsa, Sarsa(λ) . . . 53
7.3. n-step Off-policy Learning . . . 55
7.4. Per-decision Methods with Control Variates . . . 56
7.5. Off-policy Learning Without Importance Sampling: The n-step Tree Backup Algorithm . . . 57
7.6. A Unifying Algorithm: n-step $Q(\sigma)$ . . . 58
7.7. Summary . . . 59

8. Planning & Learning with Tabular Methods **61**
8.1. Models & Planning . . . 61
8.2. Dyna: Integrated Planning, Acting, and Learning . . . 62
8.3. When the Model is Wrong . . . 63
8.4. Prioritized Sweeping . . . 64
8.5. Expected vs Sample Updates . . . 64
8.6. Trajectory Sampling . . . 66
8.7. Real-Time Dynamic Programming (RTDP) . . . 67
8.8. Planning at Decision Time . . . 69
8.9. Heuristic Search . . . 69
8.10. Rollout Algorithms . . . 69
8.11. Monte Carlo Tree Search (MCTS) . . . 71
8.12. Summary . . . 74
8.13. Summary of Part I: Dimensions . . . 74

II. Approximate Solution Methods **77**

9. On-Policy Prediction with Approximation **78**
9.1. Value-Function Approximation . . . 78
9.2. The Prediction Objective ($\overline{\text{VE}}$) . . . 78
9.3. Stochastic-Gradient & Semi-Gradient Methods . . . 80
9.4. Linear Methods . . . 81
9.5. Feature Construction for Linear Methods . . . 83
9.6. Selecting Step-Size Parameters Manually . . . 86
9.7. Nonlinear Function Approximation: Artificial Neural Networks (ANNs) . . . 86

9.8. Least-Squares TD (LSTD) . . . 87
9.9. Memory-based Function Approximation . . . 88
9.10. Kernel-based Function Approximation . . . 89
9.11. Looking Deeper at On-Policy Learning: Interest & Emphasis . . . 90
9.12. Summary . . . 90

10. On-Policy Control with Approximation **91**
10.1. Episodic Semi-gradient Control . . . 91
10.2. Semi-gradient n-step Sarsa . . . 91
10.3. Average Reward: A New Problem Setting for Continuing Tasks . . . 92
10.4. Deprecating the Discounted Setting . . . 93
10.5. Differential Semi-gradient n-step Sarsa . . . 94
10.6. Summary . . . 94

11. Off-Policy Methods with Approximation **95**
11.1. Semi-gradient Methods . . . 95
11.2. Examples of Off-Policy Divergence . . . 96
11.3. The Deadly Triad . . . 99
11.4. Linear Value-Function Geometry . . . 99
11.5. Gradient Descent in the Bellman Error . . . 102
11.6. The Bellman Error is Not Learnable . . . 103
11.7. Gradient-TD Methods . . . 104
11.8. Emphatic-TD Methods . . . 106
11.9. Reducing Variance . . . 107
11.10Summary . . . 107

12. Eligibility Traces **109**
12.1. The λ-return . . . 109
12.2. TD(λ) . . . 111
12.3. n-step Truncated λ-return Methods . . . 114
12.4. Redoing Updates: Online λ-return Algorithm . . . 115
12.5. True Online TD(λ) . . . 116
12.6. Dutch Traces in Monte Carlo Learning . . . 117
12.7. Sarsa(λ) . . . 118
12.8. Variable λ and γ . . . 120
12.9. Off-Policy Traces with Control Variates . . . 121
12.10Watkins's Q(λ) to Tree-Backup(λ) . . . 123
12.11Stable Off-Policy Methods with Traces . . . 124
12.12Implementation Issues . . . 126
12.13Conclusions . . . 127

13. Policy Gradient Methods **128**
13.1. Policy Approximation & Its Advantages . . . 128
13.2. The Policy Gradient Theorem . . . 129
13.3. REINFORCE: Monte Carlo Policy Gradient . . . 130
13.4. REINFORCE with Baseline . . . 131
13.5. Actor-Critic Methods . . . 132
13.6. Policy Gradient for Continuing Problems . . . 134
13.7. Policy Parametrization for Continuous Actions . . . 136
13.8. Summary . . . 137

Appendices **138**

A. Figure Gallery **138**

Notation Index **139**

Preface

These are independent personal study notes accompanying *Reinforcement Learning: An Introduction* by Richard S. Sutton and Andrew G. Barto (2nd Edition, MIT Press, 2018; book homepage). The textbook is distributed by its authors under the Creative Commons Attribution-NonCommercial-NoDerivs 2.0 Generic (CC BY-NC-ND 2.0) license.

These notes are an independent companion for study and review, not a substitute for the textbook. Mathematical notation follows standard reinforcement learning conventions from the book, and backup diagrams are included as figures.

Coverage

Chapter	Topic	Status
1	Introduction to RL	✓
2	Multi-Armed Bandits	✓
3	Finite Markov Decision Processes	✓
4	Dynamic Programming	✓
5	Monte Carlo Methods	✓
6	Temporal-Difference Learning	✓
7	n-step Bootstrapping	✓
8	Planning & Learning with Tabular Methods	✓
9	On-policy Prediction with Approximation	✓
10	On-policy Control with Approximation	✓
11	Off-policy Methods with Approximation	✓
12	Eligibility Traces	✓
13	Policy Gradient Methods	✓

Downloads

- PDF edition: all 13 chapters, built from the same sources on every push.
- EPUB edition: EPUB 3.3 with MathML equations (best in Apple Books, Kobo, Thorium, or Calibre).
- Anki deck: cloze cards with native MathJax, tagged by chapter and section. Import via Anki's *File* → *Import* (tab-separated, `#notetype:Cloze` is preset).

How to Read These Notes

These notes are meant to accompany the textbook, not replace it. They are most useful for:

- Quick review of key equations and algorithms before exams or projects.
- Checking notation and definitions across chapters.
- Following the logical progression of ideas from bandits through full RL.

Key Notation

Symbol	Meaning
s, s'	State, next state
a, a'	Action, next action
r, R	Reward
t	Discrete time step
T	Terminal time step
π	Policy
$\pi(a \mid s)$	Probability of taking action a in state s under policy π
$\pi(a\|s, \theta)$	Parametrized policy: probability of action a in state s under θ
π_*	Optimal policy
$b(a \mid s)$	Behavior policy
$v_\pi(s)$	State-value function under policy π
$v_{\pi_\theta}(s)$	True value function for policy π_θ
$q_\pi(s, a)$	Action-value function under policy π
$v_*(s)$	Optimal state-value function
$q_*(s, a)$	Optimal action-value function
$V(s)$	Estimated state-value
$Q(s, a)$	Estimated action-value
$\hat{v}(s, \mathbf{w})$	Approximate state-value function parameterized by $\mathbf{w}$
$\hat{q}(s, a, \mathbf{w})$	Approximate action-value function parameterized by $\mathbf{w}$
G_t	Return (cumulative discounted reward) at time t
$G_{t:h}$	Truncated return from t to horizon h
$G_{t:t+n}$	n-step return from t
G_t^λ	λ-return at time t
$G_t^{\lambda s}$	State-based λ-return (bootstraps from state values)
$G_t^{\lambda a}$	Action-based λ-return (bootstraps from action values)
$G_{t:h}^\lambda$	Truncated λ-return from t to horizon h
$J(\theta)$	Scalar performance measure (episodic: value of start state; continuing: average reward)
$\nabla J(\theta)$	Performance gradient w.r.t. θ
$\widehat{\nabla J(\theta_t)}$	Stochastic estimate of the performance gradient
$h(s, a, \theta)$	Numerical action preference for (s, a) under θ
λ	Trace-decay parameter, $\lambda \in [0, 1]$
λ_t	Variable trace-decay / termination function, $\lambda_t \doteq \lambda(S_t, A_t)$
λ^θ	Trace-decay rate for actor eligibility traces
$\lambda^{\mathbf{w}}$	Trace-decay rate for critic eligibility traces
γ	Discount factor, $\gamma \in [0, 1]$
γ_t	Variable discount factor, $\gamma_t \doteq \gamma(S_t)$
α	Step size (learning rate)
α^θ	Step size for policy parameter updates
$\alpha^{\mathbf{w}}$	Step size for state-value weight updates
β	Secondary step-size parameter (GTD2/TDC)
ε	Exploration parameter (ε-greedy)
n	Number of steps in n-step methods
h	Horizon (truncation point in TTD(λ))
d	Dimension of weight vector $\mathbf{w}$
d'	Dimension of policy parameter vector θ
θ	Policy parameter vector, $\theta \in \mathbb{R}^{d'}$
$\theta_\mu,\ \theta_\sigma$	Policy parameter subvectors for mean and std dev (continuous actions)

Key Notation

Symbol	Meaning
$\mathbf{w}$	Weight vector for function approximation, $\mathbf{w} \in \mathbb{R}^d$
$\mathbf{w}_t^h$	Weight vector at time t in sequence up to horizon h (online λ-return)
$\mathbf{w}_{\text{TD}}$	TD fixed point, $\mathbf{w}_{\text{TD}} = A^{-1}\mathbf{b}$
$\mathbf{v}$	Secondary weight vector (Gradient-TD cascade)
$\nabla \hat{v}(s, \mathbf{w})$	Gradient of $\hat{v}$ w.r.t. $\mathbf{w}$
$\mathbf{x}(s)$	Feature vector representing state s
$\mathbf{x}(s, a)$	Feature vector representing state-action pair (s, a)
$x_i(s)$	i-th component of feature vector, $x_i : S \to \mathbb{R}$
$\bar{\mathbf{x}}_t$	Average feature vector for S_t under the target policy (GQ(λ))
$\mathbf{z}_t$	Eligibility trace vector at time t, $\mathbf{z}_t \in \mathbb{R}^d$
$\mathbf{z}_t^b$	Accumulating eligibility trace for behavior policy (HTD(λ))
$\mathbf{z}^\theta$	Eligibility trace vector for the actor (d'-component)
$\mathbf{z}^\mathbf{w}$	Eligibility trace vector for the critic (d-component)
$\mathbf{F}_t$	Forgetting/fading matrix, $\mathbf{F}_t \doteq \mathbf{I} - \alpha \mathbf{x}_t \mathbf{x}_t^T$
$\mathbf{a}_t$	Auxiliary memory vector in dutch trace MC derivation
δ_t	TD error at time t
δ_t^s	State-based TD error (off-policy traces)
δ_t^a	Action-based / expected TD error (off-policy traces)
$\bar{\delta}_\mathbf{w}(s)$	Bellman error at state s
U_t	Update target at time t
ρ_t	Per-step importance sampling ratio, $\rho_t = \frac{\pi(A_t \mid S_t)}{b(A_t \mid S_t)}$
$\rho_{t:h}$	Importance sampling ratio from t to h
$\mu(s)$	On-policy state distribution
$\mu(s, \theta)$	Mean of action distribution for state s (continuous action policy)
$\mu_\pi(s)$	Steady-state distribution under policy π
$\sigma(s, \theta)$	Standard deviation of action distribution for state s (continuous action policy)
$r(\pi)$	Average reward under policy π
$\bar{R}_t$	Estimate of average reward $r(\pi)$ at time t
I	Discount accumulator in actor-critic, initialized to 1, updated $I \leftarrow \gamma I$
$\mathcal{I}_t$	Interest at time t, $\mathcal{I}_t \geq 0$
F_t	Followon trace at time t (Emphatic-TD(λ)), $F_t \geq 0$
M_t	Emphasis at time t (Emphatic-TD(λ)), $M_t \geq 0$
$\bar{V}_t(s)$	Expected approximate value of state s under target policy π
$\overline{\text{VE}}(\mathbf{w})$	Mean Squared Value Error
$\overline{\text{BE}}(\mathbf{w})$	Mean Squared Bellman Error
$\overline{\text{PBE}}(\mathbf{w})$	Mean Squared Projected Bellman Error
$\overline{\text{TDE}}(\mathbf{w})$	Mean Squared TD Error
$\overline{\text{RE}}(\mathbf{w})$	Mean Squared Return Error
Π	Projection operator onto the representable subspace
B_π	Bellman operator, $B_\pi : \mathbb{R}^{\lvert S \rvert} \to \mathbb{R}^{\lvert S \rvert}$
$\mathbf{B}$	Diagonal matrix with $\mu(s)$ on the diagonal, $\lvert S \rvert \times \lvert S \rvert$
$\mathbf{X}$	Feature matrix, $\lvert S \rvert \times d$, rows are $\mathbf{x}(s)^T$
A	Matrix $\mathbb{E}[\mathbf{x}_t(\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^T] \in \mathbb{R}^{d \times d}$ (linear TD)
$\mathbf{b}$	Vector $\mathbb{E}[R_{t+1}\mathbf{x}_t] \in \mathbb{R}^d$ (linear TD)
$\eta(s)$	Expected number of time steps spent in state s per episode
$h(s)$	Probability that an episode begins in state s

Symbol	Meaning
τ	Elapsed time since a state-action pair was last visited (Dyna-Q+)
k	Small bonus constant in Dyna-Q+
b	Branching factor
$N(s, a)$	Visit count for state-action pair (s, a) (MCTS)
$W(s, a)$	Total reward accumulated through (s, a) (MCTS)
c	Exploration constant in UCT, typically $\sqrt{2}$
$p(s', r \mid s, a)$	Transition probability
$\widehat{p}(s', r \mid s, a)$	Estimated/model transition probability
$\mathbb{E}_\pi[\cdot]$	Expectation under policy π

Reference

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction* (2nd ed.). MIT Press. Available at: http://incompleteideas.net/book/the-book-2nd.html

Part I.

Tabular Solution Methods

1. Introduction

1.1. Reinforcement Learning

Reinforcement learning (RL) is about learning **what to do** so as to maximize a numerical reward signal (how to map situations to actions). Unlike supervised learning, the agent isn't told which actions to take; it has to figure that out by trying things and seeing what happens. Two things make this distinctly hard: you have to search by trial and error, and rewards can be delayed; an action taken now might not pay off until much later.

RL is simultaneously a **problem**, a **class of solution methods**, and a **field of study**. Keeping these three separate matters a lot. Confusing the problem with the method is a surprisingly common source of muddy thinking.

Formally, RL is framed as the optimal control of an **incompletely-known Markov Decision Process (MDP)**. The full formalization comes in Chapter 3, but the core idea is to capture the essentials of an agent interacting with its environment over time to achieve a goal (sensation, action, and goal) in their simplest meaningful forms.

1.1.1. How RL Differs from Other ML Paradigms

- **Supervised learning** learns from labeled examples provided by an external supervisor. It's powerful, but in interactive settings you often can't get correct labeled examples for every situation the agent might face.
- **Unsupervised learning** finds hidden structure in unlabeled data. RL isn't trying to find structure, instead it's trying to maximize a reward signal. The two are genuinely different objectives.
- RL is therefore a **third paradigm**, distinct from both: no supervisor and no labels, only a reward signal from the environment.

1.1.2. The Exploration-Exploitation Tradeoff

One of the central tensions in RL that doesn't really arise in supervised or unsupervised learning is the tradeoff between:

- **Exploitation:** doing what you already know works, to rack up reward now.
- **Exploration:** trying new things, to potentially discover something better.

You can't do only one. Exploit too greedily and you'll miss better options. Explore too randomly and you waste time on things that don't pay off. Balancing the two has been studied for decades and remains an open problem.

1.1.3. The Complete Agent View

RL explicitly considers the **whole problem** of a goal-directed agent in an uncertain environment and not just isolated subproblems. Many ML approaches study useful pieces (supervised learning, planning, modeling) without asking how those pieces fit together in a real agent making real-time decisions. RL starts from the complete picture and works inward.

A "complete agent" doesn't have to mean a robot or organism, it could be a component inside a larger system, like a battery monitor inside a robot's control stack. The RL framework is general enough to cover both.

1.1.4. Connections Beyond Engineering

RL sits at the intersection of several fields:

- It addresses the classical **curse of dimensionality** from operations research and control theory through function approximation.

- It has deep ties to **psychology and neuroscience**: many core RL algorithms were inspired by biological learning, and RL has in turn produced influential models of animal learning and the brain's reward system (covered in Chapters 14 & 15).
- More broadly, RL is part of a trend in AI focusing back towards **simple, general principles** (search, learning, decision making) rather than the "vast bag of special-case tricks" view that dominated parts of the field in earlier decades.

1.2. Examples

The best way to get an intuition for what RL actually covers is through examples:

- A **chess player** decides on a move, weighing plans, counterplans, and immediate intuitions about which positions look promising.
- An **adaptive controller** tunes a petroleum refinery's parameters in real time, optimizing the yield/cost/quality tradeoff without rigidly following engineer-set points.
- A **gazelle calf** struggles to stand minutes after birth and is running at 20 mph half an hour later.
- A **mobile robot** decides whether to keep searching for trash or head back to its recharging station, based on battery level and past experience finding the charger.
- **Phil makes breakfast.** Even this mundane task involves a complex web of subgoals, conditional actions, eye movements, object manipulation, and real-time judgment calls; all in service of a goal Phil may not even be consciously tracking.

What do these have in common? A few things that are easy to miss because they're so fundamental:

- There's always an **active agent** making decisions in an environment, under uncertainty, toward a goal.
- Actions have **consequences that extend into the future**, they change the state of the environment, which changes what options are available later.
- Good decisions require **foresight**: accounting for indirect and delayed effects, not just immediate outcomes.
- The environment can't be fully predicted, so the agent has to **keep sensing and adapting**. Phil has to watch the milk as he pours it.
- Goals are **explicit and measurable**: the chess player knows if they won, the robot knows when the battery dies, Phil knows if breakfast was good.
- Critically, all these agents **improve with experience**: the chess player sharpens intuition, the gazelle gets more efficient, Phil streamlines his morning routine.

Prior knowledge (from evolution, design, or past experience) shapes what's easy to learn, but interaction with the environment is what lets the agent adapt to the specifics of its current task.

1.3. Elements of Reinforcement Learning

Beyond the agent and environment, every RL system has four core elements: three required, one optional.

1.3.1. Policy

The policy is the agent's **behavioral rulebook**, a mapping from states to actions. It can be as simple as a lookup table or as complex as a deep neural network running a search process. Policies can be deterministic or stochastic (assigning probabilities to actions). The policy alone is sufficient to fully determine an agent's behavior; it's the heart of the system.

1.3.2. Reward Signal

The reward signal defines the **goal**. At each timestep, the environment hands the agent a single number which is the reward. The agent's only objective is to maximize cumulative reward over time. Rewards are the immediate, raw signal of what's good or bad, analogous to pleasure and pain in biological systems. If an action leads to low reward, the policy should change. Rewards can also be stochastic.

1.3.3. Value Function

While rewards capture what's good **right now**, values capture what's good **in the long run**. The value of a state is the total reward an agent can expect to accumulate from that state onward. A state with a low immediate reward can still be highly valuable if it tends to lead to high-reward states down the line and vice versa.

The analogy: rewards are like pleasure and pain (immediate, visceral); values are like a more considered, farsighted judgment of how good a situation really is.

Rewards are primary. No rewards, no values. But when it comes to actually making decisions, **values are what matter**. We want actions that lead to high-value states, not just high immediate reward. The catch: rewards are given directly by the environment, but values have to be estimated and constantly re-estimated from experience. Efficient value estimation is arguably the most important insight from six decades of RL research.

1.3.4. Model (Optional)

A model mimics the environment. Given a state and action, it predicts the next state and reward. Models enable **planning**: reasoning about future situations before actually experiencing them. Methods that use models are called **model-based**; those that don't are **model-free** (pure trial and error). Chapter 8 covers systems that do both simultaneously: learning from experience, building a model, and using that model to plan.

Modern RL spans the full spectrum from raw trial-and-error to high-level deliberate planning.

1.4. Limitations and Scope

1.4.1. The Role of State

RL leans heavily on the concept of **state**: it's the input to the policy and value function, and both input and output of the model. Informally, state is whatever information is available to the agent about its environment at a given time. The formal definition comes in Chapter 3 via MDPs.

This book takes state as given, we assume some preprocessing system produces the state signal, and we don't tackle the question of how to construct, change, or learn it (except briefly in Section 17.3). This isn't because state representation is unimportant, it's very important, but because the focus here is on the **decision-making problem**: what action to take given whatever state signal you have.

1.4.2. Value Functions vs. Evolutionary Methods

Most methods in this book are built around estimating value functions, but that's not the only way to solve RL problems. **Evolutionary methods** involve genetic algorithms, genetic programming, simulated annealing, and similar optimization approaches. They never estimate value functions at all. Instead they:

- Run multiple static policies, each interacting with its own environment instance over an extended period.
- Keep the policies that earned the most reward, plus random variations of them.
- Repeat, generation after generation; analogous to biological evolution producing skilled behavior without any within-lifetime learning.

Evolutionary methods can work well when the policy space is small, when good policies are easy to stumble upon, or when the agent can't fully sense the environment state. But they have a fundamental limitation: **they don't learn from individual interactions**. They ignore which states were visited, which actions were taken, and what the consequences were; all information that can make search dramatically more efficient.

Methods that do learn from individual interactions can exploit the structure of the RL problem in ways evolutionary methods simply can't. While evolution and learning naturally complement each other, this book focuses on methods that learn online through interaction, and evolutionary methods on their own don't qualify.

1.5. An Extended Example: Tic-Tac-Toe

Tic-tac-toe is a useful teaching example precisely because it's simple enough to reason about clearly, yet rich enough to expose the real differences between RL and other approaches.

The setup: we're playing against an imperfect opponent, one who sometimes makes mistakes. Draws and losses are equally bad. How do we build a player that finds and exploits those imperfections?

1.5.1. Why Classical Methods Fall Short

- **Minimax** (game theory): assumes the opponent plays optimally. If our opponent is imperfect, minimax leaves wins on the table.
- **Dynamic programming**: can find an optimal strategy against any opponent, but requires a complete model of that opponent upfront (probabilities for every move in every state). We usually don't have that.
- **Learning the opponent model first, then applying DP**: closer, but now you're essentially doing RL anyway.

1.5.2. The Evolutionary Approach

An evolutionary method would search the space of policies directly: try many different "always do X in state Y" rulebooks, keep the ones that win most, generate variations, repeat. It can work, but it's wasteful: it only uses the final win/loss outcome of each game. Everything that happened during the game (which moves were critical, which states were visited) gets ignored. Credit is assigned to all moves equally, even ones that had nothing to do with the outcome.

1.5.3. The Value Function Approach

Here's how RL handles it:

1. Build a table with one entry per board state, the estimated probability of winning from that state. This is the **value function**.
2. Initialize terminal states correctly: 1.0 for already-won states, 0 for already-lost or drawn states, 0.5 for everything else.
3. Play games. At each turn, look up the values of all states reachable by our possible moves and usually pick the one with the highest value (**greedy**). Occasionally pick randomly instead (**exploratory move**).
4. After each greedy move, **back up** the value: nudge the value of the state we just left toward the value of the state we just entered:

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

where

$$\begin{aligned} \alpha > 0 &\equiv \text{step-size parameter (learning rate)} \\ V(S_t) &\equiv \text{estimated probability of winning from state } S_t \\ V(S_{t+1}) - V(S_t) &\equiv \text{temporal-difference error} \end{aligned}$$

This is called a **temporal-difference (TD) update**, the update is driven by the difference between value estimates at two successive timesteps.

With a properly decaying step size, this converges to the true winning probabilities against any fixed opponent, and the resulting policy is optimal against that opponent. With a non-zero step size, it can even adapt to opponents that gradually change their play.

1.5.4. What This Example Shows

- **Value function methods beat evolutionary methods** at exploiting what happens during a game. Individual states and moves can be evaluated and credited, not just the final outcome.
- The RL player naturally develops **multi-step planning** behavior (setting traps for shortsighted opponents) without an explicit search or a model of the opponent. Foresight emerges from learning.
- **Exploration vs. exploitation** is built in: mostly greedy, occasionally random.

1.5.5. What Tic-Tac-Toe Doesn't Capture (But RL Handles Anyway)

Tic-tac-toe is deliberately simple. RL is much more general:

- **Continuous problems**: no discrete episodes, rewards can come at any time.
- **Huge or infinite state spaces**: Tesauro's TD-Gammon combined this exact algorithm with a neural network to master backgammon ($\sim 10^{20}$ states), eventually surpassing the world's best human players (Section 16.1). Generalization from experience through function approximation is what makes this possible.
- **Partial observability**: RL works even when the agent can't see the full state.
- **No model required**: the tic-tac-toe player has no model of its opponent whatsoever and still learns to beat it. Model-free methods are often simpler, more robust when accurate models are hard to build, and important building blocks for model-based methods.
- **Prior knowledge**: RL doesn't require starting from scratch, prior information can be incorporated in many ways (Sections 9.5, 13.1, 17.4).
- **Hierarchical learning**: RL can operate at multiple levels simultaneously, where each "action" at a higher level is itself a complex problem-solving procedure.

1.6. Summary

Reinforcement learning is a computational approach to goal-directed learning and decision making, distinguished from other approaches by its emphasis on learning through **direct interaction with an environment**, without needing labeled examples or a complete environment model.

The formal backbone is the **Markov Decision Process (MDP)** framework, which defines the agent-environment interaction in terms of states, actions, and rewards. MDPs capture the essentials of the AI problem cleanly: cause and effect, uncertainty, and explicit goals.

The central concept is the **value function**: a way of judging states not by their immediate reward but by the long-term reward they tend to lead to. Value functions enable efficient search through the space of policies, which is what separates most RL methods from evolutionary approaches that evaluate entire policies wholesale and ignore what happens along the way.

In short:

- **Agent** learns from interaction, not supervision.
- **MDP** formalizes the problem.
- **Value functions** make the search tractable.

1.7. Early History of Reinforcement Learning

Modern reinforcement learning grew from three independent threads that developed over decades before converging in the late 1980s.

1.7.1. Thread 1: Trial-and-Error Learning

The idea that behavior is shaped by its consequences goes back to the 1850s, but **Edward Thorndike** gave it its clearest early statement in his **Law of Effect** (1911): actions followed by satisfaction become more likely; actions followed by discomfort become less likely. The greater the satisfaction or discomfort, the stronger the effect. This

became the foundation for Clark Hull's learning theories and B.F. Skinner's experimental methods. The term "reinforcement" itself entered the picture via Pavlov's 1927 work on conditioned reflexes.

Alan Turing (1948) was among the first to propose implementing this principle in a machine: a "pleasure-pain system" that made tentative decisions and made them permanent on reward or cancelled them on punishment. A wave of ingenious electro-mechanical machines followed:

- Thomas Ross (1933): a maze-solving machine with switch memory
- W. Grey Walter (1951): a learning "mechanical tortoise"
- Claude Shannon (1952): Theseus, a maze-running mouse
- Marvin Minsky (1954): analog SNARCs (Stochastic Neural-Analog Reinforcement Calculators), modeled on synaptic connections

In the 1960s, **Donald Michie** built MENACE, a tic-tac-toe learner made of matchboxes and colored beads, and later BOXES, a reinforcement learning controller that balanced a pole on a cart using only a failure signal. Pole-balancing became a landmark RL benchmark. **Minsky's "Steps Toward Artificial Intelligence"** (1961) articulated the **credit assignment problem**: how do you distribute credit for success across the many decisions that led to it? This remains a central question in RL.

Research into genuine trial-and-error learning quieted in the 1960s-70s, partly due to confusion with supervised learning. Many researchers using "reward and punishment" language were actually studying supervised learning systems (Rosenblatt, Widrow & Hoff). **Harry Klopf** (1972, 1975, 1982) revived the thread by arguing that hedonic, goal-seeking behavior (the drive to control the environment toward desired ends) was being lost as the field drifted toward supervised learning. His influence directly shaped the early work of Barto and Sutton.

1.7.2. Thread 2: Optimal Control and Dynamic Programming

In the mid-1950s, **Richard Bellman** formalized the problem of designing a controller to optimize a dynamical system's behavior over time. His key contributions:

- The **Bellman equation**: a functional equation defining optimal value
- **Dynamic programming (DP)**: the class of methods for solving it
- **Markov Decision Processes (MDPs)**: the discrete stochastic version of optimal control

Ronald Howard (1960) added policy iteration. Together, these became the theoretical backbone of modern RL.

DP is still considered the only general method for stochastic optimal control, but it suffers from Bellman's own **"curse of dimensionality"** where computational cost grows exponentially with the number of state variables. For a long time, DP was seen as an offline, model-dependent computation with little connection to learning. That perception changed slowly, with Werbos (1987) being an early advocate for integrating DP with learning, and **Chris Watkins (1989)** finally completing the integration through Q-learning.

1.7.3. Thread 3: Temporal-Difference Learning

Temporal-difference (TD) methods are driven by the difference between successive estimates of the same quantity, like the updating step in the tic-tac-toe example. This thread has roots in the psychology of **secondary reinforcement**: a neutral stimulus paired with a primary reward comes to itself predict and signal reward.

- **Arthur Samuel** (1959) was the first to implement a TD-like idea computationally, in his checkers-playing program, though he didn't connect it to animal learning or to Bellman's work.
- **Minsky** (1961) later drew the connection to secondary reinforcement.
- **Sutton** (1978) formalized TD learning and linked it to animal conditioning models.
- **Sutton and Barto** (1981) produced an influential psychological model of classical conditioning based on TD.

1.7.4. Convergence

By 1981, Barto, Sutton, and Anderson had combined TD learning with trial-and-error learning in the **actor-critic architecture**, applied to Michie and Chambers's pole-balancing task. Sutton (1988) generalized TD learning as a standalone prediction method and introduced the **TD(λ)** algorithm. Watkins's **Q-learning** (1989) brought all three threads fully together.

The field then accelerated rapidly:

- **TD-Gammon** (Tesauro, 1992), backgammon at superhuman level using TD + neural networks, brought wide attention to RL.
- Neuroscience found a striking parallel between TD algorithms and the behavior of **dopamine-producing neurons** in the brain (Montague, Dayan & Sejnowski, 1996; Schultz, Dayan & Montague, 1997), launching a productive dialogue between RL and neuroscience covered in Chapter 15.

2. Multi-Armed Bandits

2.1. The k-armed Bandit Problem

- The particular nonassociative, evaluative feedback problem is a simple version of the k-armed bandit problem.

$$q_*(a) \doteq \mathbb{E}\left[R_t \mid A_t = a\right] \quad \text{action-value function}$$

$$Q_t(a) \to q_*(a)$$

where
$Q_t(a) =$ estimated action-value of action a at time t

- **Exploitation**: always selecting the greedy actions (actions with highest estimated value)
- **Exploration**: selecting one of the nongreedy actions
- There exists an **Exploration-Exploitation tradeoff**

> **Multi-Armed Bandit**
>
> A multi-armed bandit is a tuple $\langle A, R\rangle$ where A is a known set of m actions or **"arms"** and $R^a(r)$ is an unknown probability distribution over rewards.

2.2. Action-value Methods

- Let's go through how we can estimate our action-value, $Q_t(a)$. One method is by averaging the rewards that have been received (**Sample average**)

$$Q_t(a) \doteq \frac{\text{sum of rewards when } a \text{ taken prior to } t}{\text{number of times } a \text{ taken prior to } t} = \frac{\sum_{i=1}^{t-1} R_i \cdot \mathbb{1}_{A_i=a}}{\sum_{i=1}^{t-1} \mathbb{1}_{A_i=a}}$$

where
$a =$ action
$t =$ timestep
$\mathbb{1}_{\text{predicate}} \equiv$ random variable that is 1 if predicate is true, 0 if not

- As $\sum_{i=1}^{t-1} \mathbb{1}_{A_i=a} \to \infty$, by the law of large numbers, $Q_t(a)$ converges to $q_*(a)$.
- Now that we can compute $Q_t(a)$, how do we use it to select actions?

2.2.1. Greedy Method

- Simplest action selection rule for no exploration.

$$A_t \doteq \arg\max_a Q_t(a)$$

2.2.2. ε-greedy method

- Simplest action selection rule for ensuring continual exploration.
- For m total actions:
  - With probability $1-\varepsilon$: choose the greedy action
  - With probability ε: choose a random action

$$\pi(a|s) = \left\{ \begin{array}{ll} \varepsilon/m + 1 - \varepsilon & \text{if } a^* = \arg\max_{a \in A} Q(s,a) \\ \varepsilon/m & \text{otherwise} \end{array} \right\}$$

2.3. The 10-armed Testbed

- Stationarity & determinism vs. nonstationarity & stochasticity
- Compare the relative effectiveness of greedy & ε-greedy action-value methods

2.4. Incremental Implementation

- Maintaining sample averages naively requires storing all past rewards, causing memory and computation to grow unboundedly over time.
- For a single action selected $n-1$ times, its value estimate Q_n is:

$$Q_n \doteq \frac{R_1 + R_2 + ... + R_{n-1}}{n-1}$$

- How can we compute $Q_t(a)$ estimates in a computationally and memory efficient manner?
- Instead of storing all the R values and recomputing the sum at every time step, we can derive an **incremental update rule:** Q_n estimate can be updated efficiently without storing the full history.

$$\begin{aligned} Q_{n+1} &= \frac{1}{n}\sum_{i=1}^{n} R_i \\ &= \frac{1}{n}\left(R_n + \sum_{i=1}^{n-1} R_i\right) \\ &= \frac{1}{n}\left(R_n + (n-1)\frac{1}{n-1}\sum_{i=1}^{n-1} R_i\right) \\ &= \frac{1}{n}\left(R_n + (n-1)Q_n\right) \\ &= \frac{1}{n}\left(R_n + nQ_n - Q_n\right) \end{aligned}$$

$$\boxed{Q_{n+1} = Q_n + \frac{1}{n}\left[R_n - Q_n\right]}$$

- Throughout the book, α is used to represent $\left(\frac{1}{n}\right)$ as the step-size parameter.
- In general:

$$\text{NewEstimate} \leftarrow \text{OldEstimate} + \text{StepSize}\underbrace{\left[\text{Target} - \text{OldEstimate}\right]}_{\text{error}}$$

2.5. Tracking a Nonstationary Problem

- Sample-average methods assume a stationary reward distribution, so they perform poorly on non-stationary bandits where the true action values drift/change over time.

- A simple fix is to replace the $\frac{1}{n}$ step size with a constant $\alpha \in (0, 1]$, which gives more weight to recent rewards via an exponentially decaying average over past rewards.

$$Q_{n+1} \doteq Q_n + \alpha \left[R_n - Q_n \right], \quad \alpha \in (0, 1] \text{ is constant}$$

- Expanding the recurrence:

$$\begin{aligned}
Q_{n+1} &= Q_n + \alpha[R_n - Q_n] \\
&= \alpha R_n + (1-\alpha) Q_n \\
&= \alpha R_n + (1-\alpha) \left[\alpha R_{n-1} + (1-\alpha) Q_{n-1} \right] \\
&= \alpha R_n + (1-\alpha) \alpha R_{n-1} + (1-\alpha)^2 Q_{n-1} \\
&= \alpha R_n + (1-\alpha) \alpha R_{n-1} + (1-\alpha)^2 \left[\alpha R_{n-2} + (1-\alpha) Q_{n-2} \right] \\
&= \alpha R_n + (1-\alpha) \alpha R_{n-1} + (1-\alpha)^2 \alpha R_{n-2} + (1-\alpha)^3 Q_{n-2} \\
&= \alpha R_n + (1-\alpha) \alpha R_{n-1} + (1-\alpha)^2 \alpha R_{n-2} + (1-\alpha)^3 \alpha R_{n-3} + ... + (1-\alpha)^{n-1} \alpha R_1 + (1-\alpha)^n Q_1 \qquad \vdots
\end{aligned}$$

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

2.5.1. Exponential recency-weighted average

- The weight given to R_i depends on how many rewards ago $(n-i)$ it was observed. The more recent the rweard, the higher the weight assigned to it.
- This is a **weighted average** , which gives priority/higher value to more recently acquired rewards.
- It's called a weighted average because the sum of the weights is 1.

$$(1-\alpha)^n + \sum_{i=1}^{n} \alpha (1-\alpha)^{n-i} = 1$$

- The weighted average is more specifically called an **exponential recency-weighted average** because the weight decays exponentially into the past according to the exponent on $(1-\alpha)$, $(n-i)$.
- Q_{n+1} becomes a weighted average of the previous rewards and the initial estimate Q_1.

$$\text{If } (1-\alpha) = 0 \Rightarrow \alpha = 1, \text{then:}$$

$$\begin{aligned}
Q_{n+1} &= (1-\alpha)^n Q_1 + \sum_{i=1}^{n} \alpha (1-\alpha)^{n-i} R_i \\
&= 0^n Q_1 + \sum_{i=1}^{n} 1 \cdot (0)^{n-i} R_i \\
&= Q_1 + \sum_{i=1}^{n} R_i \\
&= Q_1 + R_1 + R_2 + R_3 + ... + R_n
\end{aligned}$$

2.5.2. Step-size Parameter

$$\alpha_n(a) = \frac{1}{n} \quad \text{(sample-average)} \Rightarrow \textbf{unbiased}$$
$$\alpha_n(a) = \alpha \quad \text{(constant)} \Rightarrow \textbf{biased}$$

- Stochastic approximation theory (Robbins-Monro sequence) requires:

$$\text{(I)} \quad \sum_{n=1}^{\infty} \alpha_n(a) = \infty \quad \text{[overcome random fluctuations]}$$

$$\text{(II)} \quad \sum_{n=1}^{\infty} \alpha_n^2(a) < \infty \quad \text{[convergence]}$$

- `(I)` & `(II)` are met for $\alpha_n(a) = \frac{1}{n}$ (sample-average method)
- `(II)` is **not** met for $\alpha_n(a) = \alpha$ (constant method)

2.6. Optimistic Initial Values

- The methods discussed so far are dependent to some extent on the initial action-value estimates i.e. they are biased by their initial estimates.
- Essentially, the initial action-value estimates become model parameters that must be chosen by the user.
- For example, in a 10-armed bandit testbed if we set $Q_1(a) = +5$ instead of 0, we encourage exploration even in the greeedy case.
  - The agent will almost always be disappointed with its current samples action-values because they are less than the initial action-value estimate of +5, therefore it will continue to explore until the action-values converge.
  - This technique for encouraging exploration is called **optimistic initial values** (focuses on initial conditions)

> **i** Optimistic Initialization
>
> ***Assume the best action-value initially, until proven otherwise.*** Basically **optimistic initialization** encourages systematic exploration early on but can still lock onto suboptimal action.

2.6.1. Pros & Cons

- (+) It is **effective on stationary problems,** but not a general approach
- (-) It is **not good for nonstationary problems** because its drive for exploration is inherently temporary.

2.7. Upper-Confidence-Bound (UCB) Action Selection

- ε-greedy action selection explores indiscriminately, giving equal probability to all non-greedy actions regardless of their potential.
- A smarter approach is **Upper Confidence Bound (UCB)** action selection, which favors non-greedy actions based on both how close their value estimates are to the maximum and how uncertain those estimates are.
- Actions can be selected by:

$$A_t \doteq \arg\max_a \left[Q_t(a) + c\sqrt{\frac{\ln(t)}{N_t(a)}} \right]$$

where

$t \equiv$ number of timesteps that have passed

$N_t(a) \equiv$ number of times action a has been selected prior to time t

$c > 0$ controls the degree of exploration ($c \equiv$ confidence level)

$\sqrt{\frac{\ln(t)}{N_t(a)}} \equiv$ measure of the uncertainty/variance in the estimate of a's value

- If $N_t(a) = 0$, then a is considered to be a maximizing action.
- Introduces an exploration bonus term $c\sqrt{\ln(t)/N_t(a)}$, which measures the uncertainty in our estimate
  - It is proportional to t and inversely proportional to $N_t(a)$.

– The more time has passed, and the less we have sampled an action, the higher our upper-confidence-bound.

$$t \uparrow,\ N_t(a) \downarrow \Rightarrow \sqrt{\frac{\ln t}{N_t(a)}} \uparrow \Rightarrow \text{ UCB} \uparrow$$

- The idea of the UCB action selection is that the square-root term is a measure of the uncertainty or variance in the estimate of a's value.
- The quantity being max'ed over is a sort of upper bound on the possible true value of action a, with c determining the confidence level.
- Each time a is selected, $N_t(a)$ increments and, since it appears in the denominator, the uncertainty term decreases.
- Conversely, each time any other action is selected the uncertainty increases (t increases while $N_t(a)$ does not).
- Using $\ln t$ means the uncertainty bonus grows without bound but at a diminishing rate (increases get smaller over time), ensuring all actions are eventually selected while those with lower value estimates or higher selection counts are chosen with decreasing frequency over time.

2.7.1. Pros & Cons

- (+) UCB usually performs better than ε-greedy
- (-) UCB is more difficult than ε-greedy to **extend beyond bandits** to the more general RL settings
- (-) Difficulty in dealing with **nonstationary problems**
- (-) Difficulty in dealing with **large state spaces** (especially function approximation)

> **i** Optimism in the Face of Uncertainty
>
> ***The more uncertain we are about an action-value, the more important it is to explore that action, because it could turn out to be the best action.***

2.8. Gradient Bandit Algorithms

- Another action selection method, which isn't based on action-value estimation, entails learning a numerical **preference** for each action a, which is denoted as $H_t(a)$.
- The larger the preference ($H_t(a)$), the more often the action is taken.
- The relative preference of one action over another is more important, hence why we define the action probabilities by a **soft-max distribution** (i.e., Gibbs or Boltzmann):

$$\mathbb{P}\{A_t = a\} \doteq \frac{e^{H_t(a)}}{\sum_{b=1}^{k} e^{H_t(b)}} \doteq \pi_t(a) \quad \text{(softmax)}$$

where

$H_t(a) \in \mathbb{R} \equiv$ numerical preference for action a at time t

$\pi_t(a) =$ probability of taking action a at time t

- Initially all action preferences are the same (e.g., $H_1(a) = 0 \forall a$) so that all actions have an equal probability of being selected.
- The action preferences are updated via **stochastic gradient ascent**:

$$H_{t+1}(A_t) \doteq H_t(A_t) + \alpha(R_t - \bar{R}_t)(1 - \pi_t(A_t)), \quad \text{and}$$
$$H_{t+1}(a) \doteq H_t(a) - \alpha(R_t - \bar{R}_t)\pi_t(a), \quad \forall a \neq A_t$$

where

$\alpha > 0 \equiv$ step-size parameter

$\bar{R}_t \in \mathbb{R} \equiv$ average of rewards up to but not including time t ($\bar{R}_1 \doteq R_1$)

$\bar{R}_t \equiv$ baseline against which the reward is compared

- If $R_t > \bar{R}_t \Rightarrow \mathbb{P}(A_t = a) \uparrow$
- If $R_t < \bar{R}_t \Rightarrow \mathbb{P}(A_t = a) \downarrow$
- If $R_t > \bar{R}_t \Rightarrow \mathbb{P}(A_t = a) \downarrow \quad \forall a \neq A_t$
- If $R_t < \bar{R}_t \Rightarrow \mathbb{P}(A_t = a) \uparrow \quad \forall a \neq A_t$

2.8.1. Proof: The Bandit Gradient Algorithm as Stochastic Gradient Ascent

Gradient ascent update

$$H_{t+1}(a) \doteq H_t(a) + \alpha \frac{\partial \mathbb{E}[R_t]}{\partial H_t(a)}$$

$$\text{where} \quad \mathbb{E}[R_t] = \sum_x \pi_t(x) q_*(x)$$

Exact performance gradient

$$\begin{aligned}
\frac{\partial \mathbb{E}[R_t]}{\partial H_t(a)} &= \frac{\partial}{\partial H_t(a)} \left[\sum_x \pi_t(x) q_*(x) \right] \\
&= \sum_x q_*(x) \frac{\partial \pi_t(x)}{\partial H_t(a)} \\
&= \sum_x \left(q_*(x) - B_t \right) \frac{\partial \pi_t(x)}{\partial H_t(a)} \\
&= \sum_x \pi_t(x) \left[q_*(x) - B_t \right] \frac{\partial \pi_t(x)}{\partial H_t(a)} \frac{1}{\pi_t(x)} \\
&= \mathbb{E} \left[\left(q_*(A_t) - B_t \right) \frac{\partial \pi_t(A_t)}{\partial H_t(a)} \frac{1}{\pi_t(A_t)} \right] \quad (B_t = \bar{R}_t) \\
&= \mathbb{E} \left[\left(R_t - \bar{R}_t \right) \frac{\partial \pi_t(A_t)}{\partial H_t(a)} \frac{1}{\pi_t(A_t)} \right]
\end{aligned}$$

$$q_*(A_t) = R_t \text{ because } \mathbb{E}[R_t | A_t] = q_*(A_t)$$

Aside:

$$\frac{\partial \pi_t(x)}{\partial H_t(a)} = \pi_t(x) \left[\mathbb{1}_{a=x} - \pi_t(a) \right]$$

$$\text{where } \mathbb{1}_{a=x} = \left\{ \begin{array}{ll} 1 & \text{if } a = x \\ 0 & \text{otherwise} \end{array} \right\}$$

Therefore:

$$\begin{aligned}
\frac{\partial \mathbb{E}[R_t]}{\partial H_t(a)} &= \mathbb{E} \left[\left(R_t - \bar{R}_t \right) \pi_t(A_t) \left(\mathbb{1}_{a=A_t} - \pi_t(a) \right) \frac{1}{\pi_t(A_t)} \right] \\
&= \mathbb{E} \left[\left(R_t - \bar{R}_t \right) \left(\mathbb{1}_{a=A_t} - \pi_t(a) \right) \right]
\end{aligned}$$

$$\Rightarrow \boxed{H_{t+1}(a) = H_t(a) + \alpha \left(R_t - \bar{R}_t \right) \left(\mathbb{1}_{a=A_t} - \pi_t(a) \right)} \quad \text{for all } a$$

Prove $\dfrac{\partial \pi_t(x)}{\partial H_t(a)} = \pi_t(x)\left[\mathbb{1}_{a=x} - \pi_t(a)\right]$:

Using the quotient rule: $\dfrac{\partial}{\partial z}\left[\dfrac{f(z)}{g(z)}\right] = \dfrac{f'(z)g(z) - f(z)g'(z)}{g(z)^2}$:

$$
\begin{aligned}
\frac{\partial \pi_t(x)}{\partial H_t(a)} &= \frac{\partial}{\partial H_t(a)}\left[\frac{e^{H_t(x)}}{\sum_{y=1}^{k} e^{H_t(y)}}\right] \\
&= \frac{\frac{\partial e^{H_t(x)}}{\partial H_t(a)}\sum_{y=1}^{k} e^{H_t(y)} - e^{H_t(x)}\frac{\partial \sum_{y=1}^{k} e^{H_t(y)}}{\partial H_t(a)}}{\left(\sum_{y=1}^{k} e^{H_t(y)}\right)^2} \\
&= \frac{\mathbb{1}_{a=x}\, e^{H_t(x)}\sum_{y=1}^{k} e^{H_t(y)} - e^{H_t(x)}e^{H_t(a)}}{\left(\sum_{y=1}^{k} e^{H_t(y)}\right)^2} \\
&= \frac{\mathbb{1}_{a=x}\, e^{H_t(x)}}{\sum_{y=1}^{k} e^{H_t(y)}} - \frac{e^{H_t(x)}e^{H_t(a)}}{\left(\sum_{y=1}^{k} e^{H_t(y)}\right)^2} \\
&= \mathbb{1}_{a=x}\,\pi_t(x) - \pi_t(x)\pi_t(a) \\
&= \pi_t(x)\left(\mathbb{1}_{a=x} - \pi_t(a)\right)
\end{aligned}
$$

$$
\boxed{\frac{\partial \pi_t(x)}{\partial H_t(a)} = \pi_t(x)\left[\mathbb{1}_{a=x} - \pi_t(a)\right]}
$$

2.9. Associative Search (Contextual Bandits)

- **Nonassociative tasks:** tasks where there is no need to associate different actions with different situations.
- **Associative Search:** extend nonassociative tasks to the associative setting.
  - Consider a setting where the bandit task changes from step to step, but the value distributions of the arms within each task remain fixed.
  - These are called **associative search tasks**, or **contextual bandits**, where the agent must learn to associate different policies with different task contexts.
- These tasks are intermediate between the k-armed bandit stationary problem & the full reinforcement learning (RL) problem
- Like full RL probelm: they learn a policy
- Like k-armed bandit: each action affects only the immediate reward
- These tasks involve both trial & error learning to search for the best actions & **association** of these actions with the situations in which they are best
- **Full RL problem**: Actions are allowed to affect the **next situation** as well as the reward.

> **i** Contextual Bandit
>
> A contextual bandit is a tuple $\langle A, S, R\rangle$ where A is a known set of m actions or **"arms"**, S is an unknown probability distribution over states or **"contexts"** and $R^a(r)$ is an unknown probability distribution over rewards.

2.10. Summary

- Action values $Q_t(a)$ can be estimated online via **incremental updates**, keeping memory and per-step computation constant rather than storing every historical reward.
- Stationary tasks suit decaying $1/n$ step sizes, whereas **nonstationary tasks** call for constant step sizes α that exponentially down-weight older returns.
- Core exploration mechanisms trade off simplicity against sample efficiency:

  - ε-**greedy** injects uniform random exploration with small probability ε.
  - **Optimistic initialization** triggers early exploration even under purely greedy action selection, though its effect fades over time.
  - **UCB** adds an uncertainty bonus that deterministically prioritizes rarely sampled actions.
  - **Gradient bandits** bypass value estimation altogether, maintaining action preferences converted to probabilities through a softmax policy.
- Comparing algorithms cleanly across hyperparameter choices requires **parameter studies** (plotting average reward over a wide log-spaced parameter sweep), which typically reveal an inverted-U sweet spot for every method.
- Bayesian methods such as **Gittins indices** and **Thompson (posterior) sampling** formulate optimal exploration over belief states, connecting single-state bandit problems directly to full sequential RL (see Sutton & Barto, 2018, §2.10).

3. Finite Markov Decision Processes

- MDPs are a classical formalization of sequential decision making, where actions influence not just immediate rewards, but also subsequent states, and through those future rewards.
- Hence, there exists a tradeoff between delayed and immediate rewards.
- In bandit problems, we estimated $q_*(a)$ of each action a.
- In MDPs, we estimate the value $q_*(s, a)$ of each action a in each state s.
- These state-dependent action-values are essential for accurate credit assignment for long-term consequences to individual action selections.
- Markov Decision Processes (MDPs) are a mathematical idealized form of the reinforcement learning problem.
- To build the full reinforcement learning (RL) problem's mathematical structure, key elements are introduced: **returns, value functions, & Bellman equations.**

3.1. The Agent-Environment Interface

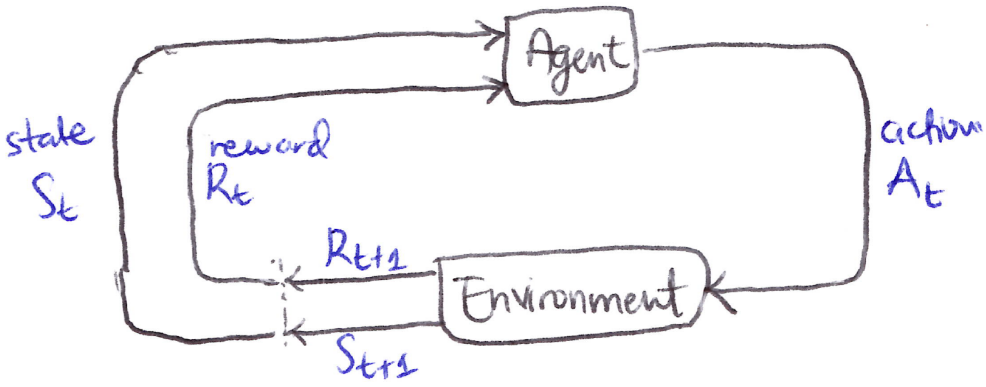

Figure 3.1. The agent-environment interaction in an MDP.

- The agent is the learner and decison maker, while the environment is everything outside the agent that the agent interacts with.
- The agent selects actions, and the environment responds with new states back to the agent.
- The environment also yields rewards, numerical values which the agent seeks to maximize over time through its choice of actions.
- The agent and environment interact continually over a sequence of discrete time steps, $t = 0, 1, 2, 3, ...$
- The MDP and agent together yield a recurring sequence of $S_t, A_t, R_{t+1}, S_{t+1}$ or a **trajectory** when expanded out for all the time steps

$$\textbf{Trajectory, } \mathcal{T} \equiv (S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, ...)$$

3.1.1. Dynamics function

- In a **finite** MDP, the sets of states S, actions A, and rewards R are all finite. The random variables S_t and R_t have well-defined discrete probability distributions dependent only on the preceding state and action, this is the **Markov property**.
- The transition dynamics are fully characterized by the function p, which gives the probability of transitioning to state $s' \in S$ and receiving reward $r \in R$, given state s and action a

- $\forall\ s', s \in \mathcal{S},\ r \in \mathbb{R},\ a \in \mathcal{A}(s)$, the dynamics function is:

$$\boxed{p(s', r \mid s, a) \doteq \mathbb{P}\{S_t = s',\ R_t = r \mid S_{t-1} = s,\ A_{t-1} = a\}}$$

- The dynamics function is an ordinary deterministic function of 4 arguments:

$$p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$$

- The "|" in the middle of $p(s', r|s, a)$ signifies conditional probability, which reminds us that p specifies a probability distribution for each choice of s and a. i.e., that

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r \mid s, a) = 1 \quad \forall\, s \in \mathcal{S},\ a \in \mathcal{A}(s)$$

3.1.2. Derived quantities

- From the 4-argument dynamics function, p, we can compute anthing we want to know about the environment. Let's go through some of these functions of interest below.
- *State-transition probabilities* as a 3-argument function $p : S \times S \times A \to [0, 1]$

$$p(s'|s, a) \doteq \mathbb{P}\{S_t = s' \mid S_{t-1} = s,\ A_{t-1} = a\} = \sum_{r \in R} p(s', r|s, a)$$

- *Expected rewards for state-action pairs* as a 2-argument function $r : S \times A \to \mathbb{R}$

$$r(s, a) \doteq \mathbb{E}\left[R_t \mid S_{t-1} = s,\ A_{t-1} = a\right] = \sum_{r \in R} r \sum_{s' \in S} p(s', r|s, a)$$

- *Expected rewards for state-action-next-state triples* as a 3-argument function $r : S \times A \times S \to \mathbb{R}$

$$r(s, a, s') \doteq \mathbb{E}\left[R_t \mid S_{t-1} = s,\ A_{t-1} = a,\ S_t = s'\right] = \sum_{r \in R} r \frac{p(s', r|s, a)}{p(s'|s, a)}$$

3.2. Goals & Rewards

- At each time step, the reward is a simple number, $R_t \in \mathbb{R}$
- The agent's goal is to maximize the cumulative reward it receives in the long run not the immediate reward.
- This idea was formalized by the **reward hypothesis**.

> **i** Reward Hypothesis (Sutton & Barto, 2018, §3.2)
>
> "That all of what 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)."

- Because the agent optimizes whatever scalar signal it receives, reward design must align maximizing cumulative reward with achieving the actual objective.
- Rewards specify *what* outcome we want, not *which* intermediate steps or heuristics the agent should use to get there.

3.3. Returns & Episodes

- The formal definition of the maximization of the cumulative reward received by the agent in the long run leads to the **expected return,** G_t we want to maximize:

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T, \qquad T = \text{final time step}$$

- Episodes are finite sequences of interactions between an agent and environment that have a natural terminal state, after which the process resets to a starting state. Each episode is independent, beginning from an initial state and ending at a terminal state S_T.
- An episode is basically a trajectory with a defined endpoint S_T.

$$\textbf{Episode} \equiv (S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, ..., S_T)$$

- **Episodic tasks**: Tasks with episodes that have a terminal state, S_T
- **Continuing tasks**: Tasks with no terminal state ($T_{\text{final}} = \infty$)

3.3.1. Discounting

- $\gamma \equiv$ discount rate/factor
- $\gamma \Rightarrow$ determines the present value of future rewards
- $0 \leq \gamma \leq 1$
- $\gamma \approx 0$: "myopic" agent
- $\gamma \to 1$: "far-sighted" agent

$$\boxed{G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}}$$

3.3.2. Recursive form of Expected return

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + ... \\ &= R_{t+1} + \gamma \left(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + ...\right) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned}$$

- From $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$, the expected return is finite if $R \neq 0$ and constant, if $\gamma < 1$. For example, if $R =$ a constant $+1$ at each timestep, then the discounted return is:

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

3.4. Unified Notation for Episodic & Continuing Tasks

- We can unify the notations for expected return for episodic and continuing tasks into a single notation by considering episode termination to be the entering of a **special absorbing state** that transitions only to itself and that generates only rewards of 0. For example, lets consider the state transtion diagram:

3.4.1. State transition diagram

$$S_0 \xrightarrow{R_1=+1} S_1 \xrightarrow{R_2=+1} S_2 \xrightarrow{R_3=+1} \square \circlearrowleft \quad \left\{ \begin{array}{l} R_4 = 0 \\ R_5 = 0 \\ \vdots \end{array} \right\}$$

- The square signifies the special absorbing state that corresponds to the end of an episode.
- Starting from S_0, the reward sequence is $+1, +1, +1, 0, 0, 0, ...$, which when summed yields the same return whether over the first T rewards ($T = 3$) or over the full infinite sequence, even with discounting present.
- The unified notation, including the possibility that $T = \infty$ or $\gamma = 1$ (but not both), for the return is:

$$\boxed{\tilde{G}_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k}$$

3.5. Policies, Value Functions & Bellman Equations

- Value functions are defined w.r.t. particular ways of acting, called **policies**.
- A **Policy** is a mapping from states to probabilities of selecting each possible action.

$$\boxed{\pi(a|s) = \mathbb{P}\left[A_t = a \mid S_t = s\right]}$$

- "|" in the middle of $\pi(a|s) \Rightarrow$ reminds us that it defines a probability distribution over $a \in A(s)$ for each $s \in S$.
- RL methods specify how the agent's policy is changed as a result of its experience.

3.5.1. State-value function for policy π

$$v_\pi(s) \doteq \mathbb{E}_\pi\left[G_t \mid S_t = s\right] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s\right], \quad \forall s \in S$$

3.5.2. Action-value function for policy π

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

- $\mathbb{E}_\pi[\cdot] \Rightarrow$ expected value of a random variable given that the agent follows policy π, if t is any time step.

3.5.3. Bellman Equation for v_π

- Recursive self-consistency conditions that must be satisfied by value functions for any policy and state in reinforcement learning and dynamic programming (DP) are called **Bellman equations.**

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi\left[G_t \mid S_t = s\right] \\ &= \mathbb{E}_\pi\left[R_{t+1} + \gamma G_{t+1} \mid S_t = s\right] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma\, \mathbb{E}_\pi\left[G_{t+1} \mid S_{t+1} = s'\right]\right] \\ &= \sum_a \pi(a|s) \sum_{s', r} p(s', r|s, a)\left[r + \gamma v_\pi(s')\right], \quad \forall\, s \in S \end{aligned}$$

- This is the **Bellman Equation for** v_π. It expresses the relationship between the value of a state and the value of its successor states.
- The Bellman equation averages over all possibilities, weighting each by its probability of occurring.
- It states that the starting state value must equal the (discounted) value of the expected next state, plus the reward expected along the way.
- We visualize this relationship between a state and its possible successor states with a **backup diagram.**
- It is called a backup diagram because it looks at how value is transferred **back up** the tree from the possible successor states of a specific state at the bottom of the tree to that specific state at the top of the tree.
- A state-value is an expectation over the available action-values under a policy:

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_{\pi,p}[G_t \mid S_t{=}s] \\ &= \mathbb{E}_{\pi,p}[q_\pi(s, A_t) \mid S_t{=}s], \quad \text{(cf. [*])} \\ &= \sum_a \pi(a|s) q_\pi(s,a) \end{aligned}$$

[*] Given a state-action pair, the return G_t is an action-value.

- G_t is a function of the state s (given by the condition "$S_t{=}s$") and the action A_t (given by the policy π under the expectation $\mathbb{E}_{\pi,p}$).

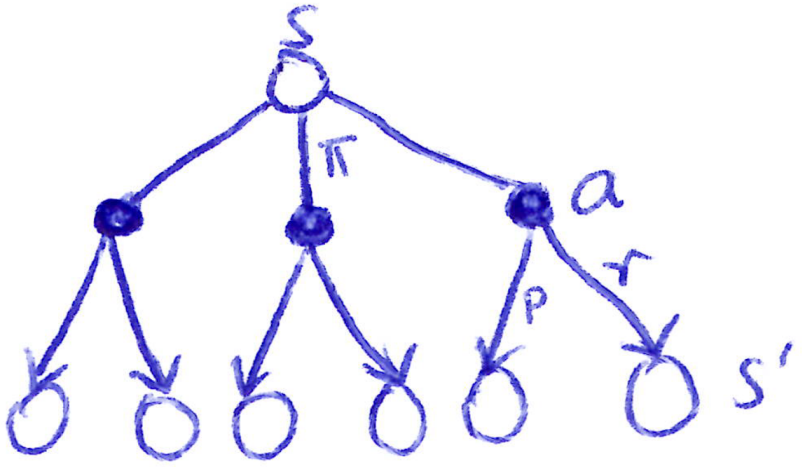

Figure 3.2. **Backup diagram for v_π.** Each open circle represents a state s, filled circles represent state-action pairs (s, a), next open circles set represent successor states s'.

- The following recursive relationship satisfied by $v_\pi(s)$ is called the *Bellman equation for state-values*. The value function $v_\pi(s)$ is the *unique solution* to its Bellman equation.

$$
\begin{aligned}
v_\pi(s) &\doteq \mathbb{E}_{\pi,p}[G_t \mid S_t{=}s] \\
&= \mathbb{E}_{\pi,p}[R_{t+1} + \gamma G_{t+1} \mid S_t{=}s] \\
&= \sum_a \pi(a|s)\mathbb{E}_{p,\pi}[R_{t+1} + \gamma G_{t+1} \mid S_t{=}s, A_t{=}a] \\
&= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\mathbb{E}_{\pi,p}[r + \gamma G_{t+1} \mid S_t{=}s, A_t{=}a, S_{t+1}{=}s'] \\
&= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\mathbb{E}_{\pi,p}[r + \gamma G_{t+1} \mid S_{t+1}{=}s'] \ \text{(by the Markov property)} \\
&= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\Big[r + \gamma\mathbb{E}_{\pi,p}[G_{t+1} \mid S_{t+1}{=}s']\Big] \\
&= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\Big[r + \gamma v_\pi(s')\Big]
\end{aligned}
$$

- Final form:

$$
\boxed{
\begin{aligned}
v_\pi(s) &= \sum_a \pi(a|s)\, q_\pi(s, a) \\
&= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\Big[r + \gamma v_\pi(s')\Big]
\end{aligned}
}
$$

3.5.4. Bellman equation for $q_\pi(s, a)$

- An action-value is an expectation over the possible next state-values under the environment's dynamics.

$$\begin{aligned}
q_\pi(s,a) &\doteq \mathbb{E}_{p,\pi}[G_t \mid S_t{=}s, A_t{=}a] \\
&= \mathbb{E}_{p,\pi}[R_{t+1} + \gamma G_{t+1} \mid S_t{=}s, A_t{=}a] \\
&= \mathbb{E}_{p,\pi}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t{=}s, A_t{=}a], \quad \text{(cf. [*])} \\
&= \sum_{s',r} p(s',r|s,a)\Big[r + \gamma v_\pi(s')\Big]
\end{aligned}$$

[*] Given only a state s, the return G_t is a state-value.

- G_t is a function of the state S_{t+1} (given by the environment's dynamics p under the expectation $\mathbb{E}_{p,\pi}$).
- The following recursive relationship satisfied by $q_\pi(s,a)$ is called the *Bellman equation for action-values*. The value function $q_\pi(s,a)$ is the *unique solution* to its Bellman equation.

$$\begin{aligned}
q_\pi(s,a) &\doteq \mathbb{E}_{p,\pi}[G_t \mid S_t{=}s, A_t{=}a] \\
&= \mathbb{E}_{p,\pi}[R_{t+1} + \gamma G_{t+1} \mid S_t{=}s, A_t{=}a] \\
&= \sum_{s',r} p(s',r|s,a)\mathbb{E}_{\pi,p}[r + \gamma G_{t+1} \mid S_t{=}s, A_t{=}a, S_{t+1}{=}s'] \\
&= \sum_{s',r} p(s',r|s,a)\mathbb{E}_{\pi,p}[r + \gamma G_{t+1} \mid S_{t+1}{=}s'], \quad \text{(by the Markov property)} \\
&= \sum_{s',r} p(s',r|s,a)\Big[r + \gamma \mathbb{E}_{\pi,p}[G_{t+1} \mid S_{t+1}{=}s']\Big] \\
&= \sum_{s',r} p(s',r|s,a)\Big[r + \gamma \sum_{a'} \pi(a'|s')\mathbb{E}_{p,\pi}[G_{t+1} \mid S_{t+1}{=}s', A_{t+1}{=}a']\Big] \\
&= \sum_{s',r} p(s',r|s,a)\Big[r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s',a')\Big]
\end{aligned}$$

- Final form:

$$\boxed{\begin{aligned}
q_\pi(s,a) &= \sum_{s',r} p(s',r|s,a)\Big[r + \gamma v_\pi(s')\Big] \\
&= \sum_{s',r} p(s',r|s,a)\Big[r + \gamma \sum_{a'} \pi(a'|s')\, q_\pi(s',a')\Big]
\end{aligned}}$$

3.6. Optimal Policies & Optimal Value Functions

- For finite MDPS, we can define an optimal policy for finite MDPs as a policy that achieves the maximum expected return.Value functions define a partial ordering over policies. A policy π is better than or equal to π' iff $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in S$.

$$\pi \geq \pi' \iff v_\pi(s) \geq v_{\pi'}(s) \quad \forall s \in S$$

- There always exists at least one **optimal policy** π_*, which is better than or equal to all other policies. All optimal policies share the same **optimal state-value function** v_*, defined as:

$$v_*(s) \doteq \max_\pi v_\pi(s) \quad \forall s \in S$$

- All optimal policies also share the same **optimal action-value function** q_*, defined as:

$$q_*(s,a) \doteq \max_\pi q_\pi(s,a) \quad \forall s \in S,\, a \in A(s)$$

- For a state-action pair (s,a), q_* gives the expected return for taking action a in state s and thereafter following an optimal policy. We can therefore write q_* in terms of v_* as:

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

3.6.1. Bellman Optimality Equations

- Self-consistency condition given by Bellman equation for state values must be satisfied by v_* and q_*.

Optimal state-value function, $v_*(s)$

$$\begin{aligned} v_*(s) &= \max_{a \in A(s)} q_{\pi_*}(s,a) \\ &= \max_a \mathbb{E}_{\pi^*}\left[G_t \mid S_t = s,\, A_t = a\right] \\ &= \max_a \mathbb{E}_{\pi^*}\left[R_{t+1} + \gamma G_{t+1} \mid S_t = s,\, A_t = a\right] \\ &= \max_a \mathbb{E}\left[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s,\, A_t = a\right] \\ &= \max_a \sum_{s',r} p(s', r|s,a)\left[r + \gamma v_*(s')\right] \end{aligned}$$

$$\boxed{v_*(s) \quad = \quad \max_a \sum_{s',r} p(s', r|s,a)\left[r + \gamma v_*(s')\right]}$$

Optimal action-value function, $q_*(s,a)$

$$\boxed{\begin{aligned} q_*(s,a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s,\, A_t = a\right] \\ &= \sum_{s',r} p(s', r|s,a)\left[r + \gamma \max_{a'} q_*(s', a')\right] \end{aligned}}$$

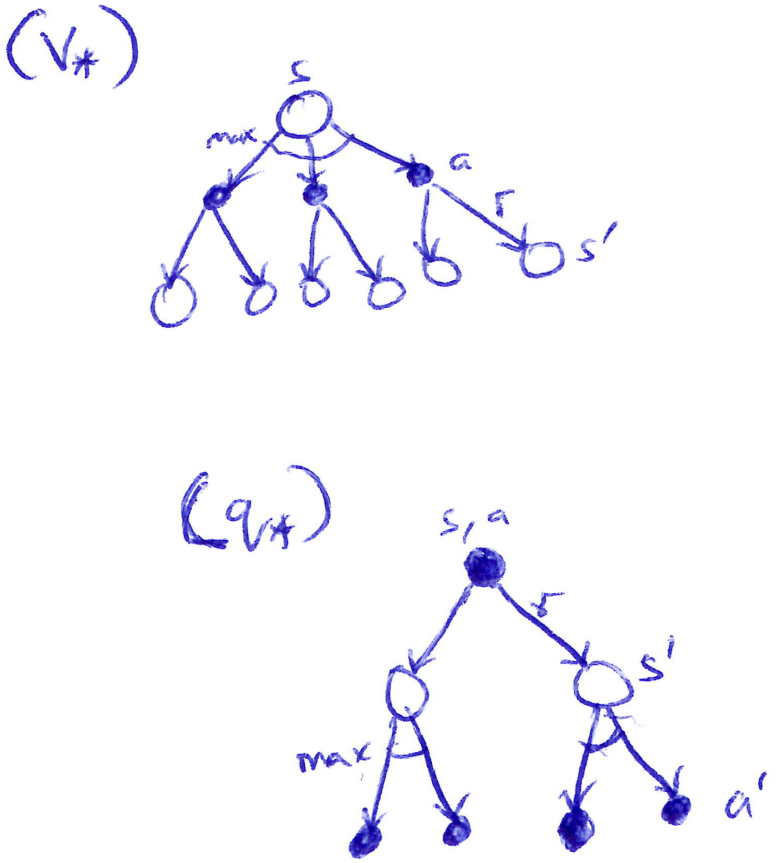

Figure 3.3. **Backup diagrams for v_* and q_*.** the max node appears at the state level for v_* and at the successor state level for q_*.

3.6.2. Takeaways

- For finite MDPs, the Bellman optimality equation has a **unique solution,** a system of n equations in n unknowns (one per state) solvable if the dynamics p are known.
- Any policy that is **greedy with respect to** v_* is an optimal policy: since v_* already accounts for all future rewards, a one-step-ahead search is sufficient to identify optimal actions.
- q_* makes optimal action selection even easier; the agent simply picks $\arg\max_a q_*(s, a)$ without needing to know the environment's dynamics or successor state values.
- Explicitly solving the Bellman optimality equation is rarely practical. It requires 3 assumptions that rarely all hold simultaneously:
  1. Known environment dynamics,
  2. Sufficient computational resources, and
  3. The Markov property
- In practice, RL methods settle for **approximate solutions** to the Bellman optimality equation, using actual experienced transitions in place of full knowledge of expected transitions.

3.7. Optimality & Approximation

- We have defined optimal value functions and optimal policies clearly.
  - A Q-table is not practical/possible in large state spaces.
  - Hence, the need for function approximation because calculation of optimality (Q-table) is too expensive.
- Computing optimal policies is rarely feasible in practice due to extreme computational cost, even with a complete and accurate model of the environment's dynamics.
- The agent is always constrained by available **computational power** (per time step) and **memory**, both of which limit how well value functions, policies, and models can be approximated.
- When the state space is small and finite, exact **tabular methods** suffice; for larger state spaces, **function approximation** with compact parameterized representations is necessary.
- RL's **online nature** allows approximations to focus more effort on frequently encountered states, at the expense of rarely visited ones. This makes useful approximate solutions achievable even when optimal ones are not.
- This ability to prioritize learning for high-frequency states is a key property that distinguishes RL from other approaches to approximately solving MDPs.

3.8. Summary

- RL is about learning from interaction to maximize cumulative reward over time, with the agent-environment interface defined by **actions, states, and rewards**.
- A **finite MDP** formalizes this with well-defined transition probabilities over finite state, action, and reward sets.
- The **return** is the (expected) function of future rewards the agent seeks to maximize (undiscounted for episodic tasks, discounted for tabular continuing tasks).
- **Value functions** assign expected returns to states or state-action pairs; **optimal value functions** give the maximum achievable return, and any policy greedy w.r.t. them is optimal.
- Problems vary by knowledge level: **complete knowledge** gives the agent full dynamics p; **incomplete knowledge** does not.
- In practice, optimal solutions are rarely computable, therefore memory and computation constraints force agents to **approximate** value functions, policies, and models.

3.9. Glossary

- **Reinforcement Learning**: learning from interaction how to behave in order to achieve a goal
- **Agent**: known & controllable; the RL agent & its environment interact over a sequence of discrete time steps
- **Environment**: incompletely controllable & may/may not be completely known
- **Actions**: choices made by the agent
- **States**: the basis/foundation for making the choices/actions
- **Rewards**: the basis for evaluating the choices/actions
- **Policy**: a stochastic rule by which the agent selects actions as a function of states (*agent's brain*)

$$\boxed{\pi(a|s) = \mathbb{P}[A_t = a | S_t = s]}$$

- **MDP**: the variables formulated above within a RL setup with well-defined transition probabilities
- A **finite MDP**: an MDP with finite state, action & reward sets.
- **MRP**: Markov Reward Process

Markov Property	**Markov Chain/Process**	**MRP**	**MDP**
$\mathbb{P}[S_{t+1} \mid S_t] = \mathbb{P}[S_{t+1} \mid S_1, ..., S_t]$	$\langle S, P \rangle$	$\langle S, P, R, \gamma \rangle$	$\langle S, A, P, R, \gamma \rangle$
State-transition probability matrix: $P_{ss'} = \mathbb{P}[S_{t+1} = s' \mid S_t = s]$	Sequences of random states $(S_1, S_2, ...)$ + Markov property	Markov Chain with Values	MRP + decisions

- **MDP components:**

$$\text{MDP} \left\{ \begin{array}{lcll} S & \rightarrow & \text{finite set of states} & \\ A & \rightarrow & \text{finite set of actions} & \\ P & \rightarrow & \text{state transition probability matrix,} & P^a_{ss'} = \mathbb{P}[S_{t+1} = s' | S_t = s,\, A_t = a] \\ R & \rightarrow & \text{reward function,} & R^a_s = \mathbb{E}[R_{t+1} | S_t = s,\, A_t = a] \\ \gamma & \rightarrow & \text{discount factor,} & \gamma \in [0, 1] \end{array} \right\}$$

- **Return**: function of future rewards that the agent seeks to maximize (in expected value)
- **Tasks**:
  - **Episodic**: have a clear starting & ending point (a terminal state) [e.g. ping pong]
  - **Continuing**: tasks/RL instances that continue forever w/o a terminal state [e.g. automated stock trading]
- **Discount factor,** γ: determines the present value of future rewards
- **Value functions** (v_π, q_π): a policy's value functions assign to each state/state-action pair, the expected return from that state/state-action pair, given that the agent uses the policy.
- **Optimal value functions** (v_*, q_*): assign to each state/state-action pair the largest expected return achievable by any policy.
- **Optimal policy** (π_*): a policy whose value functions are optimal $[v_*(s) = \max_\pi v_\pi(s)]$
  - $\pi_* > \pi,\ \forall \pi\ (\pi \geq \pi' \text{ iff } v_\pi(s) \geq v_{\pi'}(s)\ \forall s \in S)$
  - v_* & q_* are unique for a given MDP, but there can be many optimal policies (π_*)
  - Any policy that is **greedy** w.r.t. the optimal value functions (q_*, v_*) must be an optimal policy (π_*)
- **Bellman Optimality Equations**: special consistency conditions that q_*, v_* must satisfy & that can, in principle, be solved for q_*, v_*, from which an optimal policy (π_*) can be determined with relative ease.

4. Dynamic Programming

Dynamic Programming (DP): A collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as an MDP.

Assumptions:

1. The environment is a finite MDP
2. State S, action A, and reward R sets are finite
   - Its dynamics are given by a set of probabilities $p(s', r|s, a)$ for all $s \in S$, $a \in A(s)$, $r \in R$, $s' \in S^+$ (S^+ is S plus a terminal state if the problem is episodic)

$$p(s', r|s, a) = \mathbb{P}\left[R_{t+1} = r,\, S_{t+1} = s' \mid S_t, A_t\right]$$

Key idea:

- Use value functions to organize & structure the search for good policies
- Show how DP can be used to compute the value functions
- v_* or q_* that satisfy the **Bellman Optimality Equations** can then be used to easily obtain optimal policies π_*

4.0.1. Bellman Optimality Equations

$$\forall s \in S,\, a \in A(s),\, s' \in S^+$$

- **State-Value function:**

$$\begin{aligned} v_*(s) &= \max_a \mathbb{E}\left[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s,\, A_t = a\right] \\ &= \max_a \sum_{s', r} p(s', r|s, a)\left[r + \gamma v_*(s')\right] \end{aligned}$$

- **Action-Value function:**

$$\begin{aligned} q_*(s, a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s,\, A_t = a\right] \\ &= \sum_{s', r} p(s', r|s, a)\left[r + \gamma \max_{a'} q_*(s', a')\right] \end{aligned}$$

4.1. Policy Evaluation (Prediction)

Policy Evaluation: How to compute the state-value function v_π for an arbitrary policy π.

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}\left[G_t \mid S_t = s\right] \\ &= \mathbb{E}_\pi\left[R_{t+1} + \gamma G_{t+1} \mid S_t = s\right] \\ &= \mathbb{E}_\pi\left[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s\right] \\ &= \sum_a \pi(a|s) \sum_{s', r} p(s', r|s, a)\left[r + \gamma v_\pi(s')\right] \end{aligned}$$

- If the dynamics are known perfectly/completely, then the equation above 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 approximate value functions $(v_0, v_1, v_2, ...) : S^+ \Rightarrow \mathbb{R}$ with initial approximation v_0 chosen arbitrarily, then each successive approximation is obtained by using the Bellman Equation for v_π as an update rule, $\forall s \in S$:

$$\begin{aligned} v_{k+1}(s) &\doteq \mathbb{E}_\pi \left[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s \right] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_k(s') \right] \end{aligned}$$

4.1.1. Iterative Policy Evaluation

- Eventually $\{v_k\}$ sequence converges to v_π as $k \to \infty$ under the same conditions that guarantee the existence of v_π. This is **iterative policy evaluation**.
- At each iteration $k+1$, for all states $s \in S$, we update $v_{k+1}(s)$ from $v_k(s')$ where s' is a successor state of s.
- This update is called 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.

4.2. Policy Improvement

$$\begin{aligned} q_\pi(s, a) &\doteq \mathbb{E} \left[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s,\, A_t = a \right] \\ &= \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_\pi(s') \right] \end{aligned}$$

The process of making a new policy that improves on an original policy, by making it greedy w.r.t. the value function of the original policy.

- Given a deterministic policy, $a = \pi(s)$
- We can improve the policy by acting greedily: $\pi'(s) = \arg\max_{a \in A} q_\pi(s, a)$
- This improves the value from any state over one step (**monotonic policy improvement**):

$$q_\pi(s, \pi'(s)) = \max_{a \in A} q_\pi(s, a) \geq q_\pi(s, \pi(s)) = v_\pi(s)$$

- It therefore improves the value function, $v_{\pi'}(s) \geq v_\pi(s)$:

$$\begin{aligned} v_\pi(s) &\leq q_\pi(s, \pi'(s)) \\ &= \mathbb{E} \left[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s,\, A_t = \pi'(s) \right] \\ &= \mathbb{E}_{\pi'} \left[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s \right] \\ &\leq \mathbb{E}_{\pi'} \left[R_{t+1} + \gamma q_\pi(S_{t+1}, \pi'(S_{t+1})) \mid S_t = s \right] \\ &= \mathbb{E}_{\pi'} \left[R_{t+1} + \gamma \, \mathbb{E} \left[R_{t+2} + \gamma V_\pi(S_{t+2}) \mid S_{t+1}, A_{t+1} = \pi'(S_{t+1}) \right] \mid S_t = s \right] \\ &= \mathbb{E}_{\pi'} \left[R_{t+1} + \gamma R_{t+2} + \gamma^2 V_\pi(S_{t+2}) \mid S_t = s \right] \\ &\leq \mathbb{E}_{\pi'} \left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 V_\pi(S_{t+3}) \mid S_t = s \right] \\ &\;\;\vdots \\ &\leq \mathbb{E}_{\pi'} \left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + ... \mid S_t = s \right] \\ &= \mathbb{E}_{\pi'} \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s \right] \\ &= \mathbb{E}_{\pi'} \left[G_t \mid S_t = s \right] \\ &= v_{\pi'}(s) \\ \therefore \quad v_\pi(s) &\leq v_{\pi'}(s) \end{aligned}$$

- If improvements stop: $q_\pi(s, \pi'(s)) = \max_{a \in A} q_\pi(s, a) = q_\pi(s, \pi(s)) = v_\pi(s)$
- Then the Bellman optimality equation has been satisfied: $v_\pi(s) = \max_{a \in A} q_\pi(s, a)$
- Therefore $v_\pi(s) = v_*(s),\ \forall s \in S$, so π is an optimal policy.

- Essentially, if $v_\pi(s) = v_{\pi'}(s)$ then $v_{\pi'}(s)$ must be $v_*(s)$ & both $\pi(s)$ and $\pi'(s)$ are optimal policies.
- Policy improvement thus must give a **strictly better** policy except when the original policy is already optimal.

4.3. Policy Iteration

- This involves a sequence of **monotonically improving** policies & value functions.
- The algorithm is:
  1. Given a policy, evaluate the policy π to obtain the value function v_π
  2. Then improve the policy by acting greedily w.r.t. v_π to obtain new policy π'
  3. Evaluate new policy π' to obtain new value function $v_{\pi'}$
  4. Repeat until π' converges to the optimal policy π^* (only for finite MDPs)

$$\pi_0 \xrightarrow{E} V_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} V_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} ... \xrightarrow{I} \pi_* \xrightarrow{E} V_*$$

  where $\xrightarrow{E}$ denotes a policy evaluation ($v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s]$) and $\xrightarrow{I}$ denotes a policy improvement ($\pi' = \text{greedy}(v_\pi)$).
- Policy iteration often converges in surprisingly few iterations.

4.4. Value Iteration

- Policy iteration requires full policy evaluation at each iteration step, which is a computationally expensive process that (formally) requires **infinite** sweeps of the state space to approach the true value function.
- Could we truncate policy evaluation without losing the convergence guarantees of policy iteration? **YES**
- In Value Iteration, policy evaluation is stopped after just **one sweep of the state space** (one update of each state).
- It can be written as a particularly simple update operation that combines the policy improvement & truncated policy evaluation steps ($\forall s \in S$):

$$\begin{aligned} v_{k+1}(s) &\doteq \max_a \mathbb{E}\left[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s,\, A_t = a\right] \\ &= \max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma v_k(s')\right] \end{aligned}$$

- Value iteration is obtained simply by turning the Bellman Optimality Equation into an update rule.

4.5. Asynchronous Dynamic Programming

- Async DP algorithms are in-place iterative DP algorithms that are **not** organized in terms of systematic sweeps of the state set. **(localized approach)**
- Async DP updates the values of states **individually** in any order whatsoever, using whatever values of other states happen to be available. **(arbitrary order)**
- 3 simple ideas for async DP:
  - In-place DP
  - Prioritized sweeping
  - Real-time DP

4.6. Generalized Policy Iteration (GPI)

- GPI entails letting policy evaluation and policy improvement interact, independent of the granularity & other details of the two processes.
- Both processes in GPI can be viewed as both competing & cooperating. They compete in the sense that they pull in opposing directions.

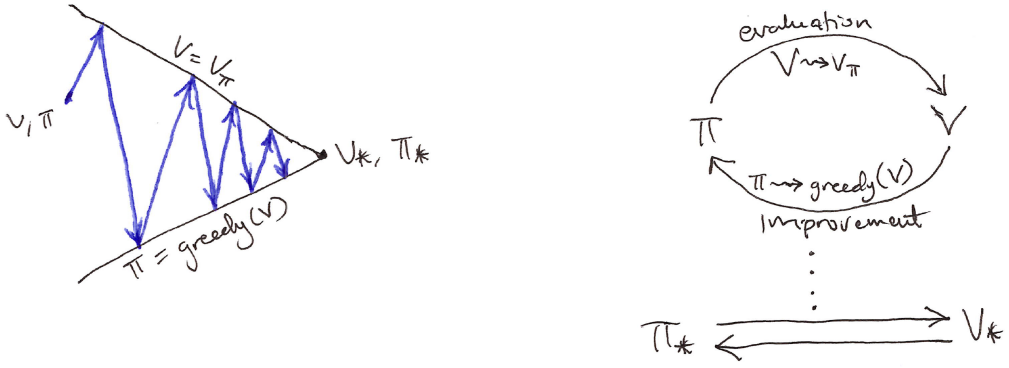

Figure 4.1. Generalized Policy Iteration.

4.7. Efficiency of Dynamic Programming

- DP may not be practical for very large problems, but are actually quite efficient compared with other methods.
- DP is exponentially faster than any direct search in policy could be, because direct search has to be exhaustive.
- For the largest problems, only DP methods are feasible.
- DP is comparatively better suited to handling large state spaces than competing methods such as direct search and linear programming.
- Async DP methods are preferred often on problems with large state spaces.

4.8. Summary

- **Policy evaluation** solves for a fixed policy's value function iteratively, while **policy improvement** produces a strictly better (or already optimal) policy by acting greedily over those values.
- Interleaving evaluation and improvement yields **policy iteration** and **value iteration**, both of which compute exact optimal values and policies when full finite-MDP dynamics are available.
- Standard DP runs full state-space sweeps using **expected updates**, weighting every possible next state by its transition probability until the corresponding **Bellman equations** (v_π, v_*, q_π, q_*) hold to convergence.
- **Generalized Policy Iteration (GPI)** unifies DP and nearly all later RL algorithms as a continuous push-and-pull between improving value estimates and making policies greedier with respect to them.
- **Asynchronous DP** drops full sweeps altogether, updating states in place in arbitrary or priority-driven order while still converging as long as all states continue to be visited.
- Every DP algorithm relies on **bootstrapping** (revising estimates using other estimates). Chapter 5 removes both the model requirement and bootstrapping via Monte Carlo sampling.

5. Monte Carlo Methods

- Monte Carlo methods are used when we do **not** have knowledge of the transition probabilities (model of the environment) and then we must learn directly from experience **(model-free).**
- Learning from **actual** experience is striking because it requires no prior knowledge of the environment's dynamics, yet can still attain optimal behavior.
- **Experience**: sample sequences of states, actions, and rewards from actual/simulated interaction with an environment.
- Monte Carlo methods are only applied to **episodic** tasks.
- MC methods estimate the value function of an **unknown** MDP by **learning** the value function from samples instead of computing from knowledge of the MDP.
- We use Generalized Policy Iteration (**GPI**), developed in Ch. 4, to handle the nonstationarity.
- First consider the prediction problem (computation of V_π & q_π for a fixed arbitrary policy π), then policy improvement, and finally the control problem and its solution by GPI.

5.1. Monte Carlo Prediction

Recall that the value of a state is the **expected cumulative future discounted reward** (expected return). An obvious way of estimating that value from experience is to simply average the returns observed after visits to that state. As more returns are observed, the average should converge to the expected value. This idea underlies all Monte Carlo methods.

- MC uses the simplest possible idea: **Value = mean return**
- MC learns from **complete** episodes: **no bootstrapping**
- MC is **model-free**: no knowledge of MDP transitions/rewards
- **Goal**: Learn v_π from episodes of experience under policy π

$$S_1, A_1, R_2, ..., S_K \sim \pi$$

  - Value function (expected return):

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

  - Return (total discounted reward):

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... + \gamma^{T-1} R_T$$

- The 2 Monte Carlo methods are **First-Visit** & **Every-Visit** MC:
  - **First-Visit**: estimates $v_\pi(s)$ as the average of the returns following **first** visits to s
  - **Every-Visit**: estimates $v_\pi(s)$ by averaging the returns following **all** visits to s
  - Both methods converge to $v_\pi(s)$ as the number of visits (or first visits) to s goes to ∞
  - Every-visit MC extends more naturally to function approximation & eligibility traces

$$\boxed{V(S_t) \leftarrow V(S_t) + \alpha\left[G_t - V(S_t)\right]}$$

5.1.1. First-Visit or Every-Visit Monte Carlo Policy Evaluation

- To evaluate state s
- The first timestep or every timestep t that state s is visited in an episode
- Increment counter: $N(s) \leftarrow N(s) + 1$
- Increment total return: $S(s) \leftarrow S(s) + G_t$

- Value is estimated by **mean return**: $V(s) = S(s)/N(s)$
- By law of large numbers, $V(s) \to V_\pi(s)$ as $N(s) \to \infty$

5.2. Monte Carlo Estimation of Action Values

- Without a model, state-values are not sufficient to determine a policy. The value of each action must be explicitly estimated in order for the values to be useful in suggesting a policy.
- One of the primary goals for Monte Carlo methods is to estimate the state-action value functions, q.
- **Issue**: Many state-action pairs may never be visited under a deterministic policy π.
- **Fix**: For policy evaluation to work for action values, we must assure **continual exploration** of the state space.
  - We can do this by specifying that the episodes **start in a state-action pair**, and that every pair has a nonzero probability of being selected as the start. This guarantees that all state-action pairs will be visited an infinite number of times in the limit of an infinite number of episodes.
  - This is called the assumption of **exploring starts**.
  - Exploring starts fails when learning directly from actual interaction with an environment.
  - A common alternative approach is to consider only policies that are **stochastic with a nonzero probability of selecting all actions in each state.**

5.3. Monte Carlo Control

- Monte Carlo estimation can be used in control to approximate optimal policies via generalized policy iteration (GPI).
- **GPI diagram:**
  - **Policy evaluation**: MC policy eval, $Q = q_\pi$
  - **Policy improvement**: ε-greedy policy improvement

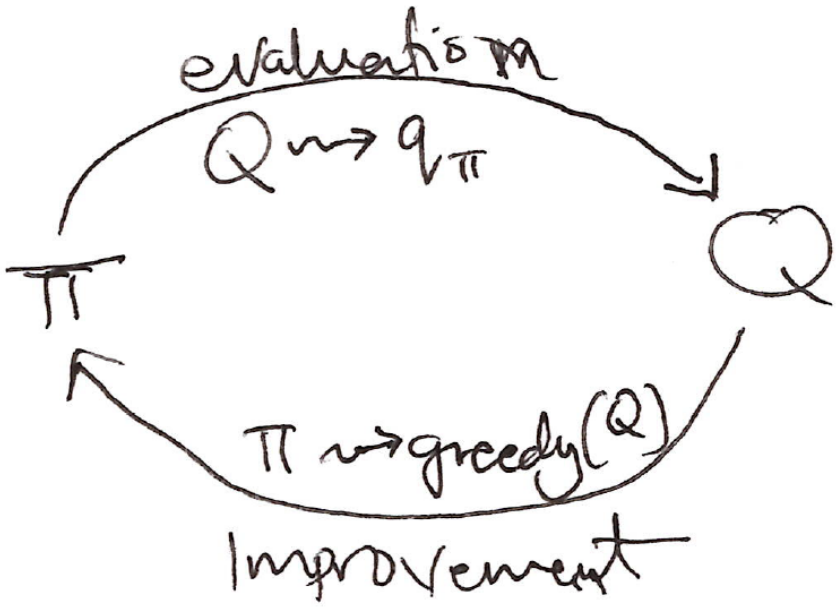

Figure 5.1. Generalized Policy Iteration.

- In GPI, one maintains both an approximate policy and an approximate value function.

$$\pi_0 \xrightarrow{E} q_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} q_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \ldots \xrightarrow{I} \pi_* \xrightarrow{E} q_{\pi_*}$$

where $\xrightarrow{E}$ denotes a complete policy evaluation & $\xrightarrow{I}$ denotes a complete policy improvement.

5.3.1. Policy Improvement

- For q_π, the greedy policy $\forall s \in S$:

$$\pi(s) \doteq \arg\max_a q_\pi(s,a)$$

- Policy improvement theorem applied to π_k & π_{k+1} for all $s \in S$:

$$\begin{aligned} q_{\pi_k}(s, \pi_{k+1}(s)) &= q_{\pi_k}\Big(s, \arg\max_a\ q_{\pi_k}(s,a)\Big) \\ &= \max_a\ q_{\pi_k}(s,a) \\ &\geq q_{\pi_k}(s, \pi_k(s)) \\ &\geq v_{\pi_k}(s) \end{aligned}$$

- The idea of GPI is to avoid the infinite number of episodes nominally required for policy evaluation. This is **value iteration**.
  - Only one iteration of iterative policy evaluation is performed between each step of policy improvement.
- For a deterministic policy for control, it's essential we use exploring starts to ensure sufficient exploration. This creates the **Monte Carlo ES Algorithm**.
  - This is **inefficient** because, for each state-action pair, it maintains a list of **all** returns & repeatedly calculates their mean.
  - More **efficient** would be to maintain just the mean & count for each state-action pair and update them **incrementally**.

5.4. Monte Carlo Control Without Exploring Starts

- How can we avoid the unlikely assumption of exploring starts? (unrealistic assumption)
- To avoid having to use exploring starts, we can use two approaches, **on-policy** and **off-policy** methods. These are the only ways to ensure that all actions are selected infinitely often by the agent continuing to select them.
  - **On-policy**: Evaluate/improve the policy that is used to make the decisions
    - Learn about policy π from experience sampled from π
    - Uses ε-**greedy** policy; generally **soft** policy (assign nonzero probability to each possible action):

$$\pi(a|s) > 0 \qquad \forall s \in S,\ a \in A(s)$$

  - **Off-policy**: Evaluate/improve a policy different from that used to generate the data
    - Learn about target policy π from experience sampled from behavior policy μ

5.4.1. ε-greedy Exploration

$$\pi(a|s) = \left\{ \begin{array}{ll} 1 - \varepsilon + \dfrac{\varepsilon}{|A(s)|} & \text{if greedy action} \\ \dfrac{\varepsilon}{|A(s)|} & \text{if non-greedy action} \end{array} \right\}$$

5.4.2. ε-greedy Policy Improvement

- For any ε-soft policy π, any ε-greedy policy w.r.t. q_π is guaranteed to be better than or equal to π.
- For any ε-soft policy π, any ε-greedy policy π' w.r.t. q_π is an improvement: $v_{\pi'}(s) \geq v_\pi(s)$

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

- By the policy improvement theorem, $\pi' \geq \pi$ (i.e., $v_{\pi'}(s) \geq v_\pi(s)$ for all $s \in S$).
- A policy π is optimal among ε-soft policies if and only if $v_\pi = \tilde{v}_*$
- $\tilde{v}_*$ is the unique solution to the Bellman optimality equation with altered transition probabilities:

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

- When equality holds and the ε-soft policy π is no longer improved, then:

$$
\begin{aligned}
v_\pi(s) &= (1-\varepsilon) \max_a q_\pi(s,a) + \frac{\varepsilon}{|A(s)|} \sum_a q_\pi(s,a) \\
&= (1-\varepsilon) \max_a \sum_{s',r} p(s', r|s, a)\, [r + \gamma v_\pi(s')] + \frac{\varepsilon}{|A(s)|} \sum_a \sum_{s',r} p(s', r|s, a)\, [r + \gamma v_\pi(s')]
\end{aligned}
$$

- Because $\tilde{v}_*$ is the unique solution, it must be that $v_\pi = \tilde{v}_*$.

5.5. Off-Policy Prediction via Importance Sampling

- A huge dilemma is faced when learning control:
  - The need to find the optimal policy, but can only do so by behaving non-optimally in order to explore all actions.
  - On-policy was a compromise; it learns action values not for the optimal policy, but for a near-optimal policy that still explores.
- How can learning control methods learn about the optimal policy while behaving according to an exploratory policy?
- An **off-policy** learning uses 2 policies:
  - A **behavior policy** (used to generate the data; action)
  - A **target policy** (for control; becomes the optimal policy)
- Off-policy learning methods are more powerful and general than on-policy methods.
- Let's consider the prediction problem, in which both target & behavior policies are fixed and given.
  - **Goal**: Estimate v_π or q_π, but all we have are episodes following another policy b, where $b \neq \pi$
    - $\pi \equiv$ target policy
    - $b \equiv$ behavior policy
- In order to use episodes from b to estimate values for π, we require that every action taken under π is also taken, at least occasionally, under b, i.e.,:

$$\pi(a|s) > 0 \implies b(a|s) > 0$$

- This is called the **assumption of coverage**.
- It follows from coverage that:
  - b must be stochastic in states where it is not identical to π
  - π may be deterministic
- Almost all off-policy methods use **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, $A_t, S_{t+1}, A_{t+1}, \dots, S_T$, occurring under any policy π is:

$$\begin{aligned}
&\Pr\{A_t, S_{t+1}, A_{t+1}, \dots, S_T \mid S_t,\ A_{t:T-1} \sim \pi\} \\
&= \pi(A_t|S_t)\, p(S_{t+1}|S_t, A_t)\, \pi(A_{t+1}|S_{t+1}) \cdots p(S_T|S_{T-1}, A_{T-1}) \\
&= \prod_{k=t}^{T-1} \pi(A_k|S_k)\, p(S_{k+1}|S_k, A_k)
\end{aligned}$$

where $p \equiv$ state-transition probability function

- The relative probability of the trajectory under the target and behavior policies (the **importance-sampling ratio**) is:

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

- The state transition probability functions cancel out in the numerator & denominator.
- We want to estimate the expected returns (values) under the target policy, but all we have are returns G_t due to the behavior policy.
  - To address this we simply multiply expected returns from the behavior policy by the importance sampling ratio to get the value function for our target policy:

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

- MC algorithm that averages returns from a batch of observed episodes following policy b to estimate $v_\pi(s)$:
  - $\mathcal{T}(s) \equiv$ set of all time steps in which state s is visited
  - $T(t) \equiv$ 1st time of termination following time t
  - $G_t \equiv$ return after t up through $T(t)$
  - $\{G_t\}_{t \in \mathcal{T}(s)} \equiv$ returns that pertain to state s
  - $\{\rho_{t:T-1}\}_{t \in \mathcal{T}(s)} \equiv$ corresponding importance-sampling ratios

Ordinary importance sampling (unbiased):

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

Weighted importance sampling (biased):

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

5.6. Incremental Implementation

- Monte Carlo prediction methods can be implemented incrementally, on an episode-by-episode basis like the way we did in Chapter 2 (Section 2.4).
- General form:

$$\text{NewEstimate} \leftarrow \text{OldEstimate} + \text{StepSize}\,[\text{Target} - \text{OldEstimate}]$$

- Suppose we have a sequence of returns $G_1, G_2, \ldots, G_{n-1}$, all starting in the same state and each with a corresponding random weight W_i (e.g. $W_i = \rho_{t_i:T(t_i)-1}$). We wish to form the estimate, V_n, and keep it up-to-date as we obtain a single additional return G_n:

$$V_n \doteq \frac{\sum_{k=1}^{n-1} W_k G_k}{\sum_{k=1}^{n-1} W_k}, \quad n \geq 2$$

- In addition to keeping track of V_n, we must maintain for each state the cumulative sum C_n of the weights given to the first n returns.
- The update rule for V_n is:

$$V_{n+1} \doteq V_n + \frac{W_n}{C_n}\left[G_n - V_n\right], \quad n \geq 1$$

$$C_{n+1} \doteq C_n + W_{n+1}$$

where $C_0 \doteq 0$ and $V_1 \rightarrow$ arbitrary.

5.7. Off-Policy Monte Carlo Control

- Using incremental implementation and importance sampling, we can now discuss off-policy Monte Carlo control (follows the behavior policy while learning about and improving the target policy).
- To explore all possibilities, we require that the behavior policy be **soft** (i.e., that it selects all actions in all states with nonzero probability).
- Off-policy Monte Carlo control method based on GPI and weighted importance sampling for estimating π_* & q_*:
  - The target policy $\pi \approx \pi_*$ is the greedy policy w.r.t. Q, which is an estimate of q_π
  - The behavior policy b can be anything, but in order to assure convergence of π to π_*, an infinite number of returns must be obtained for each state-action pair.
    - This can be assured by choosing b to be an ε-soft policy.

5.8. Discounting-aware Importance Sampling

- So far our off-policy methods consider returns as unitary wholes, without taking into account the returns' internal structures as sums of discounted rewards.
- Let's think of discounting as determining a probability of termination, or equivalently a degree of partial termination $\gamma \in (0, 1]$;

$$T \equiv \text{time of termination of the episode}$$

- The **flat partial** returns:

$$\boxed{\bar{G}_{t:h} \doteq R_{t+1} + R_{t+2} + \ldots + R_h, \quad 0 \leq t \leq h \leq T}$$

where "flat" $\equiv$ absence of discounting

"partial" $\equiv$ these returns do not extend all the way to termination,
but instead stop at h, called the **horizon**

- The conventional full return G_t can be viewed as a sum of flat partial returns:

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... + \gamma^{T-t-1} R_T \\ &= (1-\gamma) R_{t+1} + \gamma(1-\gamma)(R_{t+1} + R_{t+2}) + (1-\gamma)\gamma^2 (R_{t+1} + R_{t+2} + R_{t+3}) \\ &\quad + ... + (1-\gamma)\gamma^{T-t-2}(R_{t+1} + R_{t+2} + ... + R_{T-1}) + \gamma^{T-t-1}(R_{t+1} + ... + R_T) \\ &= (1-\gamma) \sum_{h=t+1}^{T-1} \gamma^{h-t-1} \bar{G}_{t:h} + \gamma^{T-t-1} \bar{G}_{t:T} \end{aligned}$$

- Now we need to scale the flat partial returns by an importance sampling ratio that is similarly truncated:

Ordinary importance-sampling estimator:

$$\boxed{V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \left[(1-\gamma) \sum_{h=t+1}^{T(t)-1} \gamma^{h-t-1} \rho_{t:h-1} \bar{G}_{t:h} + \gamma^{T(t)-t-1} \rho_{t:T(t)-1} \bar{G}_{t:T(t)} \right]}{|\mathcal{T}(s)|}}$$

Weighted importance-sampling estimator:

$$\boxed{V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \left[(1-\gamma) \sum_{h=t+1}^{T(t)-1} \gamma^{h-t-1} \rho_{t:h-1} \bar{G}_{t:h} + \gamma^{T(t)-t-1} \rho_{t:T(t)-1} \bar{G}_{t:T(t)} \right]}{\sum_{t \in \mathcal{T}(s)} \left[(1-\gamma) \sum_{h=t+1}^{T(t)-1} \gamma^{h-t-1} \rho_{t:h-1} + \gamma^{T(t)-t-1} \rho_{t:T(t)-1} \right]}}$$

- Both estimators are called **"discounting-aware"** importance sampling estimators.

5.9. Per-decision Importance Sampling

- Another way to account for the structure of the return as a sum of rewards in off-policy importance sampling that may be able to reduce variance even in the absence of discounting (i.e., even if $\gamma = 1$):

$$\begin{aligned} \rho_{t:T-1} G_t &= \rho_{t:T-1} (R_{t+1} + \gamma R_{t+2} + ... + \gamma^{T-t-2} R_T) \\ &= \rho_{t:T-1} R_{t+1} + \gamma \rho_{t:T-1} R_{t+2} + ... + \gamma^{T-t-1} \rho_{t:T-1} R_T \end{aligned}$$

$$\rho_{t:T-1} R_{t+1} = \frac{\pi(A_t|S_t)}{b(A_t|S_t)} \cdot \frac{\pi(A_{t+1}|S_{t+1})}{b(A_{t+1}|S_{t+1})} ... \frac{\pi(A_{T-1}|S_{T-1})}{b(A_{T-1}|S_{T-1})} R_{t+1}$$

$$\mathbb{E}\left[\frac{\pi(A_k|S_k)}{b(A_k|S_k)} \right] \doteq \sum_a b(a|S_k) \frac{\pi(a|S_k)}{b(a|S_k)} = \sum_a \pi(a|S_k) = 1$$

$$\mathbb{E}\left[\rho_{t:T-1} R_{t+1}\right] = \mathbb{E}\left[\rho_{t:t} R_{t+1}\right]$$

$$\mathbb{E}\left[\rho_{t:T-1} R_{t+k}\right] = \mathbb{E}\left[\rho_{t:t+k-1} R_{t+k}\right]$$

$$\mathbb{E}\left[\rho_{t:T-1} G_t\right] = \mathbb{E}\left[\tilde{G}_t\right]$$

where: $\tilde{G}_t = \rho_{t:t} R_{t+1} + \gamma \rho_{t:t+1} R_{t+2} + \gamma^2 \rho_{t:t+2} R_{t+3} + ... + \gamma^{T-t-1} \rho_{t:T-1} R_T$

Ordinary importance-sampling estimator using $\tilde{G}_t$ (lower variance expectation):

$$\boxed{V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \tilde{G}_t}{|\mathcal{T}(s)|}}$$

5.10. Summary

- **Monte Carlo (MC) methods** covered so far learn value functions and optimal policies from experience in the form of **sample episodes,** which lead to at least 3 kinds of advantages over DP methods for MC methods:
  - Learn optimal behavior **directly** from interaction with the environment (**model-free**)
  - Can be used with simulation or **sample models**
  - Are easier and more efficient to **focus** on a small subset of the states
  - Are less harmed by violations of the Markov property (**no bootstrapping**)
- In Chapter 4, we followed GPI overall schema in designing MC control methods.
  - MC methods provide an alternative policy evaluation process, using average of many returns that start in each state as a good approximator for that state value's expected return.
  - MC methods **intermix** policy evaluation and policy improvement steps on an episode-by-episode basis, and can be **incrementally implemented** on an episode-by-episode basis.
- Maintaining **sufficient exploration** is an issue in MC control methods.
  - One fix is by assuming that episodes begin with state-action pairs randomly selected to cover all possibilities. These are called **exploring starts**.
- **On-policy methods** utilize one policy for both action and target selection.
- **Off-policy methods** use a different behavior policy from the deterministic target policy.
- **Off-policy prediction** refers to learning the value function of a target policy from data generated by a **different** behavior policy.
  - These learning methods are based on some form of **importance sampling,** i.e., on weighting returns by the ratio of the probabilities of taking the observed actions under the 2 different policies, thereby transforming their expectations from the **behavior policy** to the **target policy.**
    - **Ordinary importance sampling** uses a simple average of the weighted returns (unbiased estimates)
    - **Weighted importance sampling** uses a weighted average (biased estimates)
- The MC methods in this chapter differ from the DP methods in Chapter 4 in 2 key ways:
  - They operate on sample experience, and thus can be used for direct learning without a model.
  - They do not bootstrap (no update of value estimates on the basis of other value estimates).
- In chapter 6, we'll consider methods that learn from experience, like MC methods, but also bootstrap, like DP methods.

6. Temporal-Difference Learning

- TD methods learn directly from raw experience.
- TD is **model-free** (no model of the environment's dynamics).
- TD methods update estimates based on other learned estimates; in other words TD learns from **incomplete** episodes. This is called **bootstrapping**.

6.1. TD Prediction

Goal: Learn/estimate V_π from experience under policy π.

- Both TD and Monte Carlo (MC) methods use experience to solve the prediction problem.
- A simple incremental every-visit MC suitable for nonstationary environments is:

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

where $G_t \equiv$ actual return following time t

$\alpha \equiv$ constant step-size parameter

- The simplest TD method, **TD(0)**, makes the update:

$$\boxed{V(S_t) \leftarrow V(S_t) + \alpha \underbrace{\left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right]}_{\text{TD Error}}}$$

where $R_{t+1} + \gamma V(S_{t+1}) \equiv$ the **TD target**



Figure 6.1. **TD(0) backup diagram.** one-step lookahead from state s through action to next state s'.

6.1.1. Pseudocode: Tabular TD(0) for Estimating v_π

Input: the policy π to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$ for all $s \in \mathcal{S}^+$, arbitrarily except that $V(\text{terminal}) = 0$
Loop for each episode:
  Initialize S
  Loop for each step of episode:
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$
    $S \leftarrow S'$
  Until S is terminal

- TD(0) bases its update in part on an existing estimate (bootstrapping):

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}\left[G_t \mid S_t = s\right] \\ &= \mathbb{E}_\pi\left[R_{t+1} + \gamma G_{t+1} \mid S_t = s\right] \\ &= \mathbb{E}_\pi\left[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s\right] \end{aligned}$$

- The TD target is an estimate because it samples the expected values in the equation above, and it uses the current estimate V instead of the true V_π.
- TD methods **combine** the sampling of MC with the bootstrapping of DP.

6.1.2. TD Error & its Relationship to MC Error

The TD error at each time is the error in the estimate **made at that time** because the TD error depends on the next state and next reward. If the array V does not change during the episode (like in MC methods), then:

$$\text{MC error} = \sum_t \text{TD error}$$

Let $\delta_t \doteq R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ (the TD error). Then:

$$
\begin{aligned}
G_t - V(S_t) &= R_{t+1} + \gamma G_{t+1} - V(S_t) + \gamma V(S_{t+1}) - \gamma V(S_{t+1}) \\
&= \underbrace{R_{t+1} + \gamma V(S_{t+1}) - V(S_t)}_{\delta_t} + \gamma G_{t+1} - \gamma V(S_{t+1}) \\
&= \delta_t + \gamma \left(G_{t+1} - V(S_{t+1}) \right) \\
&= \delta_t + \gamma \left(R_{t+2} + \gamma G_{t+2} - V(S_{t+1}) \right) \\
&= \delta_t + \gamma \left(R_{t+2} + \gamma V(S_{t+2}) - V(S_{t+1}) + \gamma G_{t+2} - \gamma V(S_{t+2}) \right) \\
&= \delta_t + \gamma \underbrace{\left(R_{t+2} + \gamma V(S_{t+2}) - V(S_{t+1}) \right)}_{\delta_{t+1}} + \gamma^2 \left(G_{t+2} - V(S_{t+2}) \right) \\
&= \delta_t + \gamma \delta_{t+1} + \gamma^2 \left(G_{t+2} - V(S_{t+2}) \right) \\
&= \delta_t + \gamma \delta_{t+1} + \gamma^2 \left(R_{t+3} + \gamma V(S_{t+3}) - V(S_{t+2}) + \gamma G_{t+3} - \gamma V(S_{t+3}) \right) \\
&= \delta_t + \gamma \delta_{t+1} + \gamma^2 \delta_{t+2} + \gamma^3 \left(G_{t+3} - V(S_{t+3}) \right) \\
&\vdots \\
&= \delta_t + \gamma \delta_{t+1} + \gamma^2 \delta_{t+2} + \gamma^3 \delta_{t+3} + ... + \gamma^{T-t-1} \delta_{T-1} + \gamma^{T-t} \underbrace{\left[G_T - V(S_T) \right]}_{=0}
\end{aligned}
$$

$$
\boxed{G_t - V(S_t) = \sum_{k=t}^{T-1} \gamma^{k-t} \, \delta_k}
$$

6.2. Advantages of TD Prediction Methods

- TD methods are model-free, unlike DP methods.
- TD methods can learn **before** knowing the final outcome & **without** the final outcome, unlike MC:
  - **before:** TD can learn online, incrementally, while MC must wait until the episode ends when the return is known.
  - **without:** TD can learn from incomplete sequences/episodes while MC only learns from complete sequences.
  - TD works in continuing (non-terminating) environments while MC only works for episodic/terminating environments.
- TD(0) converges to $v_\pi(s)$ for a fixed policy π, but not always for general function approximation.
- TD methods are usually **more efficient** than MC methods (constant α) on stochastic tasks; they converge faster.
- TD methods have **low variance**, while MC methods have **high variance**.

6.3. Optimality of TD(0)

- For a finite number of episodes, we can run our methods repeatedly with the same data (finite episodes) until convergence as we continually update the value function by the sum of each of the increments at the end of each batch.
- This is called **batch updating** because updates are made only after processing each complete batch of training data / finite number of episodes.
- Under batch updating, **TD(0) & constant-α MC methods** converge deterministically to a single answer independent of α, as long as α is chosen to be sufficiently small.
- In general, batch MC methods always find the estimates that **minimize mean square error** on the training set, whereas batch TD(0) always finds the estimates that would be exactly correct for the **maximum-likelihood model of the Markov process.**

- Given this model, we can compute the estimate of the value function that would be exactly correct if the model were exactly correct.
- 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.
- In general, **batch TD(0) converges to the certainty-equivalence estimate.**

6.4. Sarsa: On-policy TD Control

- Using TD prediction methods for control, we'll follow Generalized Policy Iteration (GPI), only this time using TD methods for the policy evaluation part (predicting the value function).
- For on-policy control, we learn an action-value function $q_\pi(s, a)$ for our current policy π, for all states s and actions a. Recall that an episode consists of an alternating sequence of states and state–action pairs:

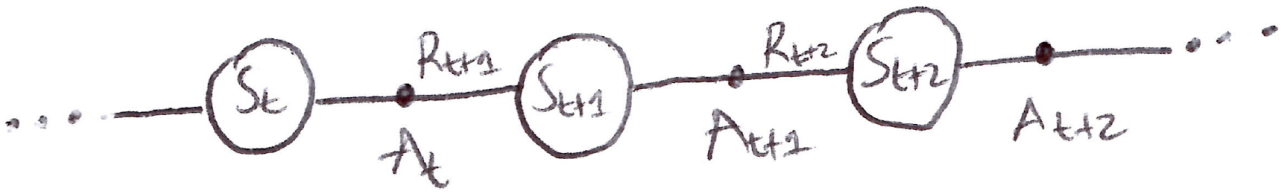

Figure 6.2. An episode.

$$\boxed{Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[\underbrace{R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})}_{\text{TD target}} - Q(S_t, A_t)\Big]}$$

- This update rule uses a quintuple of events $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$. Hence, the name **SARSA**. The backup diagram is shown below:
- As in all on-policy methods, we continually estimate q_π for the behavior policy π & simultaneously update the policy π greedily w.r.t. q_π.
- The convergence properties of the SARSA algorithm depend on the nature of the policy's dependence on Q. For example, one could use ε-greedy / ε-soft policies.
- **Goal**: Use TD in the control loop
  - apply TD to $Q(s, a)$,
  - use ε-greedy policy improvement,
  - update with every time-step.
- Every time step:
  - **Policy Evaluation:** SARSA, $Q \approx q_\pi$
  - **Policy Improvement:** ε-greedy

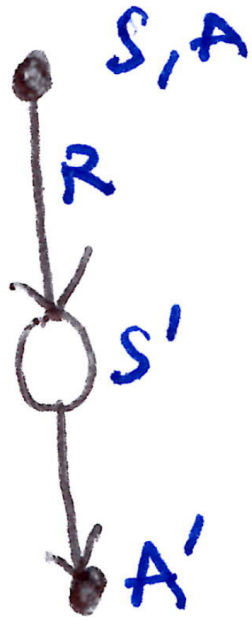

Figure 6.3. **Sarsa backup diagram.** from state-action pair (S, A) through reward R to next state S' and next action A'.

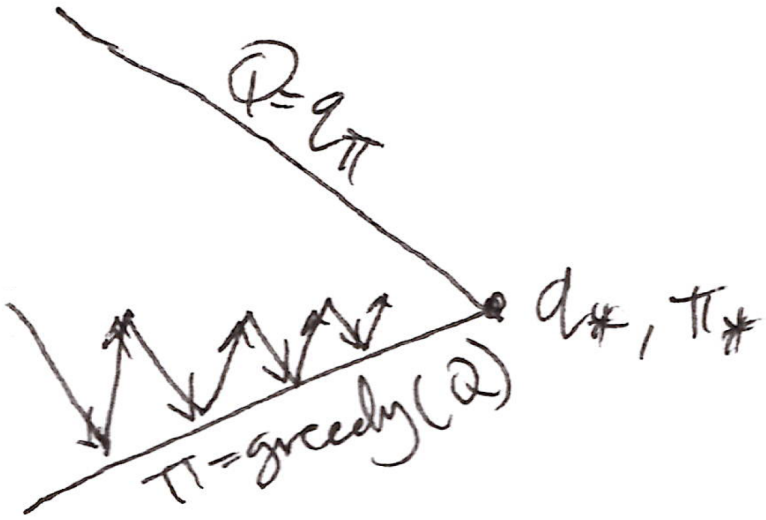

Figure 6.4. SARSA control.

6.4.1. Pseudocode: SARSA for estimating $Q \approx q_*$

Algorithm parameters: $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$ for all $s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$, arbitrarily except that $Q(\text{terminal}, \cdot) = 0$
Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g. ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g. ε-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$
    $S \leftarrow S'$
    $A \leftarrow A'$
  Until S is terminal

6.5. Q-learning: Off-Policy TD Control

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[\underbrace{R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a')}_{\text{Q-learning target}} - Q(S_t, A_t)\Big]$$

- Here, the learned action-value function Q directly approximates q_*, the optimal value function, independent of the policy being followed (off-policy). v_* can be learned with a **1-step lookahead.**
- Q-learning is off-policy so it updates the value function **greedily** directly while updating/exploring the other actions with the ε**-greedy** policy.
  - This simplifies the analysis of the algorithm & enables early convergence proofs.
  - Under the **GLIE** assumption (all state-action pairs continue to be updated) and following a **stochastic approximation theory** variant on the sequence of α, Q has been shown to converge with probability 1 to q_*:

$$Q(s, a) \longrightarrow q_*(s, a)$$

6.5.1. Q-learning (Consider Off-policy Learning of $Q(s, a)$)

- Next action is chosen using the behavior policy:

$$A_{t+1} \sim \mu(\cdot|S_t)$$

- But we consider an alternative successor action $A' \sim \pi(\cdot|S_t)$ and update $Q(S_t, A_t)$ towards the value of that alternative action, as seen in the update equation:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[\underbrace{R_{t+1} + \gamma Q(S_{t+1}, A')}_{\text{Q-learning target}} - Q(S_t, A_t)\Big]$$

- Employs value iteration.

6.5.2. Off-policy Control with Q-learning

- We improve both policies (target & behavior):

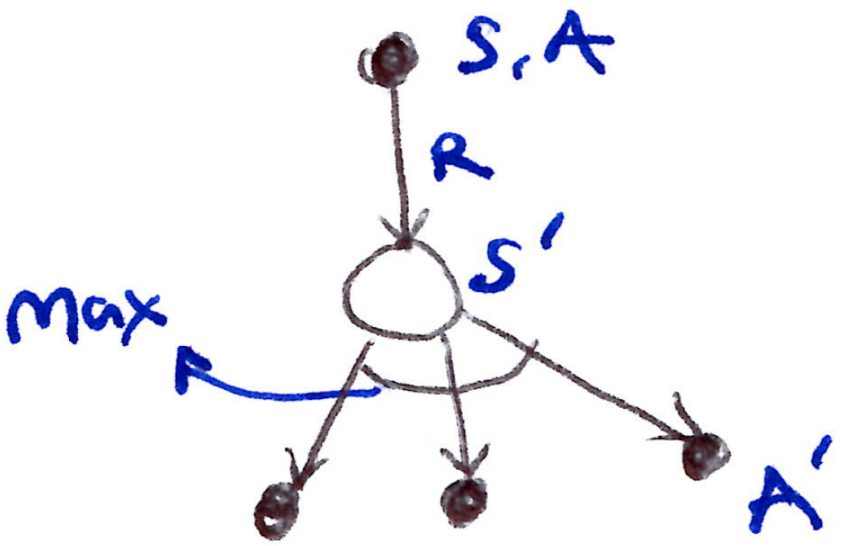

Figure 6.5. **Q-learning backup diagram.** from state-action (S, A) through reward R to next state S', with a max over all next actions A'.

  - Improve target policy π **greedily** w.r.t. $Q(s, a)$:

$$\pi(S_{t+1}) = \arg\max_{a'} Q(S_{t+1}, a')$$

  - Improve behavior policy μ ε**-greedily** w.r.t. $Q(s, a)$
- The Q-learning target then simplifies:

$$R_{t+1} + \gamma Q(S_{t+1}, A') = R_{t+1} + \gamma Q\Big(S_{t+1}, \arg\max_{a'} Q(S_{t+1}, a')\Big) = R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a')$$

6.5.3. Pseudocode: Q-learning for estimating $\pi \approx \pi_*$

Algorithm parameters: $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$ for all $s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$, arbitrarily except that $Q(\text{terminal}, \cdot) = 0$
Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g. ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\Big[R + \gamma \max_a Q(S', a) - Q(S, A)\Big]$
    $S \leftarrow S'$
  Until S is terminal

6.6. Expected Sarsa

- Instead of updating $Q(S, A)$ with the value maximising action at S_{t+1}, $\max_a Q(S_{t+1}, a)$ as used in Q-learning, or the ε-greedy exploratory action as in SARSA, we can use the **expected value** of Q at S_{t+1}, taking into account how likely each action is under the current policy. The update rule is:

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

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

- Given the next state S_{t+1}, this algorithm moves **deterministically** in the same direction as SARSA moves in **expectation,** hence the name **Expected SARSA**.
- Expected SARSA eliminates the variance due to the random selection of A_{t+1}.
- Given the same amount of experience, Expected SARSA generally performs better than SARSA.
- Expected SARSA is an on-policy learning method, but if we change our target policy such that π is the greedy policy while our behavior is more exploratory, then it becomes an off-policy algorithm.
  - Expected SARSA becomes Q-learning if π is the greedy policy and our behavior policy (μ) is more exploratory (ε-greedy).
- Essentially, it is a generalization of Q-learning that has a reliable improvement over SARSA.

6.7. Maximization Bias & Double Learning

- All the control algorithms discussed so far utilize **maximization operations** in target policy constructions (ε-greedy or greedy action selection).
- This implicitly **favors positive numbers** (significant positive bias).
- If the true values of state-action pairs $q(s, a)$ of a single state s with multiple actions a are all zero, i.e. $q(s, a) = 0 \; \forall s, a$, but whose estimated values $Q(s, A)$ are uncertain and thus distributed some above & some below zero, then the maximum of the true values is zero, but the maximum of the estimates is positive; hence a positive bias.
- This is called the **maximization bias**:

$$q(s, a) = 0 \quad \forall s, a \implies \max_{s,a} q(s, a) = 0$$

$$Q(s, a) = \left\{ \begin{array}{l} \geq 0 \\ < 0 \end{array} \right\} \implies \max_{s,a} Q(s, a) > 0$$

- We can avoid maximization bias by learning 2 independent estimates of the value function $Q_1(a), Q_2(a)$, in which we could then:
  - Use one estimate, e.g. Q_1, to determine the maximizing action: $A^* = \arg\max_a Q_1(a)$
  - Then use the other, Q_2, to estimate its value: $Q_2(A^*) = Q_2(\arg\max_a Q_1(a))$
- This idea is called **Double Learning**.
  - It doubles the memory requirements, but does not increase the amount of computation per step.
- The Q-value estimate is unbiased in the sense that $\mathbb{E}\left[Q_2(A^*) \right] = q(A^*)$.
- Double Learning for Q-learning is called **Double Q-learning.**

6.7.1. Pseudocode: Double Q-learning for estimating $Q_1 \approx Q_2 \approx q_*$

Algorithm parameters: $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q_1(s, a)$ & $Q_2(s, a)$ for all $s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$ s.t. $Q(\text{terminal}, \cdot) = 0$
Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using ε-greedy policy in $Q_1 + Q_2$
    Take action A, observe R, S'
    With probability 0.5:

$$Q_1(S, A) \leftarrow Q_1(S, A) + \alpha\Big[R + \gamma Q_2\Big(S', \arg\max_a Q_1(S', a)\Big) - Q_1(S, A)\Big]$$

    else:

$$Q_2(S, A) \leftarrow Q_2(S, A) + \alpha\Big[R + \gamma Q_1\Big(S', \arg\max_a Q_2(S', a)\Big) - Q_2(S, A)\Big]$$

    $S \leftarrow S'$
  Until S is terminal

Double Q-learning Update Rule:

$$\boxed{Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha\left[R_{t+1} + \gamma\, Q_2\Big(S_{t+1}, \arg\max_a Q_1(S_{t+1}, a)\Big) - Q_1(S_t, A_t)\right]}$$

6.8. Games, Afterstates & Other Special Cases

- Our general approach so far revolves around learning an action-value function, where the value of an action in a given state is quantified/evaluated **before** the action is taken.
- In some specialized cases, it is expedient to evaluate the value function **after** an action.
  - The state-value function that evaluates states after the action are referred to as **afterstates** & the value functions over these afterstates are called the **afterstate value functions**.
  - This is the case in Tic-tac-toe and chess (evaluate board position after agent has made its move).
- Afterstates are useful when we have partial knowledge (initial part) of the environment's dynamics but not full dynamics.
- It is also useful where many different actions may lead to the same state afterward.
- Afterstate methods are aptly described in terms of GPI, with a policy and afterstate value function interacting in essentially the same way.
  - Still have to decide between on or off policy for managing the need for persistent exploration.

6.9. Summary

- **TD methods** learn from incomplete episodes and are alternatives to MC methods for solving the prediction part of the RL problem.
- They use GPI to solve the control part of the RL problem.
- When issues arise with maintaining sufficient exploration in prediction, TD control methods deal with this issue via an on-policy or off-policy approach:
  - **Sarsa** (on-policy)
  - **Q-learning** (off-policy)
  - **Expected Sarsa** (off-policy)

  $\Rightarrow$ ***PROs:*** great simplicity, single equations that can be easily implemented
- The TD methods used here in this chapter should be rightly called **one-step, tabular, model-free** TD methods.

7. n-step Bootstrapping

- n-step methods generalize both MC methods and one-step TD methods so that one can shift from one to the other smoothly as needed to meet the demands of a particular task.
- n-step methods span a spectrum with MC methods at one end and one-step TD methods at the other.
  - The best methods are often intermediate between the 2 extremes.
- n-step methods enable bootstrapping to occur over multiple steps, freeing us from the tyranny of the single time step.
  - The idea of n-step methods is used as an introduction to the algorithmic idea of **eligibility traces**, which enable bootstrapping over multiple time intervals simultaneously.

7.1. n-step TD Prediction, TD(λ)

- Let the TD target look n steps into the future.
- Consider estimating V_π from sample episodes generated using π:
  - **MC** methods perform an update for each visited state based on the entire sequence of observed rewards from that state until the end of the episode.
  - **One-step** TD methods update is based on the one next reward only.
  - **n-step**: somewhere in between that performs an update based on an intermediate number of rewards (more than 1 but less than all the rewards in the entire episode).

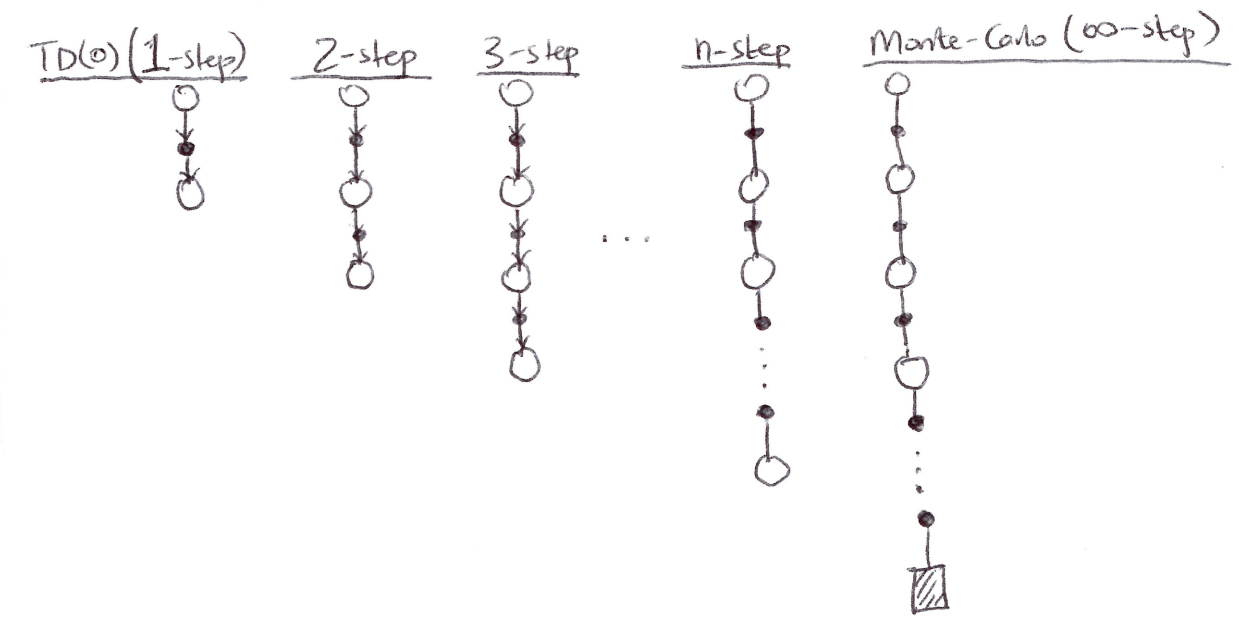

Figure 7.1. Backup diagrams for n-step methods.

- n-step methods are methods in which the temporal difference extends over n steps.
- The n-step return looks like:

$$G_t^{(n)} = G_{t:t+n} \equiv \text{the return from time step } t \text{ to } t+n$$

$$n = 1 \text{ (TD(0))}: \quad G_t^{(1)} = R_{t+1} + \gamma V_t(S_{t+1})$$

$$n = 2: \quad G_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 V_{t+1}(S_{t+2})$$

$$\vdots$$

$$n = \infty \text{ (MC)}: \quad G_t^{(\infty)} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{T-1} R_T$$

- **MC target** is the return.
- **TD(0) target** is the one-step return, the first reward plus the discounted next state estimated value ($V_t : S \to \mathbb{R}$ is the estimate at time t of v_π).
- **n-step return:**

$$\boxed{G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n}),}$$

$$\forall n, t \quad \text{s.t.} \quad n \geq 1 \text{ and } 0 \leq t \leq T - n$$

- All n-step returns can be considered approximations 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, and the n-step return becomes the ordinary full return:

$$G_{t:t+n} = G_t^{(n)} \doteq G_t \quad \text{if } t + n \geq T$$

- n-step returns for $n > 1$ involve future rewards and states that are not available at the transition time from $t \to t+1$. Hence the n-step return cannot be used until after it has seen R_{t+n} and computed V_{t+n-1} when they are both available at time $t + n$.
  - The natural state-value learning algorithm for using n-step returns is thus:

$$V_{t+n}(S_t) \doteq V_{t+n-1}(S_t) + \alpha \left[G_t^{(n)} - V_{t+n-1}(S_t) \right], \quad 0 \leq t < T$$

  - while the values of all other states remain unchanged:

$$V_{t+n}(s) = V_{t+n-1}(s), \quad \forall s \neq S_t$$

  - This algorithm is the **n-step TD**:

$$\boxed{V(S_t) \leftarrow V(S_t) + \alpha \left[G_t^{(n)} - V(S_t) \right]}$$

- The n-step return uses the value function V_{t+n-1} to correct for the 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 than V_{t+n-1} is in a worst-state sense. This is called the **error reduction property**. Using this property, all n-step TD methods converge to the correct predictions:

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

7.2. n-step Sarsa, Sarsa(λ)

- n-step value function prediction can be extended to control.
- Switch value function updates based on state rewards to state-action value function updates based on state-action pairs and then use an ε-greedy policy.
- Backup diagrams for n-step Sarsa are similar to those of n-step TD, consisting of strings of alternating states and actions, except that the Sarsa ones all start and end with an action rather than a state.

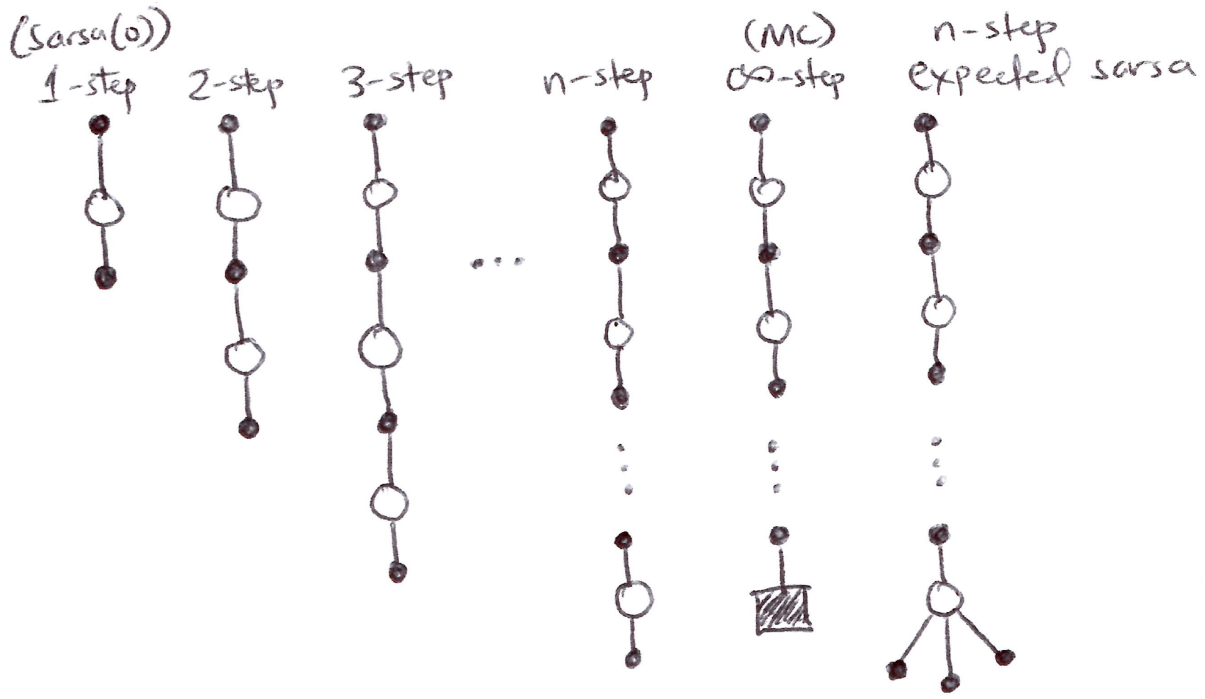

Figure 7.2. Backup diagrams for n-step SARSA methods.

- Consider the following n-step returns for $n = 1, 2, ..., \infty$:

$$G_t^{(n)} = G_{t:t+n}$$

$$n = 1 \text{ (Sarsa(0))}: \quad G_t^{(1)} = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$$

$$n = 2: \quad G_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 Q(S_{t+2}, A_{t+2})$$

$$\vdots$$

$$n = \infty \text{ (MC)}: \quad G_t^{(\infty)} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{T-1} R_T$$

- **n-step Q-return:**

$$\boxed{G_t^{(n)} \doteq R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n Q_{t+n-1}(S_{t+n}, A_{t+n})}$$

$$\forall n, t \quad \text{s.t.} \quad n \geq 1,\ 0 \leq t \leq T - n, \quad \text{with } G_t^{(n)} = G_t \text{ if } t + n \geq T$$

- The natural state-action value learning algorithm is then:

$$Q_{t+n}(S_t, A_t) \doteq Q_{t+n-1}(S_t, A_t) + \alpha \left[G_t^{(n)} - Q_{t+n-1}(S_t, A_t) \right], \quad 0 \leq t < T$$

$$Q_{t+n}(s, a) = Q_{t+n-1}(s, a) \quad \forall s, a \quad \text{s.t.} \quad s \neq S_t \text{ or } a \neq A_t$$

- This algorithm is called **n-step Sarsa**:

$$\boxed{Q_{t+n}(S_t, A_t) \leftarrow Q_{t+n-1}(S_t, A_t) + \alpha \left[G_t^{(n)} - Q_{t+n-1}(S_t, A_t)\right]}$$

7.2.1. Expected Sarsa

- Similar to n-step Sarsa (linear string of sample actions and states), except that its last element is a branch over all action possibilities weighted by their probability under π.
- The n-step return here is:

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n \bar{V}_{t+n-1}(S_{t+n}), \quad t + n < T$$

$$\text{with } G_t^{(n)} = G_t \quad \text{if } t + n \geq T$$

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

$$\bar{V}_t(s) \doteq \sum_a \pi(a|s)\, Q_t(s, a), \quad \forall s \in S$$

- Expected approximate values are essential for action-value methods in RL.
- If s is terminal, then its expected approximate value is defined to be zero:

$$\bar{V}_t(\mathcal{T}) = 0$$

7.3. n-step Off-policy Learning

- Recall that off-policy learning entails learning the value function for one policy π, while following another policy b (behavior policy).
  - Often π is the greedy policy for the current $Q(s, a)$ estimate (target policy).
  - b is a more exploratory policy, perhaps ε-greedy.
- In order to use the data from b, we must account for the **difference between the two policies**, using their relative probability of taking the actions that were taken.
- In n-step methods, returns are constructed over n steps, so we are interested in the relative probability of just those n actions.
- For example, to make a simple off-policy n-step TD, we can weight the update for time t (made at time $t + n$ actually) by the importance sampling ratio:

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

$$\text{where } \rho_{t:h} \equiv \text{relative probability under the two policies of taking the } n \text{ actions from } A_t \text{ to } A_{t+n-1}$$

- The off-policy n-step TD update when weighted by $\rho_{t:h}$ becomes:

$$\boxed{V_{t+n}(S_t) \leftarrow V_{t+n-1}(S_t) + \alpha\, \rho_{t:t+n-1} \left[G_t^{(n)} - V_{t+n-1}(S_t)\right], \quad 0 \leq t < T}$$

- If both policies are actually the same (on-policy), then the importance sampling ratio is always 1 and we have our previous update (on-policy n-step TD):

$$\text{if } \pi = b,\ \rho_{t:h} = 1 \implies V(S_t) \leftarrow V(S_t) + \alpha \left[G_t^{(n)} - V(S_t)\right]$$

- Cannot use importance sampling if $b = 0$ & $\pi \neq 0$.
- Similarly, the n-step Sarsa update can be extended for the off-policy method:

$$Q_{t+n}(S_t, A_t) \leftarrow Q_{t+n-1}(S_t, A_t) + \alpha\, \rho_{t+1:t+n} \left[G_t^{(n)} - Q_{t+n-1}(S_t, A_t)\right]$$

- Note that the importance sampling ratio here starts and ends one step later than for n-step TD because we are updating a state-action pair:
  - (n-step Sarsa) $\rho_{t+1:t+n}$ vs. $\rho_{t:t+n-1}$ (n-step TD)
  - We know that we have selected the action so we need importance sampling only for the subsequent actions.
- For the n-step Expected Sarsa off-policy version, the update is:

$$\boxed{Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\, \rho_{t+1:t+n-1} \left[G_t^{(n)} - Q(S_t, A_t)\right]}$$

- This is the same as the n-step Sarsa update except that the importance sampling ratio used is $\rho_{t+1:t+n-1}$ instead of $\rho_{t+1:t+n}$, because in Expected Sarsa all possible actions are taken into account in the last state; the one actually taken has no effect and does not need to be corrected for.

7.4. Per-decision Methods with Control Variates

- The multi-step off-policy methods presented in section 7.3 are simple and conceptually clear but not very efficient and dramatically increase variance.
- A more sophisticated approach would be using per-decision importance sampling.
  1. The ordinary n-step return can be written recursively. For the n-steps ending at horizon h, the n-step return can be written:

$$G_{t:h} = R_{t+1} + \gamma G_{t+1:h}, \quad t < h < T$$

$$\text{where } G_{h:h} \doteq V_{h-1}(S_h)$$

  2. Consider the effect of following a behavior policy b that is not the same as the target policy π. All of the resulting experiences, including the first reward R_{t+1} and the next state S_{t+1}, must be weighted by the importance sampling ratio for time t:

$$\rho_t = \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$$

- A simple approach would be to weight the RHS ($R_{t+1} + \gamma G_{t+1:h}$) by ρ_t, but a more sophisticated approach would be to use an off-policy definition of the n-step return ending at horizon h:

$$\boxed{\begin{aligned} &G_{t:h} \doteq \rho_t \left(R_{t+1} + \gamma G_{t+1:h}\right) + (1 - \rho_t) V_{h-1}(S_t), \quad t < h < T \\ &G_{h:h} \doteq V_{h-1}(S_h) \end{aligned}}$$

- If $\rho_t = 0$ (trajectory has no chance of occurrence under the target policy), then instead of the target $G_{t:h}$ being zero and causing the estimate to shrink, the target is the same as the estimate and causes no change.
- The additional term $(1 - \rho_t)V_{h-1}(S_t)$ is called a **control variate** and essentially ensures we ignore the sample when the importance sampling ratio is zero, thereby leaving the estimate unchanged.
  - The control variate does not change the expected update.
  - The importance sampling ratio has an expected value of 1 and is uncorrelated with the estimate, so the expected value of the control variate is zero.

7.4.1. For Action Values

- For action values, the off-policy definition of the n-step return is a little different because the first action does not play a role in the importance sampling (it has already been taken).
- The n-step on-policy return ending at horizon h can be written recursively, and for action values the recursion ends with $G_{h:h} \doteq V_{h-1}(S_h)$. An off-policy form with control variates is:

$$
\begin{aligned}
G_{t:h} &\doteq R_{t+1} + \gamma \left[\rho_{t+1} G_{t+1:h} + \bar{V}_{h-1}(S_{t+1}) - \rho_{t+1} Q_{h-1}(S_{t+1}, A_{t+1}) \right] \\
&= R_{t+1} + \gamma \, \rho_{t+1} \left(G_{t+1:h} - Q_{h-1}(S_{t+1}, A_{t+1}) \right) + \gamma \bar{V}_{h-1}(S_{t+1}), \quad t < h \leq T
\end{aligned}
$$

- If $h < T$, then the recursion ends with $G_{h:h} \doteq Q_{h-1}(S_h, A_h)$.
- If $h \geq T$, the recursion ends with $G_{T-1:h} = R_T$.
- The resultant prediction algorithm (after combining with n-step Sarsa update rule) is analogous to Expected Sarsa.
- Importance sampling results in high variance updates. Off-policy training is inevitably slower than on-policy training. Ways to reduce the variance include:
  1. Control Variates
  2. Adapt the step size to the observed variance (Autostep Method)
  3. Invariant updates of Karampatziakis & Langford [2010]
  4. The usage technique by Mahmood [2017; Mahmood & Sutton, 2015]

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

- Off-policy learning without importance sampling:
  - Q-learning & Expected Sarsa do it for the one-step case.
  - The n-step case is called the **tree-backup algorithm**.
- **3-step tree backup** diagram:
  - 3 sample states & rewards.
  - 2 sample actions.
  - A list of unselected actions for each state.
  - We have no sample for the unselected actions, so therefore we bootstrap using their estimated values.

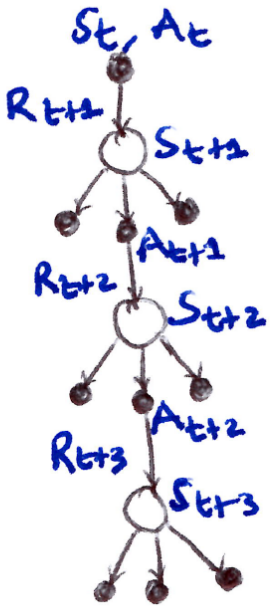

Figure 7.3. The 3-step tree-backup update.

- The target for the update of a node was obtained by combining the rewards along the way and the estimated values of the nodes at the bottom.
  - Hence the name, **tree-backup update**: an update from the entire tree of estimated action values.
  - More precisely, the update is from the estimated action values of the **leaf nodes** of the tree.

  - The **interior** action nodes, corresponding to the actual actions taken, do not participate.
- Each leaf node contributes to the target with a weight proportional to its probability of occurring under the target policy π:
  - Each first-level action a contributes with a weight of $\pi(a|S_{t+1})$.
  - The action actually taken, A_{t+1}, does not contribute at all and its probability $\pi(A_{t+1}|S_{t+1})$ is used to weight all the 2nd-level action values.
  - Each non-selected 2nd-level action a' contributes with weight $\pi(A_{t+1}|S_{t+1})\,\pi(a'|S_{t+2})$.
  - Each 3rd-level action a'' contributes with weight $\pi(A_{t+1}|S_{t+1})\,\pi(A_{t+2}|S_{t+2})\,\pi(a''|S_{t+3})$, and so on.
- Essentially each arrow to an action node in the diagram is weighted by the action's probability of being selected under the target policy and, if there's a tree below the action, then that weight applies to all leaf nodes in the tree.

7.5.1. Detailed Equations for the n-step Tree-Backup Algorithm

- The one-step return (target) is the same as that of Expected Sarsa:

$$G_{t:t+n} = G_t^{(n)}$$

$$G_t^{(1)} \doteq R_{t+1} + \gamma \sum_a \pi(a|S_{t+1})\, Q_t(S_{t+1}, a), \quad \text{for } t < T-1$$

- The two-step tree-backup return is, for $t < T-2$:

$$G_t^{(2)} \doteq R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1})\, Q_{t+1}(S_{t+1}, a) + \gamma\pi(A_{t+1}|S_{t+1}) \left(R_{t+2} + \gamma \sum_a \pi(a|S_{t+2})\, Q_{t+1}(S_{t+2}, a) \right)$$

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

- The n-step tree-backup return is, for $t < T-1,\ n \geq 2$:

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

  - becomes one-step tree-backup return when $n = 1$ and $G_{T-1:t+n} = R_T$.
- We can use $G_t^{(n)}$ in the Q-update from n-step Sarsa, for $0 \leq t < T$:

$$\boxed{Q_{t+n}(S_t, A_t) \leftarrow Q_{t+n-1}(S_t, A_t) + \alpha \left[G_t^{(n)} - Q_{t+n-1}(S_t, A_t) \right]}$$

- while the values of all other state-action pairs remain unchanged:

$$Q_{t+n}(s, a) = Q_{t+n-1}(s, a) \quad \forall s, a \quad \text{s.t.} \quad s \neq S_t \text{ or } a \neq A_t$$

7.6. A Unifying Algorithm: n-step $Q(\sigma)$

- So far we have considered 3 different kinds of action-value algorithms:
  - **n-step Sarsa** has all sample transitions.
  - **Tree-backup algorithm** has all state-to-action transitions fully branched without sampling.
  - **n-step Expected Sarsa** has all sample transitions except for the last state-to-action one which is fully branched with an expected value.
- **Unification Algorithm** $\Rightarrow$ *One Idea*: Decide on a step-by-step basis whether we want to sample, as in Sarsa, or consider the expectation over all actions instead, as in tree-backup update.
- In unification lies many possibilities as seen/suggested in the figure above.
- To increase the possibilities even further we can consider a continuous variation between sampling and expectation:

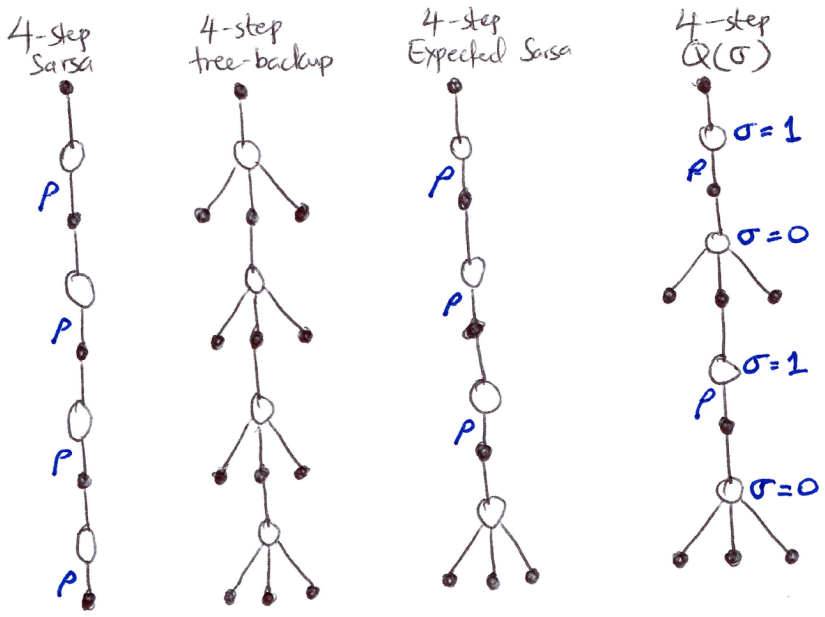

Figure 7.4. Backup diagram for different kinds of n-step Q updates.

  - Let $\sigma_t \in [0, 1]$ denote the degree of sampling on step t:
    - $\sigma = 1 \Rightarrow$ full sampling
    - $\sigma = 0 \Rightarrow$ pure expectation with no sampling
  - The random variable σ_t might be set as a function of the state, or state-action pair, at time t.
- This algorithm is called n**-step** $Q(\sigma)$.
- For the equations, first the tree-backup n-step return is expressed in terms of the horizon $h = t + n$ and the expected approximate value $\bar{V}$:

$$\begin{aligned} G_{t:h} &= R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1})\, Q_{h-1}(S_{t+1}, a) + \gamma\pi(A_{t+1}|S_{t+1})\, G_{t+1:h} \\ &= R_{t+1} + \gamma \bar{V}_{h-1}(S_{t+1}) - \gamma\pi(A_{t+1}|S_{t+1})\, Q_{h-1}(S_{t+1}, A_{t+1}) + \gamma\pi(A_{t+1}|S_{t+1})\, G_{t+1:h} \\ &= R_{t+1} + \gamma\pi(A_{t+1}|S_{t+1})\, [G_{t+1:h} - Q_{h-1}(S_{t+1}, A_{t+1})] + \gamma \bar{V}_{h-1}(S_{t+1}) \end{aligned}$$

- After which it is exactly like the n-step return for Sarsa with control variates except with the action probability $\pi(A_{t+1}|S_{t+1})$ substituted for the importance sampling ratio ρ_{t+1}. For $Q(\sigma)$, we slide linearly between these 2 cases, $\forall\, t < h \leq T$:

$$\boxed{G_{t:h} \doteq R_{t+1} + \gamma \left(\sigma_{t+1}\rho_{t+1} + (1 - \sigma_{t+1})\pi(A_{t+1}|S_{t+1})\right) \left(G_{t+1:h} - Q_{h-1}(S_{t+1}, A_{t+1})\right) + \gamma \bar{V}_{h-1}(S_{t+1})}$$

- The recursion ends with $G_{h:h} = 0$ if $h < T$ or $G_{T-1:T} = R_T$ if $h = T$.
- Now we can plug the new defined return into the n-step Sarsa update w/o importance sampling (handled already in $G_{t:h}$).

7.7. Summary

- There exist a range of TD learning methods that lie in between the one-step TD (TD(0)) methods and Monte Carlo methods. These are n**-step methods.**
- These methods involve an intermediate amount of bootstrapping and are important because they will typically perform better than either extreme.

- n-step methods look ahead to the next n rewards, states, and actions.
- The two **4-step backup diagrams** below together summarize most of the methods introduced:
  - **n-step TD with importance sampling**
  - **n-step** $Q(\sigma)$ which generalizes **Expected Sarsa & Q-learning.**

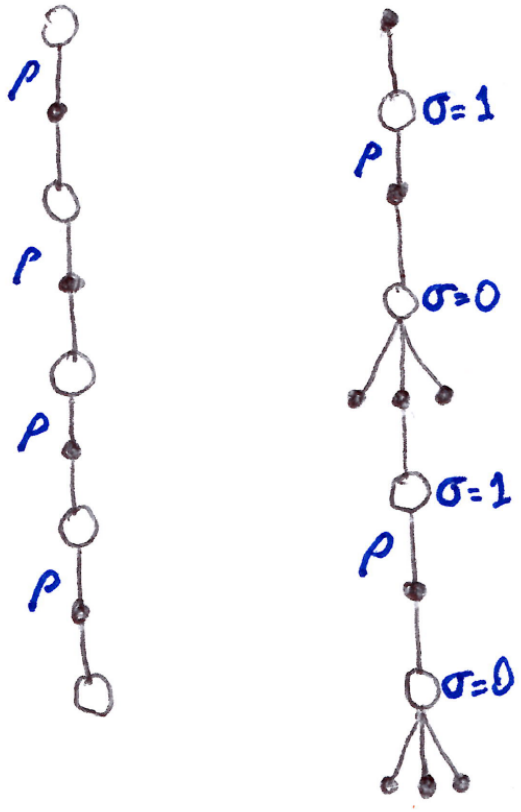

Figure 7.5. Backup diagram for 4-step TD & 4-step Q update.

- **Limitations of n-step methods:**
  - n time step **delay** before updating.
  - **More computation** per time step than previously covered methods.
  - Require **more memory** to record the states, actions, rewards, and sometimes other variables over the last n time steps than the one-step methods.
- In Chapter 12, we'll learn how multi-step TD methods can be implemented with minimal memory and computational complexity using **eligibility traces.**
- n-step methods, while being quite complex, are conceptually clear. Hence the development of two n-step off-policy learning approaches based on:
  - **Importance-sampling**: conceptually simple but has high variance.
  - **Tree-based backup updates**: natural extension of Q-learning to the multi-step case with stochastic target policies.

8. Planning & Learning with Tabular Methods

- **Model-Based RL methods** require a model of the environment and rely on **planning** as their primary component.
  - Dynamic Programming (DP), Heuristic Search
- **Model-Free RL methods** don't require a model of the environment and primarily rely on **learning**.
  - Monte Carlo (MC), Temporal-Difference (TD)
- Both methods still use value functions and both make backups to state values based on future returns.

8.1. Models & Planning

- A **model** is anything the agent can use to predict the environment's behavior.
  - Given a state and an action, a model predicts the next state and the next reward.
  - If the model is stochastic, then there are several possible next states and next rewards. The model produces:
    - All the possibilities and probabilities; these are **distribution models**.
    - One of the possibilities, sampled according to the probabilities; these are **sample models**.
- Models are used to simulate the environment and thus produce **simulated experiences**.
- **Planning** refers to any computational process that takes a model as input and produces or improves a policy for interacting with the modelled environment.

$$\text{model} \xrightarrow{\text{Planning}} \text{policy}$$

- There are 2 distinct approaches to planning:
  - **State-space planning**: search through the state space for an optimal policy. Actions cause transitions from state to state, and value functions are computed over states.
  - **Plan-space planning**: search through the space of plans. Operators transform one plan into another, and value functions, if any, are defined over the space of plans. e.g. Evolutionary methods, partial-order planning.
- **State-space planning common structure:**

$$\text{model} \rightarrow \text{simulated experience} \xrightarrow{\text{backups}} \text{values} \rightarrow \text{policy}$$

- All state-space planning methods involve computing value functions as a key intermediate step toward improving the policy.
- They compute value functions by updates or backup operations applied to simulated experience.
- Dynamic programming methods make sweeps through the space of states, generating for each state the distribution of possible transitions. Each distribution is then used to compute a backed-up value (update target) and update the state's estimated value.
- At the heart of both learning and planning is the estimation of value functions by backing-up update operations.
- The difference however is that:
  - Planning uses **simulated** experience generated by a model.
  - Learning uses **real** experience generated by the environment.
- Despite the differences, a learning algorithm can be substituted for the key update step of a planning method, because it applies just as well to simulated experience.

8.1.1. Random-sample One-step Tabular Q-Learning

- A planning method based on one-step tabular Q-learning and on random samples from a sample model.
- It converges to the optimal policy for the model under the same conditions as that for the real environment.

8.1.2. Pseudocode: Random-sample One-step Tabular Q-Learning

Loop forever:

1. Select a state $S \in \mathcal{S}$ and an action $A \in \mathcal{A}(S)$ at random
2. Send S, A to a sample model and obtain a sample next reward R and a sample next state S'
3. Apply one-step tabular Q-learning to S, A, R, S':

$$Q(S,A) \leftarrow Q(S,A) + \alpha\Big[R + \gamma \max_a Q(S',a) - Q(S,A)\Big]$$

- Planning in very small, incremental steps may be the most **efficient** approach especially in large scale problems.

8.2. Dyna: Integrated Planning, Acting, and Learning

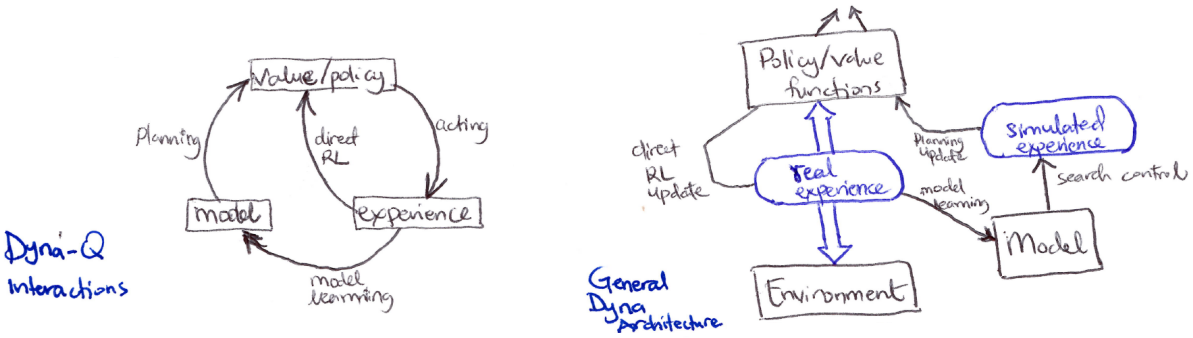

Figure 8.1. Dyna-Q interactions and general Dyna architecture.

- When planning is done online, while interacting with the environment, a number of interesting issues arise:
  - New information gained from the interaction may change the model (and thus the planning).
  - How do we divide the computational resources available between decision making and model learning?
- **Dyna-Q**: a simple architecture integrating the major functions needed in an online planning agent.
- Within a planning agent, there are at least 2 roles for real experience:
  - **Model learning**: to improve the model (to make it more accurately match the real environment).
  - **Direct RL**: to directly improve the value function and policy.
  - **Indirect RL**: to indirectly improve the value functions and policies via the model (planning).
- Both direct and indirect methods have advantages and disadvantages:
  - Indirect methods often make fuller use of a limited amount of experience.
  - Direct methods are much simpler and are not affected by biases in the design of the model.
- **Dyna-Q includes all of the RL processes in the interactions diagram shown above occurring continuously: planning, acting, model-learning, and direct RL.**
- The planning method is the random-sample, one-step tabular Q-planning.
- The model-learning method is also table-based and assumes the environment is deterministic.
  - After each transition $S_t, A_t \rightarrow R_{t+1}, S_{t+1}$, the model records in its table entry for S_t, A_t the prediction that R_{t+1}, S_{t+1} will deterministically follow.
  - Thus, if the model is queried with a state-action pair that has been experienced before, it simply returns the last S_{t+1}, R_{t+1} experienced as its prediction.
  - During planning, the Q-learning algorithm randomly samples only from state-action pairs that have previously been experienced.
- Based on the overall architecture of Dyna agents:
  - **Search control**: the process that selects the starting states & actions for the simulated experiences.
  - Planning is achieved by applying RL methods to the simulated experiences just as if they had really happened.

- Typically, as in Dyna-Q, the same RL method is used both for learning from real experience and for planning from simulated experience.
- Learning and planning share almost all the **same** machinery, differing only in the source of their experience.
- Conceptually, planning, acting, direct RL, and model learning occur simultaneously and in parallel in Dyna agents. For computational concreteness and implementation, we specify the order of occurrence within a time step:
  - Acting, model-learning and direct RL processes require little time.
  - Planning takes the remaining time in each step because it is inherently computationally intensive.

8.2.1. Pseudocode: Tabular Dyna-Q

Initialize $Q(s,a)$ and Model(s,a) for all $s \in \mathcal{S},\ a \in \mathcal{A}(s)$

Loop forever:

(a) $S \leftarrow$ current (non-terminal) state

(b) $A \leftarrow \varepsilon$-greedy(S,Q)

(c) Take action A; observe resultant reward R and state S'

(d) $Q(S,A) \leftarrow Q(S,A) + \alpha\Big[R + \gamma \max_a Q(S',a) - Q(S,A)\Big]$

(e) Model$(S,A) \leftarrow R, S'$ (assuming deterministic environment)

(f) **Loop repeat** n **times:**

$S \leftarrow$ random previously observed state

$A \leftarrow$ random action previously taken in S

$R, S' \leftarrow$ Model(S,A)

$Q(S,A) \leftarrow Q(S,A) + \alpha\Big[R + \gamma \max_a Q(S',a) - Q(S,A)\Big]$

- In the pseudocode algorithm for Dyna-Q above:
  - Model(S,a) denotes the contents of the model (predicted S_{t+1} & R_{t+1}) for state-action pair.
  - Direct RL, model-learning and planning are steps **(d), (e)** and **(f)** respectively.
  - If **(e)** and **(f)** were omitted, the algorithm becomes one-step tabular Q-learning.
- A simple maze example shows that adding planning ($n > 0$) dramatically improves the agent's behavior.
- Due to the incremental nature of planning, it is trivial to intermix planning & learning. Both proceed quickly.
  - The agent is always reactive and deliberative, and yet always planning and model-learning in the background.

8.3. When the Model is Wrong

- Models may be incorrect because:
  - The environment is **stochastic** and only a limited number of samples have been observed.
  - The model was learned using function approximation that has **generalized** imperfectly.
  - The environment has **changed** and its new behavior has not yet been observed.
- In some scenarios, the suboptimal policy computed by planning quickly leads to the discovery and correction of the modeling error.
  - This tends to happen when the model aims to predict a greater reward than is possible.
  - The planned policy attempts to exploit these opportunities and discovers that they do not exist.
- Issues arise when the environment changes to become better than it was before, and yet the formerly correct policy does not reveal the improvement.
  - In these cases the current optimal policy remains unchanged, but there lies 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 **exploration/exploitation tradeoff/conflict**, there is no perfect and practical solution, but there are simple heuristics that can be effective.
  - One proposed algorithm that used one such heuristic was the **Dyna-Q+** agent.

8.3.1. Dyna-Q+

- 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 has **changed** and the model of it is incorrect.
- A special bonus reward is added to simulated experience involving these actions to encourage **exploratory behavior** towards long-untried actions ($k\sqrt{\tau}$).
- The modelled rewards for each state-action pair now becomes $r + k\sqrt{\tau}$ for some small k.
  - r is the initial unchanged model reward without any heuristic applied.
- This encourages the agent to be more exploratory to all accessible state transitions, and even though this may be costly computationally, it's **well worth it.**

8.4. Prioritized Sweeping

- In Dyna agents, simulated transitions are started in state-action pairs selected uniformly at random from all previously experienced state-action pairs.
  - Uniform selection is usually not the best; focusing on particular state-action pairs could be much more efficient for planning.
- **Prioritized sweeping** optimizes Dyna-style planning by selectively updating state-action pairs based on expected magnitude of value change, rather than uniform random selection.
  - Steps:
    1. Keep a priority queue of which state-action pairs need updating most.
    2. Update the ones with biggest potential changes first.
    3. Work backwards from important states (like the goal).
  - Mechanism:
    1. Maintain priority queue of (s, a) pairs ranked by Bellman error magnitude.
    2. Propagate updates **backward** from states with changed values.
    3. Queue predecessors weighted by: $|R + \gamma V(s') - Q(s, a)|$
  - Key advantages:
    1. **Efficiency**: avoid wasteful updates (such as $0 \to 0$ reward transitions).
    2. **Convergence speed**: dramatic empirical improvements.
    3. **Backward focusing**: value propagation follows reverse trajectory from changed states.
- Extensions of prioritized sweeping to **stochastic environments** are straightforward:
  - **Expected updates**: enumerate all s' with transition probabilities, which is comprehensive but computationally expensive on low-probability transitions.
  - **Sample updates**: lower variance per computation, better granularity enables selective focus on high-impact transitions.
  - Essentially, when outcomes are random, one can either update based on all possibilities (slow but thorough) or sample specific outcomes (faster, focuses effort).
- All planning entails sequences of value updates varying in:
  - Update type $\Rightarrow$ expected/sample, full/partial backup.
  - Update ordering $\Rightarrow$ backward/forward focusing, prioritization heuristic.
- Forward focusing prioritizes states by reachability under current policy rather than backward value propagation.

8.5. Expected vs Sample Updates

- So far in the book we've discussed dynamic programming (DP) as a way of conducting policy evaluation and policy improvement given a distribution model of the environment.
- We've also discussed sampling methods like Monte Carlo (MC), temporal-difference (TD), and n-step bootstrapping to estimate value functions in the absence of a model.
- Given a fixed computational budget, are expected or sample updates more efficient for planning?

Backup diagrams for one-step updates (7 of 8 cases shown):

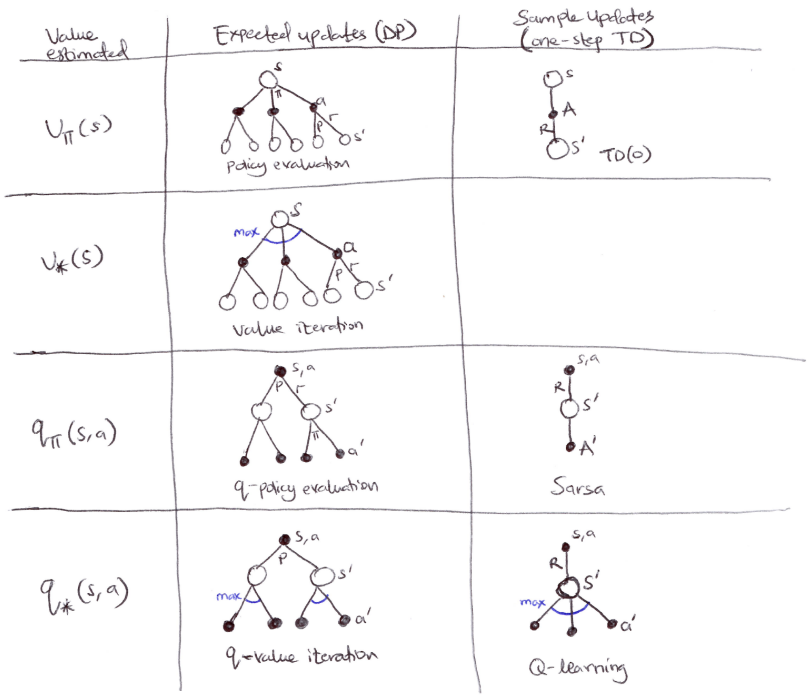

Figure 8.2. Backup diagrams for all the one-step updates covered so far.

Value estimated	Expected updates (DP)	Sample updates (one-step TD)
$v_\pi(s)$	Policy evaluation (full branching over actions & next states)	TD(0) (single sampled transition)
$v_*(s)$	Value iteration (max over actions, full branching)	-
$q_\pi(s,a)$	q-policy evaluation	Sarsa
$q_*(s,a)$	q-value iteration	Q-learning

- We have considered many value function updates. If we focus on one-step updates, they vary along 3 binary dimensions:
  - Whether they update **state values** or **action values.**
  - Whether they estimate the value for the **optimal policy** or for an **arbitrary given policy.**
  - Whether the updates are **expected updates** (consider all possible events that might happen) or **sample updates** (consider a single sample of what might happen).
- These 3 binary dimensions give rise to 8 cases, 7 of which are shown in the figure above. The 8th case does not seem to correspond to any useful update.
- Any of these one-step updates can be used in planning methods:
  - **Dyna-Q** uses q_* sample updates, but could also use q_* expected updates, or either expected or sample q_π updates.
  - **Dyna-AC** uses V_π sample updates together with a learning policy structure.
  - For stochastic problems, prioritized sweeping is always done using one of the expected updates.
- Absence of a distribution model means that expectation is impossible, but sampling can be done.
  - So which is better? Expectation or Sampling?
    - Expected updates yield a better estimate because they are uncorrupted by sampling error.
    - Expected updates require more computation, and computation is often the limiting resource in planning.

8.5.1. Computational Requirements & Formal Comparison (given discrete states/actions)

- Model: $\hat{p}(s', r|s, a)$ known.
- Goal: Approximate q_* (optimal action values).
- Branching factor: $b = |\{s' : p(s'|s, a) > 0\}|$ (effective stochasticity).
- **Expected Update (exact):**
  - Computational complexity: $O(b)$.

$$\boxed{Q(s, a) \leftarrow \sum_{s', r} \hat{p}(s', r|s, a) \left[r + \gamma \max_{a'} Q(s', a') \right]}$$

- **Sample Update (stochastic):**
  - Computational complexity: $O(1)$.

$$\boxed{Q(s, a) \leftarrow Q(s, a) + \alpha \left[R + \gamma \max_{a'} Q(S', a') - Q(s, a) \right]}$$

- In the same fixed time budget for 1 expected update, you get b sample updates.

8.5.2. Theoretical Comparison (Empirical Analysis)

- Assume all b successors are equiprobable, and initial $|\text{error}| = 1$ at (s, a); successor values are assumed already correct.
- Expected update: error $= 0$ after one update (cost: $\sim b$ units).
- Sample updates (assuming sample averages, i.e. $\alpha = \frac{1}{t}$): error $\approx \sqrt{\frac{b-1}{bt}}$.
  - For moderate b (e.g. $b = 10$) and large b, the error falls dramatically with a tiny fraction of b updates.
  - For large b, error drops exponentially fast early for sample updates, allowing broad updates across many (s, a) pairs in the same time as one expected update:

$$\text{error} = \sqrt{\frac{b-1}{bt}}$$

  - For large b: error $\approx \sqrt{\frac{1}{t}}$, $\therefore \lim_{t\to\infty} \sqrt{\frac{1}{t}} \to 0$.

$$b = 1: \quad \text{error} = 0 \quad \text{for } t \geq 1$$

$$b = 2: \quad \text{error} = \frac{1}{\sqrt{2t}} \implies \text{error}(t = 1) \approx 0.707,\ \text{error}(t = 2) = 0.5$$

$$b = 100: \quad \text{error} \approx \frac{1}{\sqrt{t}} \implies \text{error}(t = 1) \approx 0.995,\ \text{error}(t = 10) \approx 0.316,\ \text{error}(t = 100) \approx 0.1$$

- Pros of sample updates:
  - **Breadth vs depth**: cover more state space per unit computation.
  - **Bootstrap benefits**: earlier updates improve successor value estimates for subsequent backups.
  - **Diminishing returns**: marginal value of incorporating low-probability branches is low.
- Sample updates dominate in large-scale stochastic domains where exhaustive sweeping is intractable.
- Expected updates only preferable when:
  - Small branching factor ($b \leq 3$).
  - Small state space (exact solution feasible).
  - Deterministic dynamics ($b = 1$, methods are equivalent).

8.6. Trajectory Sampling

- Let's compare two ways of distributing updates:

  - **Exhaustive Sweeps**: classical DP approach that entails performing sweeps through the entire state (or state-action) space, updating each state (or state-action pair) once per sweep. Computationally inefficient especially on large tasks (no time to complete one full sweep).
  - **Trajectory Sampling**: generate simulated trajectories by rolling out the current policy in the model, performing one-step backups along the trajectory.
- 2 common sampling distributions:
  - Uniform sampling of states or state-action pairs.
  - On-policy distribution.
- For planning updates in tabular RL, should state-action pairs be selected uniformly or according to the on-policy distribution?

8.6.1. Formal Comparison

- **Uniform distribution:**
  - Cycle systematically through all $|S| \times |A|$ state-action pairs.
  - Each pair receives equal computational resources.
  - Complete coverage regardless of policy.
  - Starting state distribution $\approx$ uniform or some fixed distribution $\mu(S_t)$.
- **On-policy trajectory sampling:**
  - Sample states $S_t \sim d^\pi$ where d^π is the on-policy state distribution under policy π.
  - Select actions $a_t \sim \pi(\cdot|S_t)$.
  - Generate trajectories $\{S_0, a_0, S_1, a_1, ...\}$ following current policy.
  - Update only visited state-action pairs.
- **Advantages of trajectory sampling:**
  - **Computational focusing**: for large state spaces where $|S| \gg$ states reachable under π, trajectory sampling concentrates updates on the reachable subset.
  - **Irrelevant state pruning**: 3 categories emerge:
    * Initial states (starting distribution).
    * States reachable under optimal control.
    * Irrelevant states (never visited optimally).
- **Disadvantages/Limitations of trajectory sampling:**
  - Requires a generative model to simulate trajectories.
  - May miss important states early in learning if policy is poor.
  - Can be sample-inefficient if trajectories are long.
- On-policy distribution sampling is useful/better for problems with large state-spaces and small branching factors (prioritized sweeping).
- Trajectory sampling is orthogonal to prioritization. The former addresses which states to update, while the latter addresses in what order. They can be combined.
- Trajectory sampling anticipates importance sampling concepts in off-policy learning and naturally extends to continuous state spaces with function approximation.

8.7. Real-Time Dynamic Programming (RTDP)

- RTDP is an on-policy trajectory-sampling version of the value-iteration algorithm of dynamic programming (DP).
- RTDP is an asynchronous DP algorithm;
  - async DP algorithms are not organized in terms of systematic sweeps of the state set, instead
  - they update state values in any order whatsoever, using whatever values of other states happen to be available.
- RTDP is basically the combination of 3 ideas:
  - **On-policy trajectory sampling** $\Rightarrow$ follow the current greedy policy.
  - **Asynchronous updates** $\Rightarrow$ only update states you actually visit in any order.
  - **Focused learning** $\Rightarrow$ concentrate on **"relevant states"** (states on good paths to the goal).
- **RTDP update rule:**

$$\boxed{V(S_t) \leftarrow \max_{a \in A} \left(R^a_{S_t} + \gamma \sum_{s'} P^a_{S_t s'} V(s') \right)}$$

- RTDP's relationship to other methods:
  - **Value Iteration**:
    - VI updates all states per iteration;
    - RTDP updates only states on sampled trajectories.
  - **Prioritized Sweeping**:
    - PS uses model to work backward from goals;
    - RTDP follows forward trajectories.
  - **Trajectory Sampling**:
    - TS can use any policy;
    - RTDP uses greedy policy for sampling.
- **Computational Complexity:**
  - Traditional Value Iteration $\Rightarrow O(|S|^2|A|)$ per iteration.
  - RTDP Trial $\Rightarrow O(L)$ where L = episode length, typically $L \ll |S|$.
- RTDP bridges pure planning and pure learning (focusing on relevant state space regions).
- RTDP is guaranteed to find an optimal policy for the relevant states under certain conditions:
  1. The initial value of every goal state is zero.
  2. There exists at least one policy that guarantees that a goal state will be reached with probability one from any start state.
  3. All rewards for transitions from non-goal states are strictly negative.
  4. All the initial values are equal to or greater than their optimal values (which can be satisfied by simply setting the initial values of all states to zero).
- 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 by traditional sweep-based value iteration (i.e. dynamic programming).
  - These kinds of problems are usually expressed in cost minimization not reward maximization.

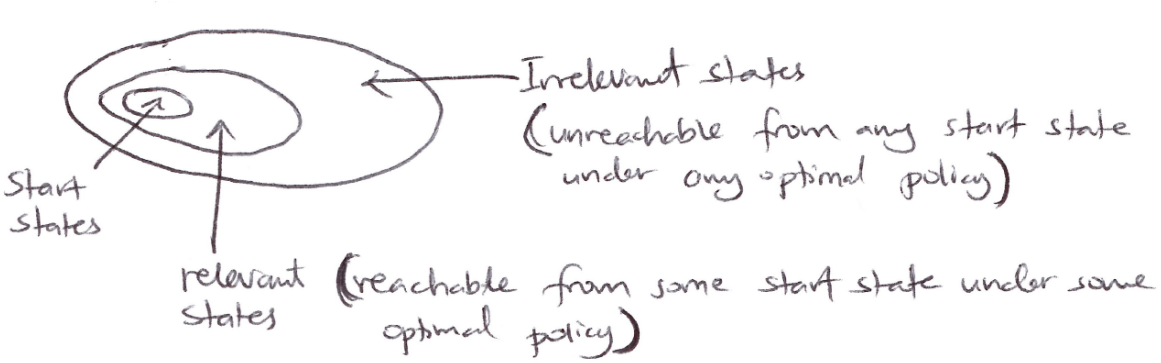

Figure 8.3. **State space diagram.** Start states on the left, irrelevant states (unreachable from any start state under any optimal policy) in the outer region, and relevant states (reachable from some start state under some optimal policy) in the inner region.

8.7.1. Heuristic Initialization (Optimistic)

- RTDP typically initializes V with an admissible heuristic $h(s)$ where $h(s) \geq V^*(s)$.
- This provides optimistic values that guide exploration towards goal states.

8.7.2. Considerations

- Most effective domains for RTDP are domains with:
  - Large state spaces.
  - Sparse goal states.
  - Clearly defined initial distribution.
  - Deterministic or near-deterministic dynamics.

8.8. Planning at Decision Time

- Planning can be used in at least 2 ways:
  - **Background planning**: planning that occurs independently of and prior to the need for action. Here planning is used to gradually improve a policy/value function on the basis of simulated experience from a model, then selects an action via lookup.
  - **Decision-time planning**: planning that occurs at the moment an action is required after encountering the current state. It's essentially planning at the time of action selection.
- Decision-time planning is useful when fast response is not required and the state space is large. In low-latency actions, background planning is better for action selection.
- Decision-time planning is **memoryless** (discards updates after selection of action), but background planning is **persistent** (permanently stores and accumulates learned values).

8.9. Heuristic Search

- The classical state-space planning methods are decision-time planning methods collectively known as **heuristic search**.
- In heuristic search, for each state encountered, a large tree of possible continuations is considered.
  - 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.
- Heuristic search is an extension of greedy policy beyond one-step to multi-step lookahead to obtain better action selections.

8.9.1. Theoretical Properties

1. **Optimality horizon**: for sufficiently large depth K where $\gamma^K \approx 0$, the selected action approaches the optimal action $a^*(s)$.
2. **Computational complexity**:
   - Full tree expansion: $O(b^K)$ where $b =$ branching factor.
   - With pruning/selection: $O(f(b, K))$ where $f < b^K$.
3. **Memory**: $O(bK)$ with depth-first implementation.
4. **Backed-up value interpretation**: $v_{\text{tree}}(s)$ estimates the K-step optimal value starting from s.

8.10. Rollout Algorithms

- Rollout algorithms are decision-time planning algorithms based on Monte Carlo control applied to simulated trajectories that all begin at the current environment state.
- They estimate action values for a given policy by averaging the returns of many simulated trajectories that start with each possible action and then follow the given policy.
- **Key characteristics:**
  - It is memoryless; it doesn't store/update a permanent value table (no persistence).

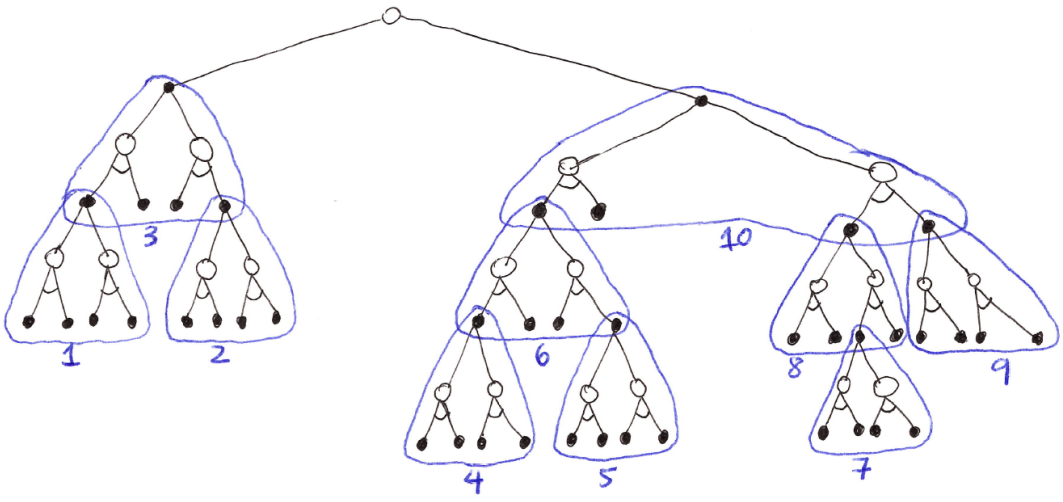

Figure 8.4. **Heuristic Search diagram (selective depth-first search).** a large tree rooted at the current state, with branches for each action and subtrees for each successor. The tree policy traverses the tree greedily, evaluating and backing up values from the leaf nodes toward the root.

  - The rollout policy Π is usually the current greedy policy.
  - Relies on MC averaging to reduce variance (more rollouts $\rightarrow$ better estimates).
  - It's quite very simple; no tree-building, no backups during rollouts.
- **Goal:**
  - Not to estimate a complete q_* or q_π for a given policy π; instead the focus is on MC estimates of action values only for each current state and for a given fixed policy called the **rollout policy**.
  - Improve upon rollout policy, not find optimal policy.
- Rollout algorithms follow the **policy improvement theorem** by acting greedily w.r.t. $\hat{Q}(s,a)$:
  - If $q_\pi(s,a) \geq v_\pi(s)$ for any 2 policies π and π', then $\pi' \geq \pi$.

8.10.1. Computational Complexity (quite expensive due to many full episodes)

- Per decision $\Rightarrow$:
  - $|A(s)|$ = number of actions to evaluate,
  - n = rollouts per action,
  - L = average episode length.

$$\text{total cost} = O\left(|A(s)| \cdot n \cdot L\right)$$

- As we can see the computational time depends on many factors and balancing these factors is very important and challenging. To handle the challenge:
  - It is possible to run many trials in parallel on separate processors (because the MC trials are independent of one another).
  - Truncate the simulated trajectories short of complete episodes, correcting the truncated returns by means of a stored evaluation function.
  - Pruning away candidate actions that are unlikely to be the best.
- Rollout algorithms aren't learning algorithms because of no long-term memory of values/policies.
  - But they still use RL techniques: **Monte Carlo control + Policy Improvement Theorem.**
  - Use MC control to estimate action values via averaging the returns of a collection of sample trajectories.
  - Take advantage of the policy improvement property by acting greedily w.r.t. the estimated action values.

8.11. Monte Carlo Tree Search (MCTS)

- MCTS is a rollout algorithm that is enhanced by the addition of a means for accumulating value estimates obtained from the MC simulations in order to successively direct simulations toward more highly-rewarding trajectories.
- It is a best-first search algorithm that builds a decision tree incrementally by iteratively sampling trajectories through an MDP, using statistical confidence bounds for exploration-exploitation balance.
- 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 4 operations:

8.11.1. 1. Selection

- Starting at the root node, a tree policy based on the action values attached to the edges of the tree traverses the tree to select a leaf node.
- Traverse tree using tree policy (typically Upper Confidence bounds for Trees, **UCT**):

$$\boxed{\mathrm{UCT}(s,a) = \underbrace{\frac{W(s,a)}{N(s,a)}}_{\text{exploitation}} + c\underbrace{\sqrt{\frac{\ln(N(s))}{N(s,a)}}}_{\text{exploration}} = Q(s,a) + c\sqrt{\frac{\ln(N(s))}{N(s,a)}}}$$

- where:
  - $Q(s,a)$ = average return from $(s,a) = \frac{W(s,a)}{N(s,a)}$
  - $W(s,a)$ = total reward accumulated through (s,a)
  - $N(s,a)$ = visit count for (s,a)
  - $N(s)$ = visit count for parent state s
  - c = exploration constant (typically $\sqrt{2}$);
    - $c < 0.5 \Rightarrow$ more exploitation
    - $c > 2 \Rightarrow$ more exploration
- Selection terminates when:
  - Leaf node is reached (not fully expanded).
  - Terminal state is reached.

8.11.2. 2. Expansion

- On some iterations, the tree is expanded from the selected leaf node by adding one or more child nodes reached from the selected node via unexplored actions.
- Expansion strategies:
  - Single child per iteration (standard).
  - All children at once (batch expansion).
  - Progressive widening for continuous actions.

8.11.3. 3. Simulation

- From the selected node, or from one of its newly-added child nodes (if any), simulation of a complete episode is run with actions selected by the rollout policy (actions are selected first by the tree policy and beyond the tree by the rollout policy).

8.11.4. 4. Backup

- The return generated by the simulated episode is backed up to update, or to initialize, the action values attached to the edges of the tree traversed by the tree policy in this iteration of MCTS.
- Propagate simulation result up the path and update all nodes/edges on the path.

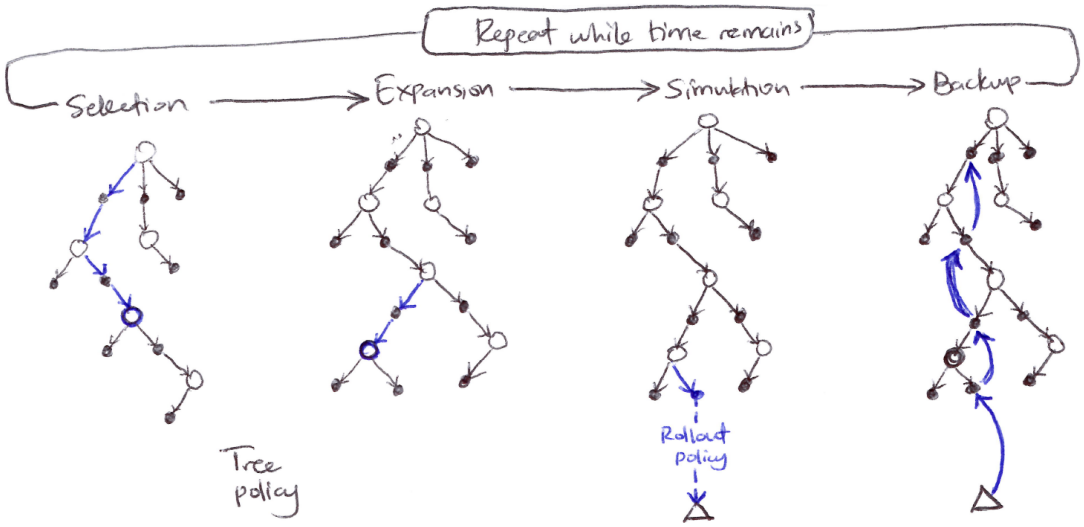

Figure 8.5. **MCTS diagram.** 4 stages shown left to right: Selection (tree policy traverses with blue arrows to a leaf), Expansion (leaf expanded), Simulation (rollout policy runs from expanded node to terminal Δ), Backup (return propagated back up with blue arrows).

- MCTS continues executing these 4 steps, starting each time at the tree's root node, until no more time is left, or some other computational resource is exhausted.
- Then finally, an action from the root node (representative of the environment's current state) is selected according to some mechanism that depends on the accumulated statistics in the tree (action with largest action value or action with largest visit count to avoid outliers).

8.11.5. Pseudocode: MCTS

Initialize: root = current state S_0
for $i = 1$ **to** num_simulations:

▷ **Selection**
node ← root
While node is fully expanded and non-terminal:

$$\text{node} \leftarrow \arg\max_a \left[\frac{W(\text{node}, a)}{N(\text{node}, a)} + c\sqrt{\frac{\ln N(\text{node})}{N(\text{node}, a)}} \right]$$

▷ **Expansion**
If node is non-terminal and not fully expanded:
Select unvisited action $a' \in \mathcal{A}(S) \setminus \{\text{visited actions}\}$
Create child node $S' \sim p(\cdot|S, a')$ and add to tree
node ← S'

▷ **Simulation**
$S \leftarrow$ node; $G \leftarrow 0$; $t \leftarrow 0$
While S is non-terminal:
$a \sim \Pi_{\text{default}}(\cdot|S)$
$r, S' \sim p(\cdot|S, a)$
$G \leftarrow G + \gamma^t r$; $S \leftarrow S'$; $t \leftarrow t + 1$

▷ **Backup**
While node $\neq$ null:
$N(\text{node}) \leftarrow N(\text{node}) + 1$
$W(\text{node}) \leftarrow W(\text{node}) + G$
node ← node.parent

return $\arg\max_a N(S_0, a)$

8.11.6. Computational Complexity

- Per iteration (where d = tree depth, L = episode length):
  - Selection: $O(d)$,
  - Expansion: $O(1)$,
  - Simulation: $O(L)$,
  - Backup: $O(d)$

$$\Rightarrow \text{total: } O(n \cdot (d + L)) \text{ for } n \text{ simulations}$$

8.11.7. Summary of MCTS

- MCTS is a decision-time planning algorithm based on Monte Carlo control applied to simulations that start from the root state.
- MCTS benefits from online, incremental, sample-based value estimation & policy improvement.
- MCTS saves action-value estimates attached to the tree edges and updates them using RL's sample updates.
- MCTS, via incremental tree expansion, effectively grows a lookup table to store a partial action-value function.
- MCTS thus avoids the problem of globally approximating an action-value function while it retains the benefit of using past experience to guide exploration.

8.11.8. Pros & Cons of MCTS

- **Pros**:
  - anytime algorithm,
  - asymmetric tree growth,
  - no domain heuristic required,
  - handles high branching factors.
- **Cons**:
  - high computational cost,
  - may miss deep forced sequences,
  - random rollouts that are weak in tactical domains,
  - finite simulations miss long-term consequences.

8.12. Summary

- Planning requires a model of the environment.
  - A **distribution model** consists of the probabilities of next states and rewards for possible actions. Dynamic Programming requires a distribution model because it uses expected updates, which involve computing expectations over all the possible next states and rewards.
  - A **sample model** is required for simulating the environment interaction using sample updates.
  - Sample models are generally much easier to obtain than distribution models.
- There exists a close relationship between planning optimal behavior and learning optimal behavior:
  - Both involve estimating the same value functions.
  - Both naturally update the estimates incrementally, in a long series of small backup operations.
  - Any of the learning methods can be converted into planning methods simply by applying them to simulated rather than real experience (model-based not model-free).
- It is straightforward to integrate incremental planning methods with acting and model-learning. Planning, acting and model-learning interact in a circular fashion, each producing what the other needs to improve. All processes naturally proceed asynchronously and in parallel.
- **Dimensions of variation among state-space planning methods:**
  - **Size of updates**: the smaller the updates, the more incremental the planning methods can be. One-step updates, as in Dyna, are the smallest updates.
  - **Distribution of updates**: primarily regarded as the focus of search.
    - **Prioritized sweeping** focuses backward on the predecessors of recently changed states.
    - **On-policy trajectory sampling** focuses on states/state-action pairs that are likely.
- **Real-time DP (RTDP),** an on-policy trajectory sampling version of value iteration, illustrates some of the advantages that focusing on the relevant regions of the state-space has over conventional sweep-based policy iteration (exhaustive).
- Planning can also focus forward from pertinent states, such as states actually encountered during an agent-environment interaction, and the most important form of this is when **planning is done at decision time** as part of the action-selection process.
  - Another example of this is **classical heuristic search.**
  - Other examples are **rollout algorithms & Monte Carlo Tree Search (MCTS)** that both benefit from online, incremental, sample-based value estimation and policy improvement.

8.13. Summary of Part I: Dimensions

Two axes for the update diagram:

$$\text{HORIZONTAL (L to R): sample backups} \xrightarrow{\text{width of update}} \text{full/expected backups}$$

$$\text{VERTICAL (Top to Bottom): shallow backups} \xrightarrow{\text{depth/length of update}} \text{deep backups}$$

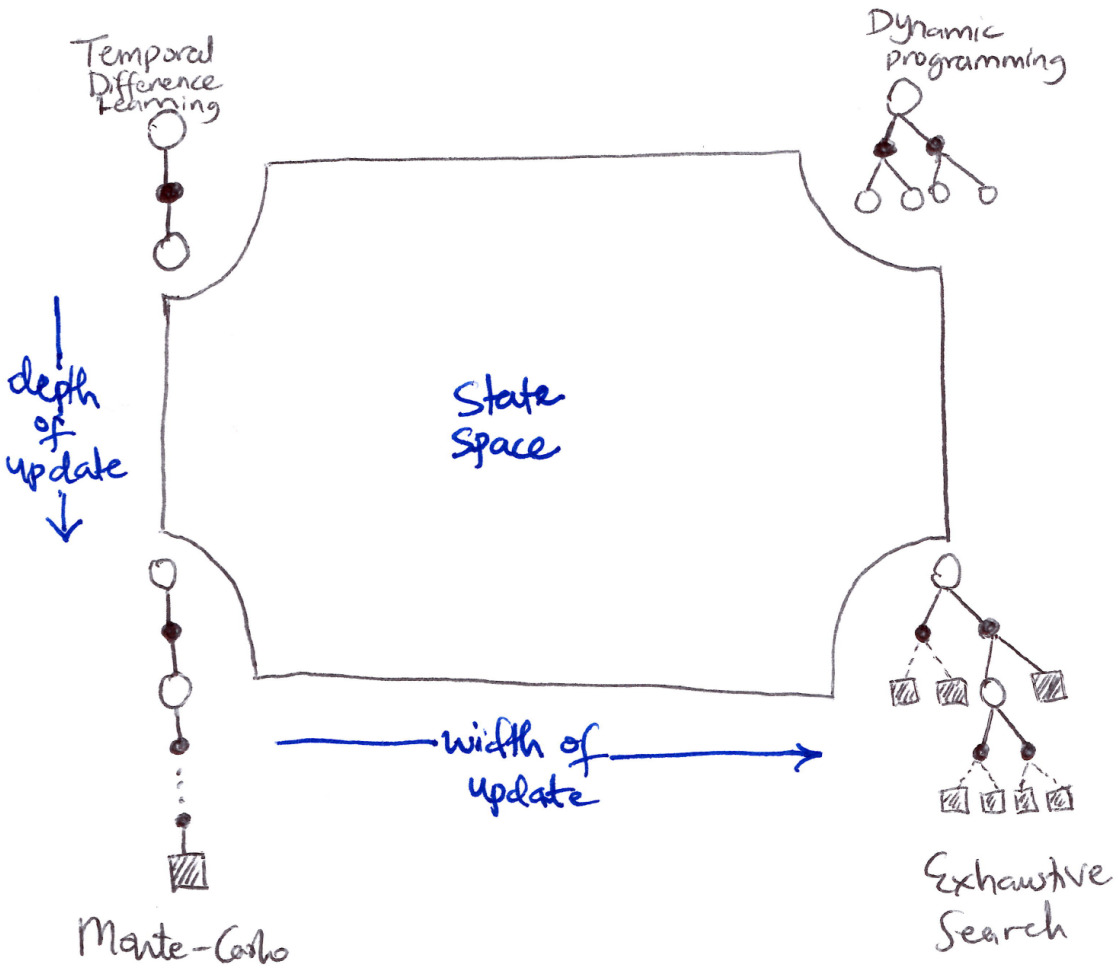

Figure 8.6. Unified view of RL depicting a slice through the space of RL methods.

- Each RL idea presented so far can be viewed as a dimension along which methods vary. The set of the dimensions spans a large space of possible methods (quasi-infinite possibilities).
- All the methods discussed so far in this book have 3 key ideas in common:
  - They all seek to estimate value functions.
  - They all operate by backing up values along actual or possible state trajectories.
  - They all follow the general strategy of **generalized policy iteration (GPI).** This means they maintain an approximate value function and an approximate policy, and they continually try to improve each on the basis of the other.
- 3 important **dimensions** along which RL methods vary:
  - **Width of updates**: sample updates (based on a sample trajectory) vs expected updates (based on a distribution of possible trajectories).
  - **Depth of updates**: degree of bootstrapping (λ).
  - **On-policy vs off-policy methods**.
- **Other dimensions along which RL methods vary:**
  - **Definition of return**: is the task episodic or continuing, discounted or undiscounted?
  - **Action values vs state values vs afterstate values**.
  - **Action selection/exploration**: how are actions selected to ensure a suitable exploration/exploitation trade-off? Simple ways considered are ε-greedy, optimistic initialization, soft-max, upper confidence bound (UCB).
  - **Synchronous vs asynchronous**: are the updates for all states performed simultaneously or one-by-one in some order?
  - **Real vs simulated experience**.
  - **Location of updates**: what states or state-action pairs should be updated? Model-free methods can choose only among encountered states, but model-based methods can choose arbitrarily.
  - **Timing of updates**: should updates be done as part of action selection, or only afterward?
  - **Memory for updates**: how long should updated values be retained?
    - Should they be retained permanently? (**persistence**)
    - Or only while computing an action selection and then discarded? (**memoryless**)
- These dimensions are **neither exhaustive nor mutually exclusive.** e.g. Dyna methods use both real and simulated experience to affect the same value function.
- These dimensions constitute a coherent set of ideas for description & exploration of a wide space of possible methods.
- The most important dimension not mentioned or covered yet is that of **function approximation**:
  - Function approximation can be viewed as an orthogonal spectrum of possibilities ranging from **tabular methods** at one extreme through **state aggregation**, a variety of **linear methods**, and then a diverse set of **non-linear methods**.

Part II.

Approximate Solution Methods

9. On-Policy Prediction with Approximation

- Study of **function approximation** in RL by considering its use in estimating the state-value function from on-policy data, i.e. in approximating v_π from experience generated using a known policy π.
- The approximate value function is represented as a parameterized functional form with **weight vector** $\mathbf{w} \in \mathbb{R}^d$:

$$\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$$

- The function above is for the approximate value of state s given weight vector $\mathbf{w}$.
- $\hat{v}$ might be a linear function, a multi-layer artificial neural network, or a decision tree.
- Extending RL to function approximation makes it applicable to **partially observable problems**, in which the full state is unavailable to the agent.
- Function approximation cannot augment the state representation with memories of past observations.

9.1. Value-Function Approximation

- All the prediction methods covered so far involve an update to an estimated value function that shifts its value towards a "backed-up value" (update target, u). Let's denote an individual state update by $s \mapsto u$:

$$\begin{aligned}
\text{Monte-Carlo (MC):} &\quad S_t \mapsto G_t \\
\text{TD(0):} &\quad S_t \mapsto R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t) \\
n\text{-step TD:} &\quad S_t \mapsto G_{t:t+n} \\
\text{Dynamic Programming (DP):} &\quad s \mapsto \mathbb{E}_\pi[R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t) \,|\, S_t = s]
\end{aligned}$$

- Each update is interpretable as an example of the desired input-output behaviour of the value function.
  - Until now, the value function updates have been made in the tabular setting, which is quite trivial because estimated values of all other states are left unchanged.
  - Now using arbitrarily complex and sophisticated methods to perform the update, updating at s generalizes so that the estimated value of many other states are changed as well.
  - Machine learning methods that learn to mimic input-output examples in this way are called **supervised learning** methods, and when the outputs are numbers, like u, the process is called **function approximation**.
- Function approximation methods expect to receive examples of the desired input-output behavior of the function they are trying to approximate.
  - We view each update as a conventional training example.
  - This allows us to use any of a wide range of existing function approximation methods for value prediction.
  - Whatever function approximation method is chosen needs to be able to perform **online learning**.

9.2. The Prediction Objective ($\overline{\text{VE}}$)

- We have more states s than weights $\mathbf{w}$, therefore we cannot feasibly approximate the value function perfectly.
- Making one state's estimate more accurate invariably leads to making the others' less accurate.
- Therefore, it is necessary to define which states we care most about, based on a **state distribution,** $\mu(s)$, that represents how much we care about the error in each state s.

$$\mu(s) \geq 0, \quad \sum_s \mu(s) = 1$$

- Weighting the error in a state s, the difference between the approximate value $\hat{v}(s, \mathbf{w})$ and the true value $v_\pi(s)$, over the state space by μ leads to obtaining a natural objective function called the **mean squared value error**, denoted by $\overline{\text{VE}}$:

$$\boxed{\overline{\text{VE}}(\mathbf{w}) \doteq \sum_{s \in S} \mu(s) \left[v_\pi(s) - \hat{v}(s, \mathbf{w})\right]^2}$$

- $\sqrt{\overline{\text{VE}}}$ tells us roughly how much the approximate values differ from the true values.
- Often μ is chosen to be the fraction of time spent in s, called the **on-policy distribution** under on-policy training.

9.2.1. On-policy Distribution

1. **Continuing tasks**: the on-policy distribution is the stationary distribution under π:

$$\mu(s) = \sum_{s'} \mu(s') \sum_a \pi(a|s')\, p(s|s', a), \quad \forall s \in S$$

where
$\mu(s)$ = stationary probability of being in state s under π
s' = preceding state

> **Balance Equation**
>
> The probability of being in state s equals the sum over all ways of arriving in s from any previous state s' under policy π.

2. **Episodic tasks**: it depends on how the initial states of episodes are chosen:

$$\eta(s) = h(s) + \sum_{\bar{s}} \eta(\bar{s}) \sum_a \pi(a|\bar{s})\, p(s|\bar{s}, a), \quad \forall s \in S$$

where
$h(s)$ = probability that an episode begins in each state s
$\eta(s)$ = number of time steps spent, on average, in state s in a single episode
$\bar{s}$ = preceding state

> **Visitation Equation**
>
> The expected number of visits to state s equals the probability of starting in state s plus the expected number of visits to all preceding states s' that transition into s under policy π.

- This system of equations can be solved for the expected number of visits $\eta(s)$.
- The on-policy distribution is the fraction of time spent in each state, normalized to sum to 1:

$$\mu(s) = \frac{\eta(s)}{\sum_{s'} \eta(s')}, \quad \forall s \in S$$

- If discounting exists, then we redefine $\eta(s)$:

$$\eta(s) = h(s) + \gamma \sum_{\bar{s}} \eta(\bar{s}) \sum_a \pi(a|\bar{s})\, p(s|\bar{s}, a), \quad \forall s \in S$$

whereγ = discount factor

9.2.2. Performance Objective

- Although $\overline{\text{VE}}$ is a good starting point, it's not completely clear that it is the right performance objective.
- The **ultimate goal** (reason for learning a value function) is to find a better policy.
- If we use $\overline{\text{VE}}$, the goal is to find a **global optimum** (optimal weight vector, $\mathbf{w}^*$):

$$\mathbf{w}^* \quad \text{for which} \quad \overline{\text{VE}}(\mathbf{w}^*) \leq \overline{\text{VE}}(\mathbf{w}), \quad \forall \mathbf{w}$$

- Complex function approximators may converge to a **local optimum**:

$$\mathbf{w}^* \quad \text{for which} \quad \overline{\text{VE}}(\mathbf{w}^*) \leq \overline{\text{VE}}(\mathbf{w}), \quad \forall \mathbf{w} \quad \text{in some neighborhood of} \quad \mathbf{w}^*$$

9.3. Stochastic-Gradient & Semi-Gradient Methods

9.3.1. Stochastic Gradient Descent (SGD)

- SGD methods are among the most widely used of all function approximation methods and are well suited to online RL.
- The function approximator is parameterized by a weight vector with a fixed number of real components (column vector):

$$\mathbf{w} \doteq (w_1, w_2, w_3, ..., w_d)^T$$

- In SGD, we update the weight vector at each time step by moving it in the direction that minimises the error most quickly for the example shown:

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

$$\boxed{\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)]\,\nabla\hat{v}(S_t, \mathbf{w}_t)}$$

where
$\hat{v}(s, \mathbf{w}) =$ approximate value function (differentiable in $\mathbf{w}$, $\forall s \in S$)
$\alpha =$ positive step-size parameter
$\nabla f(\mathbf{w}) =$ column vector of partial derivatives of $f(\mathbf{w})$ WRT components of $\mathbf{w}$

- The equation for $\nabla f(\mathbf{w})$, the gradient of f WRT $\mathbf{w}$, is:

$$\nabla f(\mathbf{w}) \doteq \left(\frac{\partial f(\mathbf{w})}{\partial w_1}, \frac{\partial f(\mathbf{w})}{\partial w_2}, \frac{\partial f(\mathbf{w})}{\partial w_3}, \ ..., \ \frac{\partial f(\mathbf{w})}{\partial w_d}\right)^T$$

- SGD methods are ***"gradient descent"*** methods because the overall step in $\mathbf{w}_t$ is proportional to the negative gradient of the example's squared error ($\mathbf{w}_{t+1}$). This is the direction in which the error falls most rapidly.
- Gradient descent methods are called ***"stochastic"*** because the update is done on only a single example, which might have been selected stochastically.
- If α decreases as expected in satisfaction of the standard stochastic approximation conditions, then SGD is guaranteed to converge to a local optimum.

9.3.2. True Value Estimates

- When the true value function $v_\pi(S_t)$ is unknown, we can approximate it by substituting U_t in place of $v_\pi(S_t)$.
- This yields the following general SGD method for state-value prediction:

$$\boxed{\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha[U_t - \hat{v}(S_t, \mathbf{w}_t)]\,\nabla\hat{v}(S_t, \mathbf{w}_t)}$$

- If U_t is an ***unbiased*** estimate, that is, if $\mathbb{E}[U_t|S_t = s] = v_\pi(s)$ for each t, then $\mathbf{w}_t$ is guaranteed to converge to a local optimum under the usual stochastic approximation conditions for decreasing α.
- An example of an unbiased estimator is the **Monte Carlo** estimate for state S_t:

$$U_t = G_t$$

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[G_t - \hat{v}(S_t, \mathbf{w})]\,\nabla\hat{v}(S_t, \mathbf{w}), \quad \alpha > 0$$

- **Bootstrapping methods** are ***biased*** in that their target is dependent on the current value of the weights $\mathbf{w}$.
  - **Semi-gradient** (bootstrapping) methods converge reliably in linear cases.
  - Bootstrapping targets could be n**-step returns** or **dynamic programming (DP)** targets.

$$\begin{aligned}
\text{(semi-gradient)} \quad & U_t = R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}) \\
(n\text{-step}) \quad & U_t = G_{t:t+n} \\
\text{(DP)} \quad & U_t = \sum_{a,s',r} \pi(a|S_t)\, p(s', r|S_t, a)[r + \gamma\hat{v}(s', \mathbf{w}_t)]
\end{aligned}$$

9.3.3. State Aggregation

- A simple form of generalizing function approximation in which states are grouped together, with one estimated value (one component of the weight vector $\mathbf{w}$) for each group.
- A special SGD case in which:

$$\nabla\hat{v}(S_t, \mathbf{w}_t) = \left\{ \begin{array}{ll} 1 & \text{for } S_t\text{'s group's component} \\ 0 & \text{for the other components} \end{array} \right\}$$

9.4. Linear Methods

- One of the simplest cases for function approximation is when the approximate value function is a **linear combination** of the weight vector.
- Linear methods approximate the state-value function by the inner product between $\mathbf{w}$ and $\mathbf{x}(s)$:

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

$$\begin{aligned}
\text{where} \quad & \mathbf{x}(s) = \textbf{feature vector} \text{ representing state } s \\
& \mathbf{x}(s) \doteq \big(x_1(s),\ x_2(s),\ x_3(s),\ \ldots,\ x_d(s)\big)^T \\
& x_i(s) = \text{value of function } x_i : S \to \mathbb{R}
\end{aligned}$$

- 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 WRT $\mathbf{w}$ is:

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

- Thus in the linear case, the general SGD update reduces to:

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

- The semi-gradient TD(0) algorithm converges under linear function approximation:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha(R_{t+1} + \gamma\mathbf{w}_t^T\mathbf{x}_{t+1} - \mathbf{w}_t^T\mathbf{x}_t)\,\mathbf{x}_t$$

$$= \mathbf{w}_t + \alpha\Big(R_{t+1}\mathbf{x}_t - \mathbf{x}_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^T\,\mathbf{w}_t\Big)$$

$$\text{where} \quad \mathbf{x}_t = \mathbf{x}(S_t)$$

- Once the system has reached steady state, for any given $\mathbf{w}_t$ the expected next weight vector can be represented as:

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

$$\text{where} \quad b \doteq \mathbb{E}[R_{t+1}\,\mathbf{x}_t] \in \mathbb{R}^d$$
$$A \doteq \mathbb{E}\Big[\mathbf{x}_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^T\Big] \in \mathbb{R}^{d\times d}$$

- It is clear that, if the system converges, it must converge to the weight vector $\mathbf{w}_{\text{TD}}$ at which:

$$\mathbf{b} - \mathbf{A}\mathbf{w}_{\text{TD}} = \mathbf{0}$$

$$\mathbf{b} = \mathbf{A}\mathbf{w}_{\text{TD}}$$

$$\mathbf{w}_{\text{TD}} \doteq \mathbf{A}^{-1}\mathbf{b}$$

- $\mathbf{w}_{\text{TD}}$ is called the **TD fixed point**. It is the point that linear semi-gradient TD(0) converges to.
- $\mathbf{A}$ needs to be ***positive definite*** to ensure that $\mathbf{A}^{-1}$ exists:

$$y^T\mathbf{A}y > 0, \text{ for any } y \neq 0$$

- The semi-gradient n-step TD algorithm is the natural extension of the tabular n-step TD algorithm in Ch. 7 to semi-gradient function approximation:

$$\boxed{\begin{array}{l}\mathbf{w}_{t+n} \doteq \mathbf{w}_{t+n-1} + \alpha[G_{t:t+n} - \hat{v}(S_t, \mathbf{w}_{t+n-1})]\,\nabla\hat{v}(S_t, \mathbf{w}_{t+n-1}), \quad 0 \leq t < T \\ \\ G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{n-1}R_{t+n} + \gamma^n\hat{v}(S_{t+n}, \mathbf{w}_{t+n-1}), \quad 0 \leq t \leq T-n\end{array}}$$

9.4.1. Bounded Expansion

- At the TD fixed point, $\overline{\text{VE}}$ is within a bounded expansion of the lowest possible error:

$$\boxed{\overline{\text{VE}}(\mathbf{w}_{\text{TD}}) \leq \frac{1}{1-\gamma}\min_{\mathbf{w}}\overline{\text{VE}}(\mathbf{w})}$$

- That is, the asymptotic error of the TD method is no more than $\frac{1}{1-\gamma}$ times the smallest possible error (attained in the limit by the MC method).
- γ is often near 1, so the expansion factor can be quite large; therefore there is substantial potential loss in asymptotic performance with the TD method.
- A bound analogous to TD's fixed point & bound above on $\overline{\text{VE}}(\mathbf{w}_{\text{TD}})$ applies to other on-policy bootstrapping methods as well.
- One-step **action-value** methods such as ***semi-gradient Sarsa(0)*** converge to an analogous fixed point and an analogous bound.
- For **episodic** tasks, there exists a bound [Bertsekas & Tsitsiklis, 1996].

9.5. Feature Construction for Linear Methods

- Let's discuss different ways of constructing features, which is important for function approximation in linear methods.

9.5.1. Polynomials

- There may be a need to design features with higher complexity that can be captured with linear methods.
- Say for example a state has 2 numerical dimensions, $s_1, s_2 \in \mathbb{R}$. Then it is insufficient to represent this state as $\mathbf{x}(s) = (s_1, s_2)^T$ because it does not take into account any interactions between the 2 dimensions.
- We can overcome this limitation by instead representing s by the 4-D feature vector:

$$\mathbf{x}(s) = \left(1,\ s_1,\ s_2,\ s_1 s_2\right)^T$$

- For k numerical dimensions, suppose each state s corresponds to k numbers, $s_1, s_2, s_3, \dots, s_k$ with each $s_i \in \mathbb{R}$, each order-n polynomial-basis feature x_i can be written as:

$$\boxed{x_i(s) = \prod_{j=1}^{k} s_j^{c_{i,j}}}$$

where

$$c_{i,j} = \text{integer in } \{0, 1, 2, 3, \dots, n\}, \quad \text{for } n \geq 0$$
$$(n+1)^k = \text{no. of distinct features for a } k\text{-dimensional state space}$$

9.5.2. Fourier Basis

- Fourier series express periodic functions as weighted sums of sine and cosine basis functions of different frequencies, with a function f being periodic if:

$$f(x) = f(x + \tau) \quad \forall x, \text{ and for some period } \tau$$

- The 1-D, order-n Fourier cosine basis consists of the $n + 1$ features:

$$x_i(s) = \cos(i\pi s), \quad s \in [0, 1] \quad \text{for } i = 0, 1, 2, \dots, n$$

- For k-dimensional state space, the i-th feature in the order-n Fourier cosine basis is:

$$\boxed{x_i(s) = \cos(\pi \mathbf{s}^T \mathbf{c}^i)}$$

where

$$\mathbf{c}^i = \left(c_1^i, c_2^i, \dots, c_k^i\right)^T$$
$$c_j^i \in \{0, 1, 2, \dots, n\} \quad \text{for } j = 1, 2, \dots, k \text{ and } i = 1, 2, \dots, (n+1)^k$$
$$i \equiv \text{features}$$
$$j \equiv \text{dimensions}$$

9.5.3. Coarse Coding

- Mapping circles to the features of a state space for 2-dimension.
- In the diagram above, if the state is inside a circle, then the corresponding feature has the value of 1 and is said to be **present**; otherwise the feature is 0 and is said to be **absent**.
- This kind of 1-0 valued feature is called a **binary feature**.

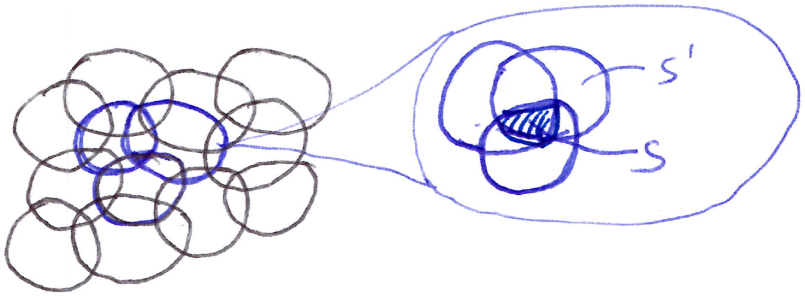

Figure 9.1. **Coarse coding.** Overlapping circular receptive fields partition the continuous state space; updating s affects s' in proportion to the number of regions that contain both states.

- Representing a state with features that overlap in this way is known as **coarse coding**.
- If we train at one state (a point in the space), then the weights of all circles intersecting that state will be affected (each circle has a corresponding weight, a single **w** component).
- Generalization in linear function approximation methods is determined by the **sizes** and **shapes** of the features' receptive fields.

9.5.4. Tile Coding

- Tile coding is a form of coarse coding for multi-dimensional continuous spaces that is flexible and computationally efficient.
- It may be the most practical feature representation for modern sequential digital computers.
- In tile coding, the receptive fields of the features are grouped into partitions of the state space.
- Each such partition is called a **tiling**, and each element of the partition is called a **tile**.
- These tiles can be sets that are overlapping, uniform, or asymmetrically distributed.
- The tiles do not need to be squares; they can be irregular shapes, horizontal/vertical/log lines.
- **Advantages:**
  - Because tile coding works with partitions, the overall number of features that are active at one time is the **same** for any state.
  - Because tile coding uses binary feature vectors, computing the approximate value function reduces to simply summing the n weight components corresponding to the n active tiles, rather than performing d multiplications.
- Generalization extends to states sharing tiles with the trained state, proportional to **tiles in common.**
  - **Uniform offsets** can introduce directional artifacts (e.g. diagonal bias).
  - **Asymmetric offsets** produce better-centered patterns.
- Tilings are offset by $\frac{w}{n}$ (the fundamental unit).
  - **Uniform offsets** use displacement vector $(1, 1)$.
  - **Asymmetric offsets** use $(1, 3)$ for superior generalization.
- Miller & Glanz (1996) recommend displacement vectors of **first odd integers** $(1, 3, 5, \ldots, 2k-1)$ for k dimensions, with n set to a **power of 2** $\geq 4k$.
- Tile **number & size** determine resolution, while tile **shape** determines generalization.
  - Square tiles generalize equally.
  - Elongated tiles generalize along their longer dimension.
  - Diagonal tiles generalize along a diagonal.
- **Mixed tile shapes** (horizontal, vertical, conjunctive) allow per-dimension generalization while still learning values for specific conjunctions. Tile choice fully determines generalization behavior.

- **Hashing** pseudorandomly collapses a large tiling into a much smaller set of tiles (each consisting of noncontiguous, disjoint regions), drastically reducing memory requirements with little performance loss.
  - This sidesteps the curse of dimensionality since memory need only match the task's real demands rather than grow exponentially with dimensions.

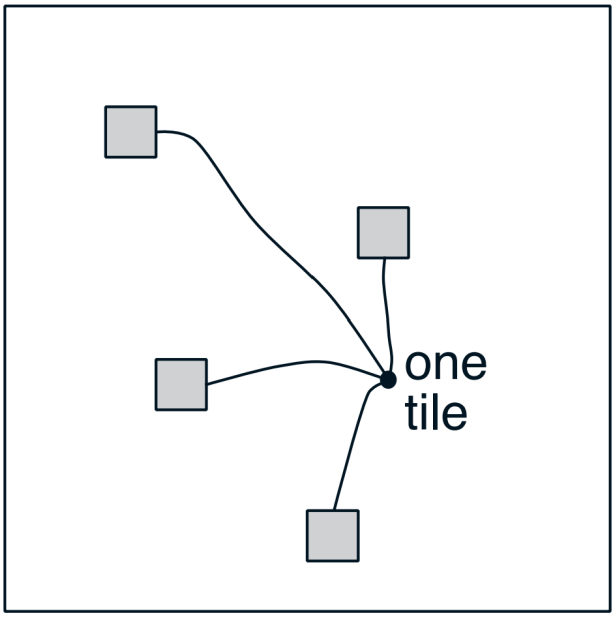

Figure 9.2. **Hashing.** 4 subtiles collapse into 1 tile in the above diagram.

9.5.5. Radial Basis Functions (RBF)

- RBFs are the natural extension/generalization of coarse coding to continuous-valued domain/feature space.
  - Rather than a feature being 0 or 1 (binary), it can take any value in the interval $[0, 1]$.
- A typical RBF feature, x_i, has a Gaussian (bell-shaped) response $x_i(s)$ dependent only on the distance between the state s and the feature's center state c_i, and relative to the feature's width σ_i:

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

- Let's see a 1-D example with a Euclidean distance metric:
- RBFs are more advantageous to binary features: they produce approximate functions that vary smoothly and are differentiable.
- An **RBF network** is a linear function approximator using RBFs for its features.
- Some learning methods for RBF networks change the features' centers and widths, making them nonlinear function approximators.
- Nonlinear methods are much more precise in fitting the target functions, but (nonlinear) RBF networks are ***much more computationally complex*** and require more manual tuning before learning.

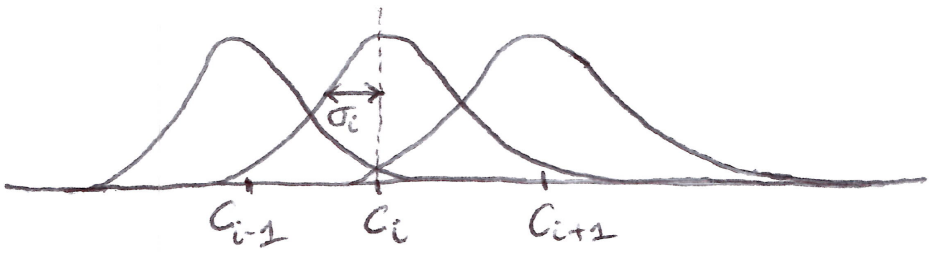

Figure 9.3. **1-D RBF.** One-dimensional radial basis function.

9.6. Selecting Step-Size Parameters Manually

- What is the best way to select α for function approximation?
- So far:
  - The theory of stochastic approximation gives us conditions on a slowly decreasing step-size sequence that are sufficient to guarantee convergence, but these tend to result in learning that is **too slow.**
  - The sample-average classical method $\alpha_t = \frac{1}{t}$ is **not appropriate** for TD methods, for nonstationary problems, or for any function approximation method.
  - For linear methods, there are recursive least-square methods that set an optimal matrix step size which can be extended to LSTD methods (seen in Section 9.8), but these require $O(d^2)$ step-size parameters, or d times more parameters than we are learning. Therefore, it is inappropriate for function approximation.
- A good rule of thumb for setting the step-size parameter of linear SGD models is:

$$\boxed{\alpha \doteq \left(\tau\, \mathbb{E}[\mathbf{x}^T\mathbf{x}]\right)^{-1}}$$

  where

  $\mathbf{x}$ = random feature vector chosen from the same distribution as input vectors in the SGD

  τ = number of experiences within which learning converges

- This rule of thumb works best if the feature vectors do not vary greatly in length; ideally $\mathbf{x}^T\mathbf{x}$ is a constant.

9.7. Nonlinear Function Approximation: Artificial Neural Networks (ANNs)

- ANNs are widely used for nonlinear function approximation.
- An ANN is a network of interconnected units that have some of the properties of neurons, the main components of nervous systems.
- The units (circles in the figure above) compute a weighted sum of their input signals, and then apply a nonlinear function, called the **activation function**, to the result.
- Some activation functions include:

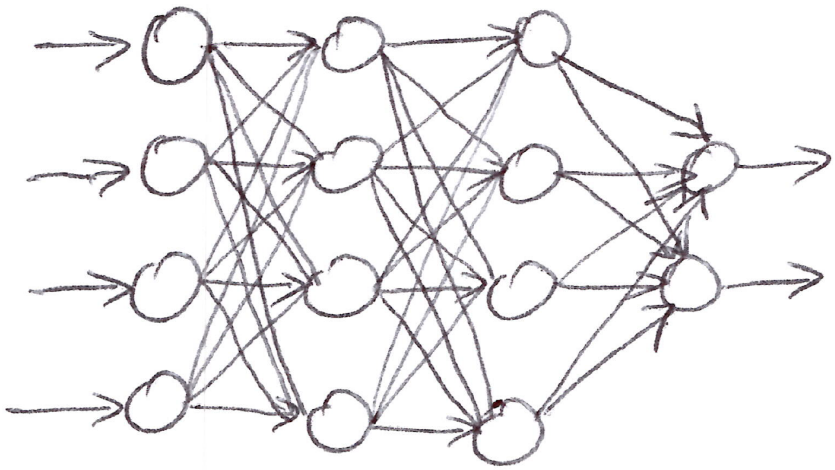

Figure 9.4. **ANN.** A generic feedforward ANN with 4 input units, 2 output units, and 2 hidden layers.

$$\text{Sigmoid:} \quad f(x) = \frac{1}{1+e^{-x}}$$

$$\text{ReLU:} \quad f(x) = \max(0, x)$$

$$\text{Binary step:} \quad f(x) = \left\{ \begin{array}{ll} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{array} \right\}$$

- ANNs can use TD errors to learn value functions, or they can aim to maximize expected reward as in a gradient bandit (Section 2.8) or a policy gradient algorithm (Chapter 13).
- Overfitting in ANNs is a problem for function approximation that can be reduced through the **dropout** method [Srivastava, Hinton, Krizhevsky, Sutskever & Salakhutdinov, 2014].
- **Deep belief networks** presented a major step towards solving the training problem of deep layers of a deep ANN [Hinton, Osindero & Teh, 2006].
- **Batch normalization** makes it easier to train deep ANNs by normalizing the output of deep layers before they feed into the following layer. This improves the learning rate of deep ANNs [Ioffe & Szegedy, 2015].
- **Deep residual learning** is another technique useful for training deep ANNs [He, Zhang, Ren & Sun, 2016].
- **Deep Convolutional Networks** have proven to be very successful in impressive RL algorithms [LeCun, Bottou, Bengio & Haffner, 1998].
- In summary, advances in the design and training of ANNs, of which we have only mentioned a few, all contribute to RL.
- Although current RL theory is mostly limited to methods using tabular or linear function approximation methods, the impressive performances of notable RL applications owe much of their success to nonlinear function approximation by multi-layer ANNs.

9.8. Least-Squares TD (LSTD)

- As established earlier, TD(0) with linear function approximator converges asymptotically (for appropriately decreasing step sizes) to the TD fixed point:

$$\boxed{\mathbf{w}_{\text{TD}} = \mathbf{A}^{-1}\mathbf{b}}$$

where

$$A \doteq \mathbb{E}\Big[\mathbf{x}_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^T\Big]$$
$$b \doteq \mathbb{E}[R_{t+1}\,\mathbf{x}_t]$$

- Instead of computing $\mathbf{w}_{\text{TD}}$ iteratively, let's compute estimates of $\mathbf{A}$ and $\mathbf{b}$ and then directly compute the TD fixed point. This is what **Least-Squares TD (LSTD)** does exactly. It forms the natural estimates:

$$\hat{\mathbf{A}}_\mathbf{t} \doteq \sum_{k=0}^{t-1} \mathbf{x_k}(\mathbf{x}_k - \gamma\mathbf{x}_{k+1})^T + \varepsilon\mathbf{I}$$

$$\hat{\mathbf{b}}_\mathbf{t} \doteq \sum_{k=0}^{t-1} R_{k+1}\,\mathbf{x}_k$$

where

$\mathbf{I}$ = identity matrix, and

$\varepsilon\mathbf{I}$, for some small $\varepsilon > 0$, ensures that $\hat{\mathbf{A}}_\mathbf{t}$ is always invertible

- The LSTD estimated TD fixed point is:

$$\boxed{\mathbf{w}_t \doteq \hat{\mathbf{A}}_\mathbf{t}^{-1}\hat{\mathbf{b}}_\mathbf{t}}$$

9.8.1. Computational Complexity

- The inverse of $\hat{\mathbf{A}}_\mathbf{t}$ computation costs $O(d^3)$, but this can be reduced to $O(d^2)$ using the **Sherman-Morrison** formula:

$$\hat{\mathbf{A}}_\mathbf{t}^{-\mathbf{1}} = \big(\hat{\mathbf{A}}_{\mathbf{t-1}} + \mathbf{x}_{\mathbf{t-1}}(\mathbf{x}_{\mathbf{t-1}} - \gamma\mathbf{x}_t)^T\big)^{-1}$$

$$\boxed{\hat{\mathbf{A}}_\mathbf{t}^{-\mathbf{1}} = \hat{\mathbf{A}}_\mathbf{t-1}^{-\mathbf{1}} - \frac{\hat{\mathbf{A}}_\mathbf{t-1}^{-\mathbf{1}}\mathbf{x}_{t-1}(\mathbf{x}_{t-1} - \gamma\mathbf{x}_t)^T\hat{\mathbf{A}}_\mathbf{t-1}^{-\mathbf{1}}}{1 + (\mathbf{x}_{t-1} - \gamma\mathbf{x}_t)^T\hat{\mathbf{A}}_\mathbf{t-1}^{-\mathbf{1}}\mathbf{x}_{t-1}}, \quad \text{for } t > 0, \text{ with } \hat{\mathbf{A}}_\mathbf{0} \doteq \varepsilon\mathbf{I}}$$

- LSTD does **not** require a step-size and as such means that it never forgets, which is sometimes desirable but often not in RL problems where both the policy and environment change with time.
- However in control applications, LSTD typically has to be combined with some other mechanism to induce forgetting, negating its advantage of not requiring a step-size parameter.

9.9. 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. This is called the **parametric** approach.
- Memory-based function approximators store training examples in memory as they arrive without updating any parameters and retrieve them from memory upon query of a state value's estimate.
- Memory-based function approximators are examples of **non-parametric** methods.
- Memory-based function approximation is sometimes called **lazy learning** because processing training examples is postponed until the system is queried to provide an output.
- Non-parametric methods could produce increasingly accurate approximations of any target function with increasing number of training examples accumulating in memory.
- There are different memory-based methods, but let's focus on **local learning**:

  - Local learning methods approximate a value function only locally in the neighborhood of the current query state by retrieving states in memory via a distance metric between the query state and training example states.
  - Local learning methods discard the local approximation after the query state is assigned a value.
- Examples of local learning methods:
  - **Nearest Neighbor**: retrieve from memory the closest state to the query state and return that example's value as the local approximation of the query state.
  - **Weighted Average**: retrieve a set of nearest neighbor examples and return a weighted average of their target values (weigh their value via some distance metric).
  - **Locally Weighted Regression**: similar to weighted average, but fits a surface to the values of a set of nearest states based on a parametric approximation method.
- Pros of non-parametric, memory-based methods:
  - Do not limit approximations to pre-specified functional forms.
  - The more data accumulated, the better the accuracy.
  - Allow for relatively immediate effect on value estimates in the neighborhood of the current state.
  - Handle/address the curse of dimensionality, which is a big problem for global approximation. For example, for a state space with K dimensions,
    * A **tabular method** storing a global approximation requires memory ***exponential in*** K***,*** while
    * Storing examples in a **memory-based method** requires only memory ***proportional to*** K***, or linear in the number of examples*** n***.***

9.10. Kernel-based Function Approximation

- Weighted average and locally weighted regression depend on assigning weights based on some distance metric between examples s' and a query state s.
- The function that assigns these weights is called a **kernel function** or simply a **kernel**.
- A kernel function $k : \mathbb{R} \to \mathbb{R}$ assigns weights to distances between states, but more generally weights do not have to depend on distances; they can also depend on some similarity measure:

$$k : S \times S \to \mathbb{R}$$

- $k(s, s')$ is a measure of the strength of generalization from s' to s.
- Kernel functions numerically express **how relevant** knowledge about any state is to any other state.
- **Kernel regression** is the memory-based method that computes a kernel weighted average of the targets of **all examples** stored in memory:

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

where

$\hat{v}(s, D)$ = value function approximation for query state s over stored examples D

D = set of stored examples

$g(s')$ = target for state s' in a stored example

- A common kernel is the Gaussian Radial Basis Function (RBF) discussed earlier in Section 9.5.5.
- Kernel regression with RBF differs from linear-based RBF in 2 ways:
  - It is **memory-based** ⇒ RBFs are centered on the states of the stored examples.
  - It is **non-parametric** ⇒ no parameters to learn.
- We can recast any linear parametric regression method with feature vectors $\mathbf{x}(s) = (x_1(s), x_2(s), x_3(s), \ldots, x_d(s))^T$ into a kernel regression as:

$$k(s, s') = \mathbf{x}(s)^T \mathbf{x}(s')$$

- Kernel methods allow evaluation in high-dimensional feature spaces, only using stored examples, and without the need for a complicated, parametric model. This is the so-called **kernel trick** that forms the basis for many machine learning methods.

9.11. Looking Deeper at On-Policy Learning: Interest & Emphasis

- So far we have treated all encountered states with equal importance, however it seems more likely that we often have more interest in some states than others.
- For this scenario, we introduce 2 new concepts: **Interest** and **Emphasis**.

9.11.1. Interest

- Interest is the degree to which we value an accurate estimate of the values of a given state. How interested are we in the accurate valuation of a given state at time t?

$$\mathcal{I}_t \in [0, 1]$$

9.11.2. Emphasis

- Emphasis is a non-negative scalar random variable that multiplies the learning update and thus emphasizes or de-emphasizes the learning done at time t.
- The general n-step learning update is:

$$\mathbf{w}_{t+n} \doteq \mathbf{w}_{t+n-1} + \alpha M_t [G_{t:t+n} - \hat{v}(S_t, \mathbf{w}_{t+n-1})] \nabla \hat{v}(S_t, \mathbf{w}_{t+n-1}), \quad 0 \leq t < T$$

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \hat{v}(S_{t+n}, \mathbf{w}_{t+n-1}), \quad 0 \leq t \leq T - n$$

- The emphasis is determined recursively from the interest:

$$\boxed{M_t = \mathcal{I}_t + \gamma^n M_{t-n}, \quad 0 \leq t < T}$$

$$M_t \doteq 0, \quad \forall t < 0$$

9.12. Summary

- **Generalization** is a must for RL systems for AI applications and **supervised-learning function approximation** helps achieve this.
- The prediction objective $\overline{\text{VE}}(\mathbf{w})$, called the **mean squared value error,** gives us a clear way to rank different value-function approximations in the on-policy case.
- Most techniques use stochastic gradient descent (SGD) to find the set of weight parameters that minimize $\overline{\text{VE}}(\mathbf{w})$.
- Linear methods converge to the global optimum under certain conditions.
- Features constructed for linear function approximations could be represented as: **polynomials, Fouriers, coarse codings, tile codings, and radial basis functions (RBFs).**
- A huge success in notable RL applications could be attributed to multi-layer ANNs as nonlinear function approximators.
- Non-parametric models help us **avoid the curse of dimensionality.**
- **Interest and Emphasis** enable us to focus the function approximation on the states we're more interested in.

10. On-Policy Control with Approximation

- Let's dive into the control problem now with parametric approximation of the action-value function $\hat{q}(s, a, \mathbf{w}) \approx q_*(s, a)$, where $\mathbf{w} \in \mathbb{R}^d$ is a **finite-dimensional weight vector.**
- We'll focus on **semi-gradient Sarsa**, the natural extension of semi-gradient TD(0) to action values and to on-policy control.
- We'll look at this extension in both the episodic and continuing case.
- We'll look at n-step linear Sarsa.

10.1. Episodic Semi-gradient Control

- The extension of the semi-gradient prediction methods of Chapter 9 to action values is straightforward.
- It is the approximate action-value function, $\hat{q} \approx q_\pi$, that is represented as a parametrized functional form with weight vector $\mathbf{w}$.
- Before, the training examples had the form $S_t \mapsto U_t$; now the examples have the form $S_t, A_t \mapsto U_t$.
- The update target U_t can be any approximation of $q_\pi(S_t, A_t)$, including the usual backed-up values such as the full Monte Carlo (MC) return G_t or any n-step Sarsa return $G_{t:t+n}$.
- The general gradient-descent update for action-value prediction is:

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

- The update for the one-step Sarsa method is:

$$\boxed{\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha[R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t)] \nabla \hat{q}(S_t, A_t, \mathbf{w}_t)}$$

- This method is called **episodic semi-gradient one-step Sarsa**. For a constant policy, this method converges in the same way that TD(0) does with the same kind of error bound.
- **Control** = action-value prediction + policy improvement & action selection:

$$\boxed{\begin{aligned} a, S_{t+1} \longrightarrow \hat{q}(S_{t+1}, a, \mathbf{w}_t) \longrightarrow A^*_{t+1} = \arg\max_a \hat{q}(S_{t+1}, a, \mathbf{w}_t) \\ \longrightarrow \varepsilon\text{-greedy policy improvement} \longrightarrow \varepsilon\text{-greedy action selection} \end{aligned}}$$

- Linear function approximation for the action-value function is:

$$\hat{q}(s, a, \mathbf{w}) \doteq \mathbf{w}^T \mathbf{x}(s, a) = \sum_{i=1}^{d} w_i \cdot x_i(s, a)$$

10.2. Semi-gradient n-step Sarsa

- We use an n-step return as the update target for episodic semi-gradient n-step Sarsa. The n-step return generalizes from its tabular form to a function approximation form:

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n \hat{q}(S_{t+n}, A_{t+n}, \mathbf{w}_{t+n-1}), \quad t + n < T$$

$$\text{with } G_{t:t+n} \doteq G_t \text{ if } t + n \geq T$$

- The n-step update equation is:

$$\boxed{\mathbf{w}_{t+n} \doteq \mathbf{w}_{t+n-1} + \alpha[G_{t:t+n} - \hat{q}(S_t, A_t, \mathbf{w}_{t+n-1})] \nabla \hat{q}(S_t, A_t, \mathbf{w}_{t+n-1}), \quad 0 \leq t < T}$$

- Performance is best if an intermediate level of bootstrapping is used ($n > 1$).

10.3. Average Reward: A New Problem Setting for Continuing Tasks

- Average reward applies to continuing problems for goal formulation in MDPs.
- Average reward uses **no discounting**; the agent has the same level of care for immediate and delayed rewards.
- Average reward setting is more commonly considered in dynamic programming and less commonly in reinforcement learning (RL).
- The discounted setting is problematic with function approximation, hence the need for average reward to replace it.
- 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, denoted as $r(\pi)$:

$$\begin{aligned} r(\pi) &\doteq \lim_{h\to\infty} \frac{1}{h} \sum_{t=1}^{h} \mathbb{E}[R_t \mid S_0, A_{0:t-1} \sim \pi] \\ &= \lim_{t\to\infty} \mathbb{E}[R_t \mid S_0, A_{0:t-1} \sim \pi] \\ &= \sum_s \mu_\pi(s) \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\, r \end{aligned}$$

- The expectations in the above equations are conditioned on the initial state S_0, and on the subsequent actions $A_0, A_1, \ldots, A_{t-1}$, being taken according to π.
- The 2nd and 3rd equations above hold if the MDP is **ergodic**, i.e., if the steady-state distribution exists and is independent of the starting state S_0:

$$\mu_\pi(s) \doteq \lim_{t\to\infty} \Pr\{S_t = s \mid A_{0:t-1} \sim \pi\}$$

- In an ergodic MDP, the starting state can have only a temporary effect, but in the long run the expectation of being in a state depends only on the policy and the MDP transition probabilities.
- Ergodicity is sufficient but not necessary to guarantee the existence of the limit in the $r(\pi)$ equation above.
- It may be adequate practically to simply order policies according to their average reward per time step, otherwise called the **return rate**.
- All policies that reach the maximal value of $r(\pi)$ are optimal.
- The steady-state distribution μ_π is the special distribution under which, if you select actions according to π, you remain in the same distribution, i.e., for which:

$$\sum_s \mu_\pi(s) \sum_a \pi(a|s)\, p(s'|s, a) = \mu_\pi(s')$$

- In the average-reward setting, returns are defined in terms of differences between rewards and the average reward; this is called the **differential return**:

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

- The corresponding value functions for the differential return are known as **differential value functions**:

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\ q_\pi(s, a) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] \end{aligned}$$

- Differential value functions also have Bellman equations:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{r,s'} p(s', r|s, a)[r - r(\pi) + v_\pi(s')]$$

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

$$v_*(s) = \max_a \sum_{r,s'} p(s', r|s, a) \Big[r - \max_\pi r(\pi) + v_*(s') \Big]$$

$$q_*(s, a) = \sum_{r,s'} p(s', r|s, a) \Big[r - \max_\pi r(\pi) + \max_{a'} q_*(s', a') \Big]$$

- The differential form of the 2 TD errors:

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

$$\text{where} \quad \bar{R}_t = \text{average reward } r(\pi) \text{ estimate at time } t$$

- Most of the algorithms covered so far don't change for the average-reward setting. For example, the semi-gradient Sarsa average-reward version is the same as the regular version except with the differential version of the TD error:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\, \delta_t \nabla \hat{q}(S_t, A_t, \mathbf{w}_t)$$

10.4. Deprecating the Discounted Setting

- For the tabular case, the continuing, discounted problem formulation is useful, but in the approximate case, is this problem formulation necessary?
- Should we use the discounted reward or average reward in continuing tasks?
- It turns out that the average of the discounted return is proportional to the average reward.
- The ordering of all policies in the average discounted return setting would be exactly the same as in the average-reward setting.
- This idea of the **futility of discounting in continuing problems** can be proven by the **symmetry argument.**
  - Let's choose an objective that saves discounting by summing discounted values over the distribution with which states occur under the policy (where $v_\pi^\gamma \equiv$ discounted value function):

$$\begin{aligned}
J(\pi) &= \sum_s \mu_\pi(s)\, v_\pi^\gamma(s) \\
&= \sum_s \mu_\pi(s) \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)[r + \gamma v_\pi^\gamma(s')] \\
&= r(\pi) + \sum_s \mu_\pi(s) \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)\, \gamma v_\pi^\gamma(s') \\
&= r(\pi) + \gamma \sum_{s'} v_\pi^\gamma(s') \sum_s \mu_\pi(s) \sum_a \pi(a|s)\, p(s'|s, a) \\
&= r(\pi) + \gamma \sum_{s'} v_\pi^\gamma(s')\, \mu_\pi(s') \\
&= r(\pi) + \gamma J(\pi) \\
&= r(\pi) + \gamma(r(\pi) + \gamma J(\pi)) \\
&= r(\pi) + \gamma r(\pi) + \gamma^2 J(\pi) \\
&= r(\pi) + \gamma r(\pi) + \gamma^2 r(\pi) + \gamma^3 r(\pi) + \gamma^4 r(\pi) + ... \\
&= r(\pi)[1 + \gamma + \gamma^2 + \gamma^3 + ...]
\end{aligned}$$

$$\boxed{J(\pi) = \left(\frac{1}{1-\gamma}\right) r(\pi)}$$

  - * *The proposed discounted objective orders policies identically to the undiscounted (average reward) objective.*
  - * *The discount rate γ does not influence the ordering.*
- The root cause of the difficulties with the discounted control setting is that with function approximation we have lost the policy improvement theorem.
- Now if we change the policy to improve the discounted value of one state, we are no longer guaranteed to have improved the overall policy.

10.5. Differential Semi-gradient n-step Sarsa

- We need an n-step version of the TD error in order to generalize to n-step bootstrapping.
- Let's generalize the n-step return to its differential form, with function approximation:

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

$$\text{where} \quad \bar{R} \equiv \text{an estimate of } r(\pi), \quad n \geq 1 \;\&\; t+n < T$$
$$G_{t:t+n} \doteq G_t \quad \text{if } t+n \geq T$$

- The n-step TD error is:

$$\delta_t \doteq G_{t:t+n} - \hat{q}(S_t, A_t, \mathbf{w})$$

10.6. Summary

- Moving from prediction to control with function approximation is straightforward in episodic tasks (replacing $\hat{v}$ with $\hat{q}$ in semi-gradient Sarsa and n-step Sarsa).
- In continuing tasks with approximation, discounting breaks down because policies can no longer be ordered unambiguously without a start-state distribution.
- Optimizing the **average reward** $r(\pi)$ per time step resolves this cleanly, giving every policy a single scalar performance score.
- The average-reward framework introduces **differential** returns, value functions, Bellman equations, and TD errors that closely mirror their discounted counterparts and yield parallel differential control algorithms.

11. Off-Policy Methods with Approximation

- Let's discuss the extension of off-policy methods from the tabular case (Ch. 6 & 7) to function approximation.
- We'll explore the convergence problems, the theory of linear function approximation, the notion of learnability, and stronger convergence off-policy algorithms.
- Off-policy learning with function approximation has 2 challenges:
  1. Finding the target of the update.
  2. The off-policy distribution of updates does not match that of the on-policy distribution.

11.1. Semi-gradient Methods

- Let's discuss the extension of previous off-policy methods to function approximation as semi-gradient methods.
- This is how we find the update target (or change it) to address the first challenge.
- Recall the semi-gradient update:

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

$$U_t = R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t)$$

- In the tabular case, we update the array (V or Q), but now we update the weight vector $\mathbf{w}$.
- Many off-policy algorithms use the per-step importance sampling ratio:

$$\rho_t \doteq \rho_{t:t} = \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$$

- The off-policy, semi-gradient **TD(0)** is same as that of the on-policy TD(0) except for the addition of the ρ_t term:

$$\boxed{\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \rho_t \, \delta_t \nabla \hat{v}(S_t, \mathbf{w}_t)}$$

$$\begin{aligned} \text{(episodic)} \quad & \delta_t = R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t) \\ \text{(continuing)} \quad & \delta_t = R_{t+1} - \bar{R}_t + \hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t) \end{aligned}$$

- For action values, the off-policy, semi-gradient **Expected Sarsa** update rule is (no importance sampling):

$$\boxed{\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \, \delta_t \nabla \hat{q}(S_t, A_t, \mathbf{w}_t)}$$

$$\begin{aligned} \text{(episodic)} \quad & \delta_t = R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) \hat{q}(S_{t+1}, a, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t) \\ \text{(continuing)} \quad & \delta_t = R_{t+1} - \bar{R}_t + \sum_a \pi(a|S_{t+1}) \hat{q}(S_{t+1}, a, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t) \end{aligned}$$

- The lack of use of importance sampling in Expected Sarsa is an unclear choice since we might want to weight different state-action pairs **differently** once they all contribute to the same overall approximation. This issue can only be properly resolved by more thorough understanding of the **theory of function approximation** in RL.
- In the multi-step generalizations of the algorithms, both the state-value and action-value algorithms involve importance sampling. For example, the off-policy, semi-gradient **n-step Sarsa** update is:

$$\boxed{\mathbf{w}_{t+n} \doteq \mathbf{w}_{t+n-1} + \alpha\rho_{t+1}\cdots\rho_{t+n}[G_{t:t+n} - \hat{q}(S_t, A_t, \mathbf{w}_{t+n-1})]\,\nabla\hat{q}(S_t, A_t, \mathbf{w}_{t+n-1})}$$

$$\begin{aligned}
\text{(episodic)} \quad & G_{t:t+n} = R_{t+1} + ... + \gamma^{n-1}R_{t+n} + \gamma^n\hat{q}(S_{t+n}, A_{t+n}, \mathbf{w}_{t+n-1}) \\
\text{(continuing)} \quad & G_{t:t+n} = R_{t+1} - \bar{R}_t + ... + R_{t+n} - \bar{R}_{t+n-1} + \hat{q}(S_{t+n}, A_{t+n}, \mathbf{w}_{t+n-1})
\end{aligned}$$

$$\text{where } \rho_k = 1 \quad \text{for } k \geq T \quad \text{and} \quad G_{t:t+n} = G_t \quad \text{if } t+n \geq T$$

- The off-policy, semi-gradient **n-step backup tree** (no importance sampling) algorithm is:

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

$$G_{t:t+n} \doteq \hat{q}(S_t, A_t, \mathbf{w}_{t+n}) + \sum_{k=t}^{t+n-1} \delta_k \prod_{i=t+1}^{k} \gamma\pi(A_i|S_i)$$

where δ_t is the Expected Sarsa TD error defined earlier in this section.

11.2. Examples of Off-Policy Divergence

- Now let's discuss the 2nd off-policy function approximation challenge.
- We'll look at some instructive counterexamples where the semi-gradient algorithm diverges.

11.2.1. Example 1

Consider part of a larger MDP with 2 states whose estimated values are w and $2w$:



Figure 11.1. **Simple Counterexample.** 2-state part of an MDP.

- w updates will diverge to infinity, since the transition will always look good (higher next-state estimated value than current state estimated value).
- The TD error on a transition between the 2 states is:

$$\begin{aligned}\delta_t &= R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t) \\ &= 0 + \gamma \cdot 2\mathbf{w}_t - \mathbf{w}_t \\ &= (2\gamma - 1)\,\mathbf{w}_t\end{aligned}$$

- The off-policy, semi-gradient TD(0) update is:

$$\begin{aligned}\mathbf{w}_{t+1} &= \mathbf{w}_t + \alpha\rho_t\,\delta_t \nabla \hat{v}(S_t, \mathbf{w}_t) \\ &= \mathbf{w}_t + (\alpha)(1)[(2\gamma - 1)\mathbf{w}_t]\,(1) \\ &= \mathbf{w}_t[1 + \alpha(2\gamma - 1)]\end{aligned}$$

$$\begin{aligned}\Rightarrow \quad & 1 + \alpha(2\gamma - 1) > 1 \\ & \alpha(2\gamma - 1) > 0 \\ & 2\gamma - 1 > 0 \\ & \gamma > \tfrac{1}{2} \quad \longrightarrow \quad w \to \pm\infty\end{aligned}$$

11.2.2. Example 2 (Baird's Counterexample)

- Now let's look at an entire complete system with instability (divergence).
- Consider the episodic 7-state, 2-action MDP shown below.

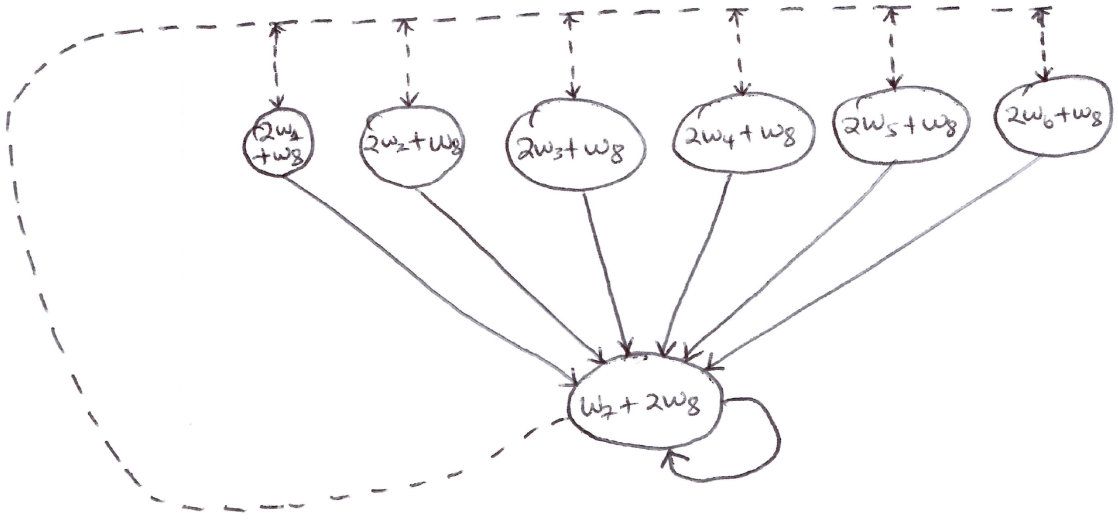

Figure 11.2. **Baird's Counterexample.** Episodic 7-state, 2-action MDP.

- **Assumptions/knowns:**
  - $b(\text{dashed}\,|\,\cdot) = 6/7$
  - $b(\text{solid}\,|\,\cdot) = 1/7$
  - $\pi(a\,|\,\cdot) = \pi(\text{solid}\,|\,\cdot) = 1$
  - $R = 0$ (on all transitions)

  - $\gamma = 0.99$
  - The state values are estimated via linear parametrization.
- The estimated value of the leftmost state is $2w_1 + w_8$, which corresponds to a feature vector for the 1st state being:

$$\mathbf{x}(1) = (2, 0, 0, 0, 0, 0, 0, 1)^T$$

$$R = 0 \quad \therefore \quad v_\pi(s) = 0 \; \forall s, \text{ which can be exactly approximated if } \mathbf{w} = \mathbf{0}$$

- Since there are 8 components of the weight vector (more than the 7 non-terminal states), there exist many solutions.
- Applying semi-gradient TD(0) to this problem will cause the weights to diverge to infinity. This also applies for the dynamic programming (DP) case.
- The semi-gradient DP update is:

$$\mathbf{w}_{k+1} \doteq \mathbf{w}_k + \frac{\alpha}{|S|} \sum_s \left(\mathbb{E}_\pi [R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_k) \mid S_t = s] - \hat{v}(s, \mathbf{w}_k) \right) \nabla \hat{v}(s, \mathbf{w}_k)$$

- This example shows that even the simplest combination of bootstrapping and function approximation can be unstable in the off-policy case.
  - Simplest bootstrapping: DP and TD.
  - Simplest function approximation: linear, semi-gradient descent method.

11.2.3. Example 3 (Tsitsiklis & Van Roy's Counterexample)

This extends Example 1 with a terminal state and $R = 0$:

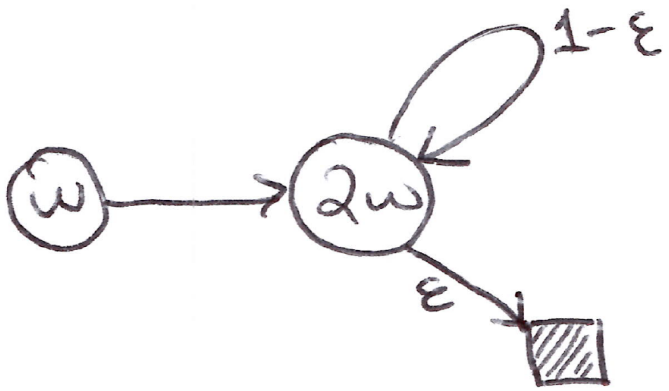

Figure 11.3. **Tsitsiklis & Van Roy's Counterexample.** Extension of Example 1 with probability ε of transitioning to the terminal state (shaded).

- Let's find w_{k+1} at each step that minimizes the $\overline{\text{VE}}$ between the estimated value and the expected one-step return:

$$
\begin{aligned}
w_{k+1} &= \arg\min_{w \in \mathbb{R}} \sum_{s \in S} \left(\hat{v}(s, w) - \mathbb{E}_\pi[R_{t+1} + \gamma \hat{v}(S_{t+1}, w_k) \mid S_t = s]\right)^2 \\
&= \arg\min_{w \in \mathbb{R}} \left(w - \gamma \cdot 2w_k\right)^2 + \left(2w - (1-\varepsilon)\gamma \cdot 2w_k\right)^2 \\
&= \left(\frac{6 - 4\varepsilon}{5}\right) \gamma w_k
\end{aligned}
$$

- The sequence $\{w_k\}$ diverges when $\gamma > \dfrac{5}{6-4\varepsilon}$ and $w_0 \neq 0$.

11.2.4. Takeaways

- Instability can be prevented by using special methods for function approximation.
- These special methods guarantee stability because they do not extrapolate from the observed targets. They are called **averagers**.
- Averagers include:
  1. Nearest neighbor
  2. Locally weighted regression
  3. Tile coding
  4. Artificial neural networks (ANNs)

11.3. The Deadly Triad

- The danger of instability and divergence arises when we combine these 3 elements, which make up the **deadly triad**:
  - Function approximation
  - Bootstrapping
  - Off-policy training
- Instability can be avoided if one of the elements is absent:
  - Function approximation cannot be given up (needed for large-scale problems).
  - Bootstrapping can be given up but at the cost of computational and data efficiency.
  - Off-policy can be given up (replace Q-learning with Sarsa).
  - There is no perfect solution as we still need off-policy for planning and parallel learning.

11.4. Linear Value-Function Geometry

- To better understand the stability challenge of off-policy learning, let's think about value-function approximation **more abstractly and independently** of how learning is done.
- Let's consider the case with 3 states $S = \{s_1, s_2, s_3\}$ and 2 parameters $\mathbf{w} = (w_1, w_2)^T$.
  - All value functions exist in a 3-D space, however the parameters provide a 2-D subspace.
  - Any weight vector $\mathbf{w} = (w_1, w_2)^T$ is a point in the 2-D subspace and thus also a complete value function $v_{\mathbf{w}}$ that assigns values to all 3 states.
  - In linear value-function approximation, the subspace is a simple plane.
- How do we represent v_π in the d-dimensional space?
  - We need to perform a projection operation.
  - TD methods present other solutions.

11.4.1. Projection Operation

- For the projection operation, the distance between value functions using the norm is:

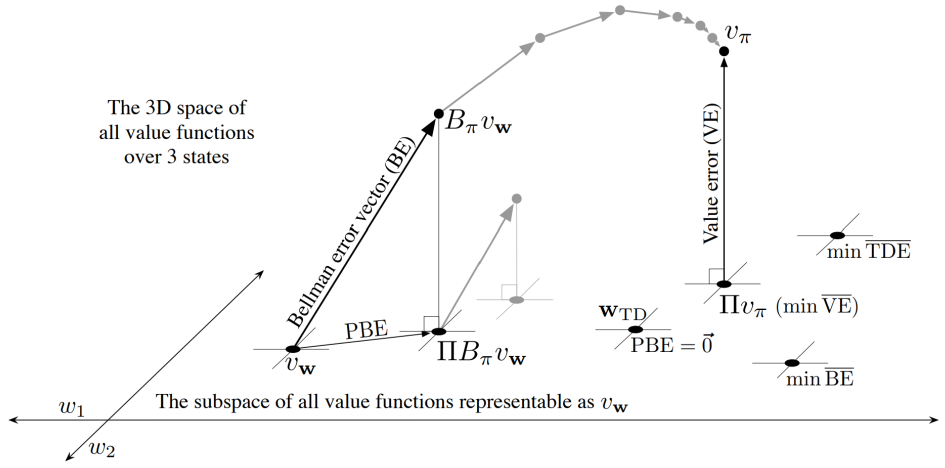

Figure 11.4. **The Geometry of Linear Value-Function Approximation.** Value functions over three states form a 3D ambient space, while a 2-parameter linear approximator $\mathbf{w} = (w_1, w_2)^T$ spans a 2D plane within it. Projecting the true value vector v_π onto this plane via Π yields the minimum-VE solution, whereas minimizing BE, PBE, or TDE generally lands on distinct points in the plane.

$$\|v\|_\mu^2 \doteq \sum_{s\in S} \mu(s)\, v(s)^2$$

$$\overline{\text{VE}}(\mathbf{w}) = \|v_\mathbf{w} - v_\pi\|_\mu^2$$

$$\Pi\, v \doteq v_\mathbf{w}$$

$$\text{where } \mathbf{w} = \arg\min_{\mathbf{w}\in\mathbb{R}^d} \|v - v_\mathbf{w}\|_\mu^2 \quad \text{and} \quad \Pi \equiv \text{projection operator}$$

- The representable value function that is closest to the true value function V_π is its projection ΠV_π (MC method asymptotic solution).
- **Projection matrix**: with $\mathbf{D} \equiv |S| \times |S|$ diagonal matrix with $\mu(s)$ on the diagonal and $\mathbf{X} \equiv |S| \times d$ matrix whose rows are the feature vectors $\mathbf{x}(s)^T$:

  $$\Pi \doteq \mathbf{X}(\mathbf{X}^T\mathbf{D}\mathbf{X})^{-1}\mathbf{X}^T\mathbf{D}$$

  If the inverse does not exist, the pseudoinverse is substituted. Using these matrices, the squared norm of a vector can be written as:

  $$\|v\|_\mu^2 = v^T\mathbf{D}\, v$$

  and the approximate linear value function written as:

  $$v_\mathbf{w} = \mathbf{X}\mathbf{w}$$

11.4.2. TD Solutions

Bellman Error

- The true value function v_π solves the Bellman equation exactly.
- The **Bellman error** shows how far off $v_\mathbf{w}$ is from v_π. The Bellman error at state s is:

$$\begin{aligned}\bar{\delta}_\mathbf{w}(s) &\doteq \left(\sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma v_\mathbf{w}(s')]\right) - v_\mathbf{w}(s) \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\mathbf{w}(S_{t+1}) - v_\mathbf{w}(S_t) \mid S_t = s, A_t \sim \pi]\end{aligned}$$

- The Bellman error is the expectation of the TD error.
- The vector of all the Bellman errors, at all states, $\bar{\delta}_\mathbf{w} \in \mathbb{R}^{|S|}$, is called the **Bellman error vector** (BE).
- The overall size of BE is the **Mean Squared Bellman Error**, $\overline{\text{BE}}$:

$$\overline{\text{BE}}(\mathbf{w}) = \|\bar{\delta}_\mathbf{w}\|_\mu^2$$

- The **Bellman operator** $B_\pi : \mathbb{R}^{|S|} \to \mathbb{R}^{|S|}$ is defined by:

$$(B_\pi v)(s) \doteq \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma v(s')], \quad \forall s \in S \text{ and } v : S \to \mathbb{R}$$

$$\bar{\delta}_\mathbf{w} = B_\pi v_\mathbf{w} - v_\mathbf{w}$$

$$v_\pi = B_\pi v_\pi$$

- The projection of the Bellman error vector back into the representable space creates the **Projected Bellman Error** (**PBE**) vector:

$$\text{PBE} = \Pi\, \bar{\delta}_\mathbf{w}$$

- The size of PBE, in the norm, is another measure of error in the approximate value function, called the **Mean Square Projected Bellman Error**, $\overline{\text{PBE}}$:

$$\overline{\text{PBE}}(\mathbf{w}) = \|\Pi\,\bar{\delta}_{\mathbf{w}}\|^2_\mu$$

- With linear function approximation, there always exists an approximate value function (within the subspace) with zero $\overline{\text{PBE}}$; this is the TD fixed point $\mathbf{w}_{\text{TD}}$.

11.5. Gradient Descent in the Bellman Error

- Let's apply the approach of SGD in dealing with the challenge of stability in off-policy learning.

11.5.1. TD Error (Naive Residual-Gradient Algorithm)

- Let's take the minimization of the expected square of the TD error as the objective, TD(0):

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

- Using the TD error, we can find the Mean Squared TD error, the objective function $\overline{\text{TDE}}$:

$$\begin{aligned}
\overline{\text{TDE}}(\mathbf{w}) &= \sum_{s\in S} \mu(s)\,\mathbb{E}\big[\delta_t^2 \mid S_t = s, A_t \sim \pi\big] \\
&= \sum_{s\in S} \mu(s)\,\mathbb{E}\big[\rho_t\,\delta_t^2 \mid S_t = s, A_t \sim b\big] \\
&= \mathbb{E}_b\big[\rho_t\,\delta_t^2\big]
\end{aligned}$$

- Following the standard SGD approach, the per-step update based on a sample of this expected value:

$$\begin{aligned}
\mathbf{w}_{t+1} &= \mathbf{w}_t - \tfrac{1}{2}\alpha\nabla(\rho_t\,\delta_t^2) \\
&= \mathbf{w}_t - \alpha\rho_t\,\delta_t\nabla\delta_t \\
&= \mathbf{w}_t + \alpha\rho_t\,\delta_t(\nabla\hat{v}(S_t, \mathbf{w}_t) - \gamma\nabla\hat{v}(S_{t+1}, \mathbf{w}_t))
\end{aligned}$$

- This is the same as the semi-gradient TD algorithm except for the additional final term.
- This method is **naive** because it achieves temporal smoothing-like behavior rather than accurate prediction by penalizing all TD errors.

11.5.2. Bellman Error (Residual-Gradient Algorithm)

- Consider the minimization of the Bellman error (if the exact values are learned, the Bellman error is zero everywhere).
- This yields the **residual gradient algorithm**:

$$\begin{aligned}
\mathbf{w}_{t+1} &= \mathbf{w}_t - \tfrac{1}{2}\alpha\nabla\Big(\mathbb{E}_\pi[\delta_t]^2\Big) \\
&= \mathbf{w}_t - \tfrac{1}{2}\alpha\nabla\Big(\mathbb{E}_b[\rho_t\,\delta_t]^2\Big) \\
&= \mathbf{w}_t - \alpha\,\mathbb{E}_b[\rho_t\,\delta_t]\,\nabla\mathbb{E}_b[\rho_t\,\delta_t] \\
&= \mathbf{w}_t - \alpha\,\mathbb{E}_b[\rho_t(R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w}))]\,\mathbb{E}_b[\rho_t\nabla\delta_t] \\
&= \mathbf{w}_t + \alpha[\mathbb{E}_b[\rho_t(R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}))] - \hat{v}(S_t, \mathbf{w})][\nabla\hat{v}(S_t, \mathbf{w}) - \gamma\,\mathbb{E}_b[\rho_t\nabla\hat{v}(S_{t+1}, \mathbf{w})]]
\end{aligned}$$

- Two ways to make the residual-gradient algorithm work:
  - In the case of deterministic environments.
  - Obtain 2 independent samples of the next state S_{t+1} from S_t.
- In both ways above, the algorithm is guaranteed to converge to a minimum of the $\overline{\text{BE}}$ under the usual conditions on the step-size parameter.
- However, there are at least 3 ways in which the convergence of the residual-gradient algorithm is unsatisfactory:
  - Very slow.
  - Converges to the wrong values.

– A problem with the $\overline{\text{BE}}$ objective covered in the next section.

11.6. The Bellman Error is Not Learnable

- The Bellman error is not learnable from the observed sequence of feature vectors, actions, and rewards.
- Since the Bellman error objective cannot be learned from the observable data, this is the strongest reason not to seek it.
- Examples of non-learnable Markov Reward Processes (MRPs):

11.6.1. Example 1



Figure 11.5. **Value Error (VE) Learnability Counterexample.** Deterministic MRP pair with an endless stream of 0s and 2s.

- These MRPs have a deterministic reward with observable data of an endless stream of 0s and 2s.
- We cannot learn if the MRP has one state or two, or is stochastic or deterministic.
- The pair of MRPs shows that the $\overline{\text{VE}}$ objective is not learnable:

$$\overline{\text{VE}}(\mathbf{w}) \doteq \sum_{s \in S} \mu(s)[v_\pi(s) - \hat{v}(s, \mathbf{w})]^2$$

- The $\overline{\text{VE}}$ is not learnable, but the parameter that optimizes it is!
- We introduce a learnable natural objective function that is always observable. This is the error between the value estimate at each time and the return from that time, called the **return error**. The **Mean Square Return Error** ($\overline{\text{RE}}$) is the expectation, under μ, of the square of this return error.
- $\overline{\text{RE}}$ in the on-policy case is:

$$\begin{aligned}\overline{\text{RE}}(\mathbf{w}) &= \mathbb{E}\Big[(G_t - \hat{v}(S_t, \mathbf{w}))^2\Big] \\ &= \overline{\text{VE}}(\mathbf{w}) + \mathbb{E}\Big[(G_t - v_\pi(S_t))^2\Big]\end{aligned}$$

- The $\overline{\text{BE}}$ can be computed from knowledge of the MDP but is not learnable from data, and its minimum solution is not learnable.

11.6.2. Example 2



Figure 11.6. **Bellman Error (BE) Learnability Counterexample.** Complex Deterministic MRP pair with same distribution but different minimizing parameter vector.

- The example above serves as a counterexample to the learnability of the Bellman error.
- The 2 MRPs generate the same data distribution but have different minimizing parameter vectors, proving that the optimal parameter vector is not a function of the data and thus cannot be learned from it.
- Other bootstrapping objectives, like $\overline{\text{PBE}}$ and $\overline{\text{TDE}}$, are learnable from data and yield optimal solutions different from each other and that of $\overline{\text{BE}}$.
- $\overline{\text{BE}}$ is limited to model-based settings, therefore $\overline{\text{PBE}}$ is preferred.

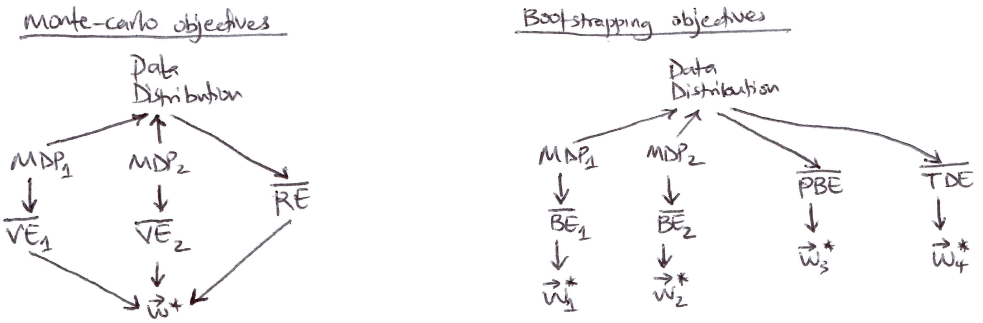

Figure 11.7. **Causal Relationships among the data distribution, MDPs & various objectives.** Monte-Carlo & Bootstrapping objectives.

11.7. Gradient-TD Methods

- Let's consider SGD methods for minimizing the $\overline{\text{PBE}}$.
- True SGD methods, **Gradient-TD methods**, have robust convergence properties even under off-policy training and nonlinear function approximation.
- In the linear case, there exists an exact solution, the TD fixed point $\mathbf{w}_{\text{TD}}$, at which the $\overline{\text{PBE}}$ is zero.
- This solution via **least-squares** methods yields a $O(d^2)$ complexity; however, we want an SGD method with $O(d)$ that converges robustly.
- Let's derive an SGD method for the $\overline{\text{PBE}}$ assuming linear function approximation:

$$\begin{aligned}\overline{\text{PBE}}(\mathbf{w}) &= \|\Pi\,\bar{\delta}_{\mathbf{w}}\|^2_\mu \\ &= \big(\Pi\,\bar{\delta}_{\mathbf{w}}\big)^T\,\mathbf{D}\,\Pi\,\bar{\delta}_{\mathbf{w}} \\ &= \bar{\delta}_{\mathbf{w}}^T\Pi^T\mathbf{D}\,\Pi\,\bar{\delta}_{\mathbf{w}} \\ &= \bar{\delta}_{\mathbf{w}}^T\mathbf{D}\mathbf{X}(\mathbf{X}^T\mathbf{D}\mathbf{X})^{-1}\,\mathbf{X}^T\mathbf{D}\,\bar{\delta}_{\mathbf{w}} \\ &= \big(\mathbf{X}^T\mathbf{D}\,\bar{\delta}_{\mathbf{w}}\big)^T(\mathbf{X}^T\mathbf{D}\mathbf{X})^{-1}\big(\mathbf{X}^T\mathbf{D}\,\bar{\delta}_{\mathbf{w}}\big)\end{aligned}$$

$\Big(\text{using } \Pi = \mathbf{X}(\mathbf{X}^T\mathbf{D}\mathbf{X})^{-1}\,\mathbf{X}^T\mathbf{D} \text{ and the identity } \Pi^T\mathbf{D}\Pi = \mathbf{D}\mathbf{X}(\mathbf{X}^T\mathbf{D}\mathbf{X})^{-1}\,\mathbf{X}^T\mathbf{D}\Big)$

- The gradient of the $\overline{\text{PBE}}$ w.r.t $\mathbf{w}$ is:

$$\nabla\overline{\text{PBE}}(\mathbf{w}) = 2\,\nabla\big[\mathbf{X}^T\mathbf{D}\,\bar{\delta}_{\mathbf{w}}\big]^T(\mathbf{X}^T\mathbf{D}\mathbf{X})^{-1}\big(\mathbf{X}^T\mathbf{D}\,\bar{\delta}_{\mathbf{w}}\big)$$

- Let's turn this into an SGD method via converting the 3 factors above into **expectations** under this distribution:

$$\mathbf{X}^T\mathbf{D}\,\bar{\delta}_{\mathbf{w}} = \sum_s \mu(s)\,\mathbf{x}(s)\,\bar{\delta}_{\mathbf{w}}(s) = \mathbb{E}[\rho_t\,\delta_t\,\mathbf{x}_t]$$

$$\begin{aligned}\nabla\big[\mathbf{X}^T\mathbf{D}\,\bar{\delta}_{\mathbf{w}}\big] &= \nabla\mathbb{E}[\rho_t\,\delta_t\,\mathbf{x}_t] \\ &= \mathbb{E}\big[\rho_t\nabla\delta_t^T\,\mathbf{x}_t^T\big] \\ &= \mathbb{E}\big[\rho_t\nabla(R_{t+1} + \gamma\mathbf{w}^T\mathbf{x}_{t+1} - \mathbf{w}^T\mathbf{x}_t)\,\mathbf{x}_t^T\big] \\ &= \mathbb{E}\big[\rho_t(\gamma\mathbf{x}_{t+1} - \mathbf{x}_t)\,\mathbf{x}_t^T\big]\end{aligned}$$

$$\mathbf{X}^T\mathbf{D}\mathbf{X} = \sum_s \mu(s)\,\mathbf{x}_s\,\mathbf{x}_s^T = \mathbb{E}\big[\mathbf{x}_t\,\mathbf{x}_t^T\big]$$

Substituting these expectations for the three factors into $\nabla\overline{\text{PBE}}$:

$$\nabla\overline{\text{PBE}}(\mathbf{w}) = 2\,\mathbb{E}\big[\rho_t(\gamma\mathbf{x}_{t+1} - \mathbf{x}_t)\,\mathbf{x}_t^T\big]\,\mathbb{E}\big[\mathbf{x}_t\,\mathbf{x}_t^T\big]^{-1}\,\mathbb{E}[\rho_t\,\delta_t\,\mathbf{x}_t]$$

- The 1st and last terms are not independent (**biased gradient estimate**).
- Could estimate all 3 terms separately and combine (**unbiased gradient estimate**) but too computationally expensive.

11.7.1. Gradient-TD

- Estimate and store the product of the last 2 terms of $\nabla\overline{\text{PBE}}(\mathbf{w})$ (product of a $d\times d$ matrix and a d-vector yields a d-vector like $\mathbf{w}$ itself):

$$\mathbf{v} \approx \mathbb{E}\big[\mathbf{x}_t\,\mathbf{x}_t^T\big]^{-1}\,\mathbb{E}[\rho_t\,\delta_t\,\mathbf{x}_t]$$

- In linear supervised learning, this is the solution to a linear least-squares problem for $\rho_t\,\delta_t$ approximation from the features.
- The standard SGD method for incrementally finding $\mathbf{v}$ that minimizes the expected squared error $\big(\mathbf{v}^T\mathbf{x}_t - \rho_t\,\delta_t\big)^2$ is known as the **Least Mean Square (LMS)** rule:

$$\mathbf{v}_{t+1} \doteq \mathbf{v}_t + \beta\,\delta_t(\delta_t - \mathbf{v}_t^T\mathbf{x}_t)\,\mathbf{x}_t$$

where

$\beta > 0 \equiv$ another step-size parameter

$\rho_t \equiv$ importance sampling ratio

$O(d) \equiv$ storage & per-step computational complexity

11.7.2. GTD2

- With $\mathbf{v}_t$, we can update $\mathbf{w}_t$ using SGD:

$$\begin{aligned}
\mathbf{w}_{t+1} &= \mathbf{w}_t - \tfrac{1}{2}\alpha\nabla\overline{\text{PBE}}(\mathbf{w}_t) \\
&= \mathbf{w}_t - \tfrac{1}{2}\alpha\Big(2\,\mathbb{E}[\rho_t(\gamma\mathbf{x}_{t+1} - \mathbf{x}_t)\,\mathbf{x}_t^T]\,\mathbb{E}[\mathbf{x}_t\,\mathbf{x}_t^T]^{-1}\,\mathbb{E}[\rho_t\,\delta_t\,\mathbf{x}_t]\Big) \\
&= \mathbf{w}_t + \alpha\,\mathbb{E}[\rho_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})\,\mathbf{x}_t^T]\,\mathbb{E}[\mathbf{x}_t\,\mathbf{x}_t^T]^{-1}\,\mathbb{E}[\rho_t\,\delta_t\,\mathbf{x}_t] \\
&\approx \mathbf{w}_t + \alpha\,\mathbb{E}[\rho_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})\,\mathbf{x}_t^T]\,\mathbf{v}_t \\
&\approx \mathbf{w}_t + \alpha\rho_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})\,\mathbf{x}_t^T\mathbf{v}_t
\end{aligned}$$

  where $O(d)$ per-step computational complexity of $(\mathbf{x}_t^T\mathbf{v}_t)$ is done first.

11.7.3. TD(0) with Gradient Correction (GTD(0) or TDC)

- Let's look at another analytical algorithm called TD(0) with gradient correction, **TDC**:

$$\begin{aligned}
\mathbf{w}_{t+1} &= \mathbf{w}_t + \alpha\,\mathbb{E}[\rho_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})\,\mathbf{x}_t^T]\,\mathbb{E}[\mathbf{x}_t\,\mathbf{x}_t^T]^{-1}\,\mathbb{E}[\rho_t\,\delta_t\,\mathbf{x}_t] \\
&= \mathbf{w}_t + \alpha(\mathbb{E}[\rho_t\,\mathbf{x}_t\,\mathbf{x}_t] - \gamma\,\mathbb{E}[\rho_t\,\mathbf{x}_{t+1}\,\mathbf{x}_t^T])\,\mathbb{E}[\mathbf{x}_t\,\mathbf{x}_t^T]^{-1}\,\mathbb{E}[\rho_t\,\delta_t\,\mathbf{x}_t] \\
&= \mathbf{w}_t + \alpha(\mathbb{E}[\mathbf{x}_t\,\mathbf{x}_t] - \gamma\,\mathbb{E}[\rho_t\,\mathbf{x}_{t+1}\,\mathbf{x}_t^T])\,\mathbb{E}[\mathbf{x}_t\,\mathbf{x}_t^T]^{-1}\,\mathbb{E}[\rho_t\,\delta_t\,\mathbf{x}_t] \\
&= \mathbf{w}_t + \alpha\Big(\mathbb{E}[\mathbf{x}_t\,\rho_t\,\delta_t] - \gamma\,\mathbb{E}[\rho_t\,\mathbf{x}_{t+1}\,\mathbf{x}_t^T]\,\mathbb{E}[\mathbf{x}_t\,\mathbf{x}_t^T]^{-1}\,\mathbb{E}[\rho_t\,\delta_t\,\mathbf{x}_t]\Big) \\
&\approx \mathbf{w}_t + \alpha(\mathbb{E}[\mathbf{x}_t\,\rho_t\,\delta_t] - \gamma\,\mathbb{E}[\rho_t\,\mathbf{x}_{t+1}\,\mathbf{x}_t^T]\,\mathbf{v}_t) \\
&\approx \mathbf{w}_t + \alpha\rho_t(\delta_t\,\mathbf{x}_t - \gamma\mathbf{x}_{t+1}\,\mathbf{x}_t^T\mathbf{v}_t)
\end{aligned}$$

  with $O(d)$ complexity if the final product $(\mathbf{x}_t^T\mathbf{v}_t)$ is done first.

11.7.4. Takeaways

- GTD2 and TDC both involve 2 learning processes: a primary one for $\mathbf{w}$ and a secondary one for $\mathbf{v}$.
- Asymmetrical dependence ($\mathbf{w}$ depends on $\mathbf{v}$ but $\mathbf{v}$ does not depend on $\mathbf{w}$) is referred to as a **cascade**.
- Gradient-TD methods are the most well-understood and widely used stable off-policy methods.
- Extensions of GTD methods include to:
  1. Action values and control: **GQ [Maei et al., 2010]**
  2. Eligibility traces: **GTD(λ), GQ(λ) [Maei, 2011; Maei & Sutton, 2010]**
  3. Nonlinear function approximation **[Maei et al., 2009]**
- Hybrid algorithms include:
  1. Midway between semi-gradient TD and gradient TD **[Hackman, 2012; White & White, 2016]**
  2. GTD + proximal methods & control variates **[Mahadevan et al., 2014; Du et al., 2017]**

11.8. Emphatic-TD Methods

- Let's explore a major strategy for obtaining a cheap and efficient off-policy learning method with function approximation.
- Recall that linear semi-gradient TD methods are stable when trained under the on-policy distribution .
- The match between the on-policy state distribution μ_π and the state-transition probabilities $p(s'|s,a)$ under the target policy does not exist in off-policy learning.
- **Mismatch Fix**:
  - Re-weight the states, emphasizing some and de-emphasizing others, so as to return the distribution of the updates to the on-policy distribution.
  - Then there would be a match, and convergence and stability would be achieved. This is the idea of **Emphatic-TD methods**.
- The **one-step Emphatic-TD algorithm** for learning episodic state values is defined by:

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

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha M_t \rho_t \, \delta_t \nabla \hat{v}(S_t, \mathbf{w}_t)$$

$$M_t = \gamma \rho_{t-1} M_{t-1} + \mathcal{I}_t$$

where

$\mathcal{I}_t \equiv$ the interest

$M_t \equiv$ the emphasis $\quad (M_{-1} = 0)$

- Applying Emphatic TD to Baird's counterexample yields very high variance results (impossible to get consistent results in experiments).
- We focus on how we reduce the variance in all these algorithms in the next section.

11.9. Reducing Variance

- Off-policy learning is inherently of greater variance than on-policy learning.
- The raison d'être of off-policy learning is to enable generalization to the vast number of related-but-not-identical policies.
- Why is variance control critical in off-policy learning based on importance sampling?
  - Recall importance sampling involves products of policy ratios:

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

  - The policy ratios always have an expected value of 1, but their actual values might be high or as low as 0:

$$\mathbb{E}\left[\frac{\pi(A_k|S_k)}{b(A_k|S_k)}\right] \doteq \sum_a b(a|S_k) \frac{\pi(a|S_k)}{b(a|S_k)} = \sum_a \pi(a|S_k) = 1$$

  - Successive ratios are uncorrelated, so their products are always 1 in expected value, but they can be of high variance.
  - These ratios multiply the step size in SGD methods, so their high variance is problematic for SGD because of the occasional huge steps.
- How can we alleviate the effects of high variance via small step-size settings that are enough to ensure the expected step taken by SGD is small? Some approaches:
  - Momentum
  - Polyak-Ruppert averaging
  - Methods for adaptively setting separate step sizes for different components of the parameter vector
  - "Importance weight aware" updates of **Karampatziakis & Langford (2015)**
  - Weighted importance sampling, which is well-behaved with lower variance updates than ordinary importance sampling, but adapting it to function approximation is challenging [**Mahmood & Sutton, 2015**]
  - Tree backup algorithm (off-policy, without importance sampling)
  - Allow the target policy π to be determined partly by the behavior policy b to limit creating large importance sampling ratios

11.10. Summary

- While tabular off-policy algorithms (Q-learning, Expected Sarsa, tree backup) behave reliably, adding even linear function approximation introduces severe stability hazards.

- Two separate obstacles arise: correcting update targets for the mismatch between behavior policy b and target policy π, and stabilizing updates when the sampled state distribution no longer matches the on-policy distribution.
- Combining **function approximation, off-policy learning, and bootstrapping** forms the **deadly triad**, where parameter weights can diverge even on tiny MDPs.
- Minimizing Bellman error ($\overline{\text{BE}}$) directly via SGD fails because $\overline{\text{BE}}$ is not identifiable from observable trajectories alone.
- **Gradient-TD** methods instead minimize the learnable projected Bellman error ($\overline{\text{PBE}}$) in $O(d)$ time per step, while **Emphatic-TD** re-weights updates to recover an on-policy-like state distribution.
- High variance from products of importance-sampling ratios remains a practical bottleneck, mitigated via conservative step sizes, variance-aware optimizers, control variates, or tree-backup style updates.

12. Eligibility Traces

- Eligibility traces are one of the basic mechanisms of RL that unify and generalize TD and Monte Carlo (MC) methods.
- TD methods augmented with eligibility traces produce a family of methods spanning a range from MC methods at one end ($\lambda = 1$) to one-step TD (TD(0)) methods at the other end ($\lambda = 0$).
- With eligibility traces, MC methods can be implemented online and on continuing problems.
- n-step methods also unify TD and MC methods but are not as elegant algorithmically as eligibility traces (ET).
- The eligibility traces algorithm entails:
  - First, we have a short-term memory vector, the **eligibility trace** $\mathbf{z}_t \in \mathbb{R}^d$, that parallels the long-term weight vector $\mathbf{w}_t \in \mathbb{R}^d$.
  - Then when a component of $\mathbf{w}_t$ participates in producing an estimated value, the corresponding component of $\mathbf{z}_t$ is bumped up and then begins to fade away.
  - Learning occurs in that component of $\mathbf{w}_t$ if a non-zero TD error occurs before the trace falls back to zero (fades away).
  - The trace-decay parameter $\lambda \in [0, 1)$ determines the rate at which the trace falls.
- Advantages of ET over n-step methods:
  - Requires only a single trace vector $\mathbf{z}_t$ rather than storing the last n feature vectors.
  - Learning occurs continually and uniformly in time rather than being delayed and playing "catch up" at episode end. This leads to immediate behavior effect from learning rather than being delayed.
  - ET is a **backward view** algorithm, unlike n-step methods that are **forward view** algorithms, which is less complex to implement.
- **Forward view** algorithms are based on looking forward from the updated state, and the updated state depends on all the future rewards.
- **Backward view** algorithms use the current TD error, looking backward to recently visited states, to achieve nearly the same updates as forward view.
- We start with ideas for state values and prediction, then extend them to action values and control, then do on-policy, then extend to off-policy learning. The field of focus is linear function approximation (covers tabular and state aggregation cases).

12.1. The λ-return

- Recall in Chapter 7 we defined an n-step return as the sum of the first n rewards plus the estimated value of the state reached in n steps, each appropriately discounted:

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n \hat{v}(S_{t+n}, \mathbf{w}_{t+n-1}), \quad 0 \leq t \leq T - n$$

- A valid update can be done not just towards any n-step return, but also towards any average of n-step returns.
  - E.g. average the 2-step and 4-step return: $\frac{1}{2} G_{t:t+2} + \frac{1}{2} G_{t:t+4}$
- Any set of n-step returns can be averaged, even an infinite set, as long as the weights on the component returns are positive and sum to 1.
- What if instead of using one n-step return, we use a weighted average of all n-step returns? This leads to averaging which produces a substantial new range of algorithms. E.g.,
  1. Averaging one-step and infinite-step returns to interrelate TD and MC methods.
  2. Averaging experience-based updates with Dynamic Programming (DP) updates to obtain a single combination of experience-based and model-based methods.
- An update that averages simpler component updates is called a **compound update** or the **λ-return**.
- The TD(λ) algorithm is one way of averaging n-step updates, each weighted proportionally by λ^{n-1} (where $\lambda \in [0, 1]$) and normalized by a factor of $(1 - \lambda)$ to ensure that the weights sum to 1.

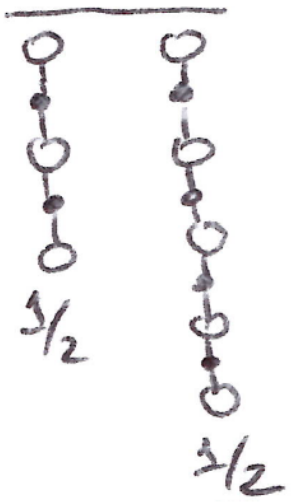

Figure 12.1. **Backup Diagram for Compound Update.** A compound target combining equal halves of a 2-step return and a 4-step return.

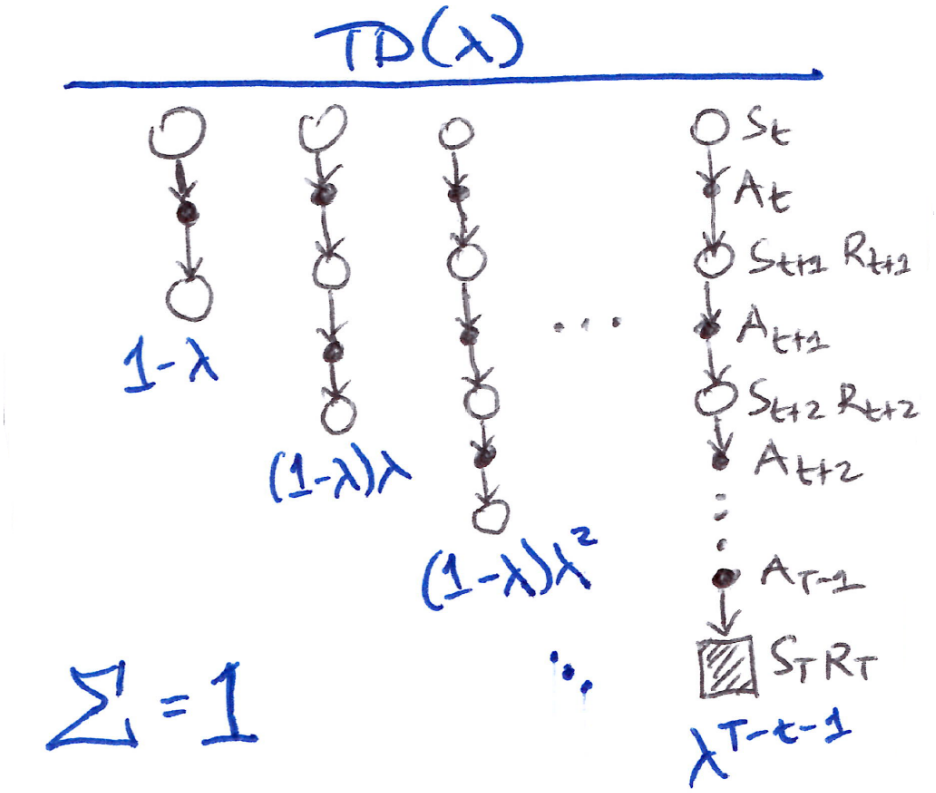

Figure 12.2. **Backup Diagram for TD(λ).** Blending every n-step return with geometric weights recovers **TD(0)** when $\lambda = 0$ and full **MC** returns when $\lambda = 1$.

- Essentially, λ-return, G_t^λ, combines all n-step returns $G_{t:t+n}$ in a weighted average manner, $(1-\lambda)\lambda^{n-1}$, and is defined in its state-based form by:

$$\boxed{G_t^\lambda \doteq (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}G_{t:t+n}}$$

12.1.1. TD(λ) Weighting

- The TD(λ) weighting function diagram illustrates the weighting on the sequence of n-step returns in the λ-return:
  - 1-step return gets the largest weight, $1-\lambda$
  - 2-step return gets the next (2nd) largest weight, $(1-\lambda)\lambda$
  - 3-step return gets the 3rd largest weight, $(1-\lambda)\lambda^2$
  - n-step return gets the n-th largest (smallest) weight, $(1-\lambda)\lambda^{n-1}$
  - The weight fades by λ with each additional step.
  - After a terminal state has been reached, all subsequent n-step returns are equal to the conventional return G_t.
  - So essentially, we can decompose G_t^λ based on the TD(λ) weighting function diagram into the main sum and post-termination terms:

$$G_t^\lambda = \underbrace{(1-\lambda)\textstyle\sum_{n=1}^{T-t-1}\lambda^{n-1}G_{t:t+n}}_{\text{pre-termination}} + \underbrace{\lambda^{T-t-1}G_t}_{\text{post-termination}}$$

- So now we can see the impact of λ more clearly:

$$\text{if } \lambda = 1: \quad G_t^\lambda = G_t \qquad \text{(MC)}$$

$$\text{if } \lambda = 0: \quad G_t^\lambda = \left\{ \begin{array}{ll} \sum_{n=1}^{T-t-1} G_{t:t+n} & \text{for } n=1 \\ 0 & \text{for } n>1 \end{array} \right\} = G_{t:t+1} \qquad \text{(TD(0))}$$

- Our first learning algorithm based on the λ-return is the **off-line λ-return algorithm**, which waits until the end of an episode to make updates. Its semi-gradient, λ-return, target update for $t = 0, 1, 2, \ldots, T-1$ is:

$$\boxed{\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\left[G_t^\lambda - \hat{v}(S_t, \mathbf{w}_t)\right]\nabla\hat{v}(S_t, \mathbf{w}_t)}$$

- The λ-return allows us to smoothly move between MC and TD(0) methods, comparable to n-step returning.

12.1.2. Forward View

- This approach is called the theoretical, forward view of a learning algorithm:
  - Update value function towards the λ-return.
  - Look forward in time to all the future rewards to compute G_t^λ.
  - Like MC, can only be computed from complete return.

12.2. TD(λ)

- TD(λ) was the first algorithm that showed a formal relationship between a forward view and backward view using eligibility traces.
- TD(λ) improves over the off-line λ-return algorithm in 3 ways:
  - Updates the weight vector on every step of an episode and not just the end.
  - Equal distributions in time of its computations rather than at an episode's end.
  - Can be applied to continuing problems and not just episodic problems.
- Let's focus on the **semi-gradient version of TD(λ)** with function approximation:
  - The **eligibility trace $\mathbf{z}_t$** has the same number of components as $\mathbf{w}_t$.

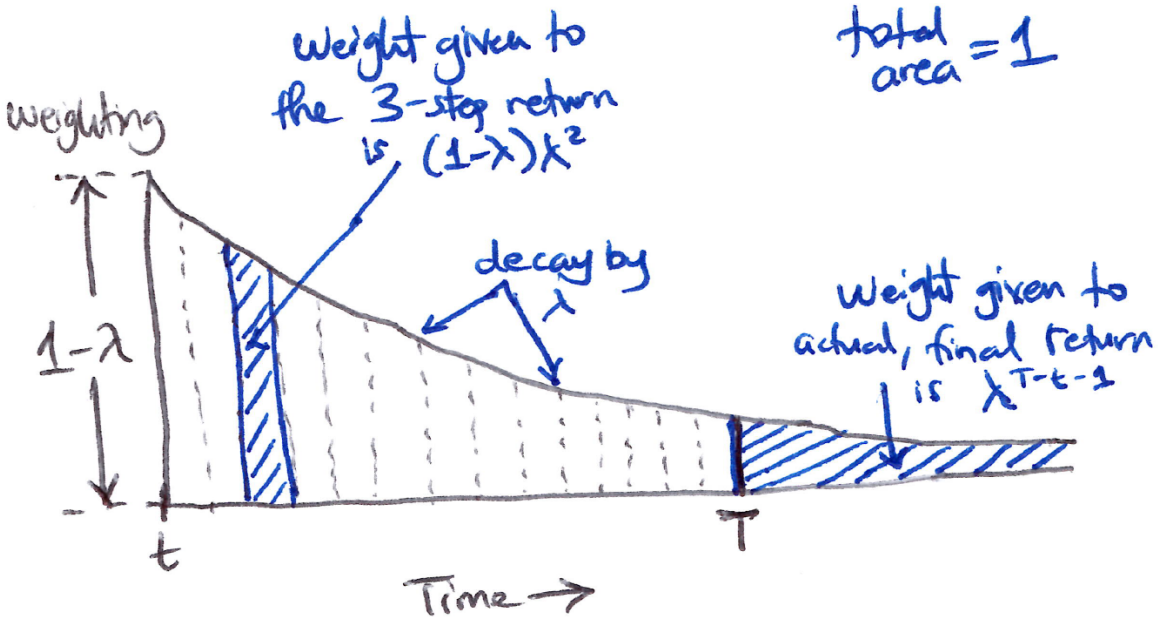

Figure 12.3. **TD(λ) Weighting.** Geometric decay profile showing how much weight each n-step return contributes to G_t^λ.

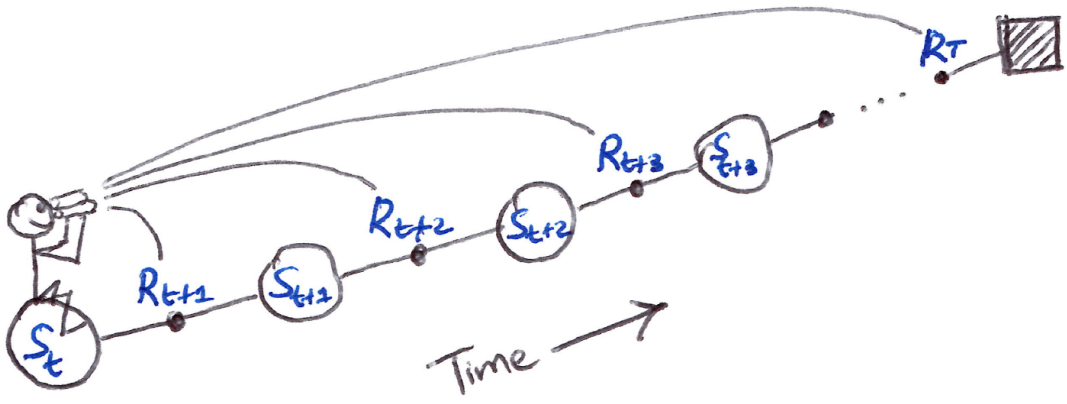

Figure 12.4. **Forward View TD(λ).** Updating a visited state conceptually looks ahead along the trajectory to aggregate upcoming rewards and values.

– $\mathbf{z}$ is initialized to $\mathbf{0}$, incremented on each time step by the value gradient, and then fades away by $\gamma\lambda$:

$$\begin{aligned}\mathbf{z}_{-1} &\doteq \mathbf{0}\\ \mathbf{z}_t &\doteq \gamma\lambda\mathbf{z}_{t-1} + \nabla\hat{v}(S_t, \mathbf{w}_t), \quad 0 \leq t \leq T\end{aligned}$$

where

$\lambda \equiv$ trace decay parameter

$\gamma \equiv$ discount rate

- The eligibility trace keeps track of which $\mathbf{w}_t$ components have contributed, positively or negatively, to recent state valuations.
- This is the **recency heuristic** used for **credit assignment**, where more credit is assigned to the most recent states. **Recent** is defined in terms of $\gamma\lambda$.
- The TD error for state-value prediction is:

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

and the weight vector update in TD(λ) is proportional to the scalar TD error and the vector eligibility trace:

$$\boxed{\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\,\delta_t\,\mathbf{z}_t}$$

12.2.1. Backward View

- Forward view provides theory but backward view provides mechanism (practical) where we update online, every step, from incomplete sequences.
- Keep an eligibility trace for every state s.
- Update value $V(s)$ for every state s in proportion to TD-error δ_t and eligibility trace $\mathbf{z}_t$:

$$\begin{aligned}\delta_t &= R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\\ V(s) &\leftarrow V(s) + \alpha\,\delta_t\,\mathbf{z}_t\end{aligned}$$

- Let's look at the effect of λ to understand the backward view of TD(λ):

$$\begin{aligned}\text{if } \lambda = 0: &\quad \mathbf{z}_t = \nabla\hat{v}(S_t, \mathbf{w}_t)\\ &\quad \mathbf{w}_{t+1} = \mathbf{w}_t + \alpha\,\delta_t\nabla\hat{v}(S_t, \mathbf{w}_t) \quad \longrightarrow \quad \text{TD(0)}\\ \text{if } 0 < \lambda < 1: &\quad \text{earlier states are given less credit for the TD error}\\ \text{if } \lambda = 1: &\quad \mathbf{z}_t = \gamma\mathbf{z}_{t-1} + \nabla\hat{v}(S_t, \mathbf{w}_t) \quad \longrightarrow \quad \text{credit for earlier states falls by } \gamma \text{ per step}\\ \text{if } \lambda = 1,\ \gamma = 1: &\quad \mathbf{z}_t = \mathbf{z}_{t-1} + \nabla\hat{v}(S_t, \mathbf{w}_t) \quad \longrightarrow \quad \text{MC-like behavior (no time decay for ET)}\\ \text{if } \lambda = 1: &\quad \text{we get TD(1)}\end{aligned}$$

- In summary, $\lambda = 1$ yields TD(1).
- TD(1) implements MC algorithms in a more general way and for a wider range of applicability:
  – Not limited to episodic tasks, can be applied to discounted continuing tasks.
  – Can be performed **incrementally and online**.
  – Learns **immediately** and alters behavior during an episode if something good or bad happens, for control methods.
- Linear TD(λ) converges in the on-policy case if the step-size parameter α is reduced over time according to stochastic approximation theory conditions.
- The convergence of linear TD(λ) is not to the minimum-error weight vector but to a nearby weight vector that depends on λ.
- The bound on solution quality generalized for any λ, for the continuing, discounted case is:

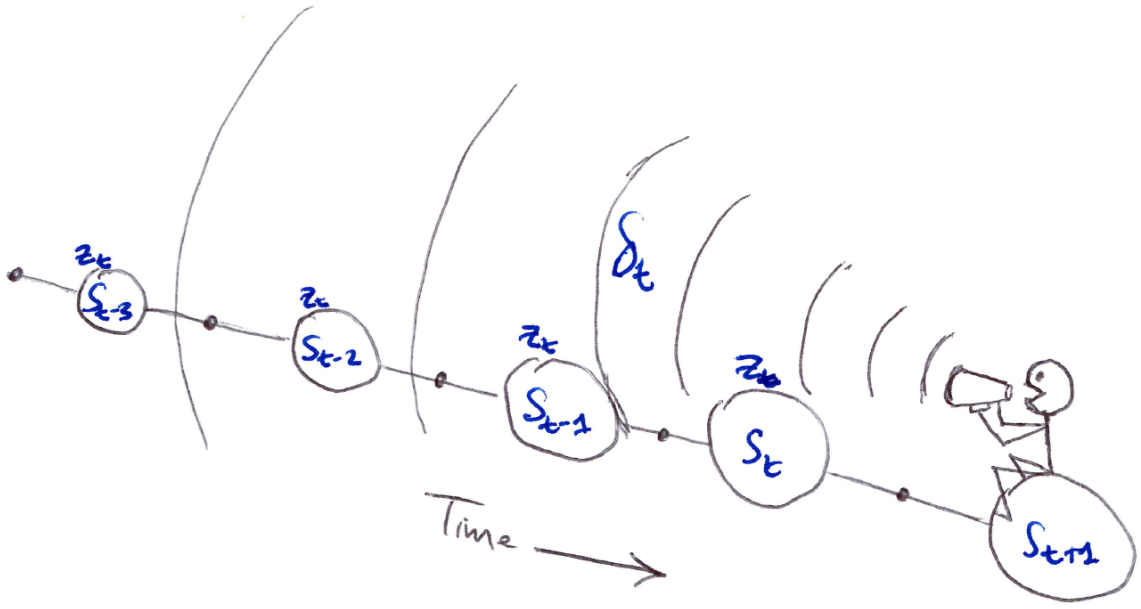

Figure 12.5. **Backward View of TD(λ).** Online updates broadcast the current step's TD error backward to recently visited states in proportion to their decaying eligibility traces.

$$\overline{\text{VE}}(\mathbf{w}_\infty) \leq \frac{1-\gamma\lambda}{1-\gamma} \min_{\mathbf{w}} \overline{\text{VE}}(\mathbf{w})$$

- That is, the asymptotic error is no more than $\frac{1-\gamma\lambda}{1-\gamma}$ times the smallest possible error for TD(λ):

$$\text{as } \lambda \to 1: \quad \overline{\text{VE}}(\mathbf{w}_\infty) \to \min_{\mathbf{w}} \overline{\text{VE}}(\mathbf{w})$$

$$\text{as } \lambda \to 0: \quad \overline{\text{VE}}(\mathbf{w}_\infty) \to \frac{1}{1-\gamma} \min_{\mathbf{w}} \overline{\text{VE}}(\mathbf{w}) = \overline{\text{VE}}(\mathbf{w}_{\text{TD}}) \quad \text{(TD(0))}$$

- However, $\lambda = 1$ is often the poorest choice.

12.3. n-step Truncated λ-return Methods

- The off-line λ-return is of limited use because the λ-return is not known until the episode ends.
- The off-line λ-return approximation is to truncate the sequence after a **fixed** number of steps.
- Hence, a natural approximation is to truncate the sequence where λ-return cannot be calculated for an arbitrarily large n. This handles the continuing case.
- The truncated λ-return for time t, given data only up to some later horizon h, is:

$$G^\lambda_{t:h} \doteq (1-\lambda) \sum_{n=1}^{h-t-1} \lambda^{n-1} G_{t:t+n} + \lambda^{h-t-1} G_{t:h}, \quad 0 \leq t \leq h \leq T$$

where

$h \equiv$ horizon (plays same role as time of termination T)

- Here the **residual weighting** is given to the longest available n-step return $G_{t:h}$.
- The truncated λ-return gives rise to a family of n-step λ-return algorithms, known in the state-value case as **Truncated TD(λ)** or **TTD(λ)**.
- TTD(λ) is defined for $0 \leq t < T$ by:

$$\boxed{\mathbf{w}_{t+n} \doteq \mathbf{w}_{t+n-1} + \alpha[G^{\lambda}_{t:t+n} - \hat{v}(S_t, \mathbf{w}_{t+n-1})] \nabla \hat{v}(S_t, \mathbf{w}_{t+n-1})}$$

- Efficient implementation of TTD(λ) relies on the k-step λ-return:

$$\boxed{G^{\lambda}_{t:t+k} = \hat{v}(S_t, \mathbf{w}_{t-1}) + \sum_{i=t}^{t+k-1} (\gamma\lambda)^{i-t} \delta'_i}$$

where

$$\delta'_i \equiv R_{i+1} + \gamma \hat{v}(S_{i+1}, \mathbf{w}_{t-1}) - \hat{v}(S_i, \mathbf{w}_{t-1})$$

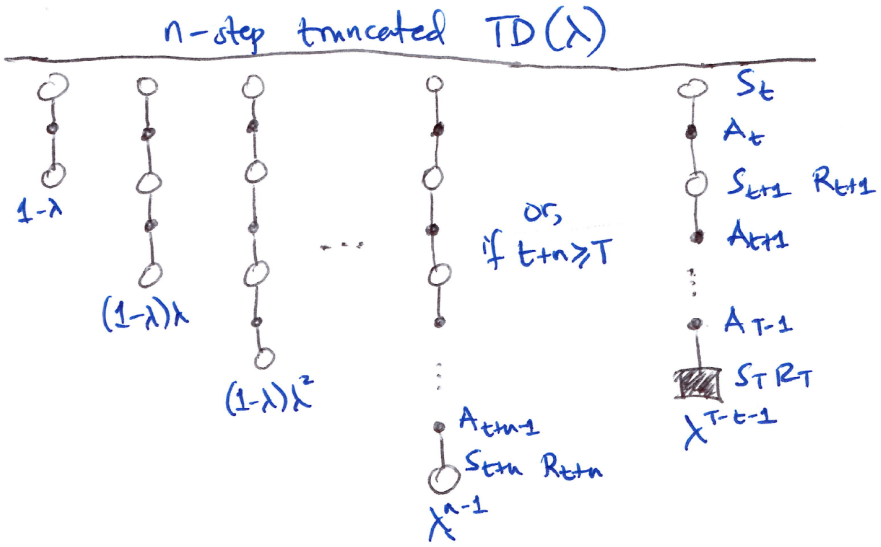

Figure 12.6. **Backup Diagram for Truncated TD(λ).** Cutting the λ-return off at a finite horizon h produces the **n-step TTD(λ)** family of algorithms.

12.4. Redoing Updates: Online λ-return Algorithm

- How do we choose the truncation parameter n in TTD(λ)?
- It involves a tradeoff:
  - n should be large so that TTD(λ) closely approximates off-line λ-return, but
  - n should also be small so that the updates can be made sooner and can influence behavior sooner.
- In principle, we can achieve both cases via the **online λ-return algorithm**, but at the cost of computational complexity.
- Essentially at each time step, we go back and redo all the updates since the beginning of the episode as we gather new increment of data:
  - The new updates are better than the old ones because now they account for the time step's new data.
  - Basically this conceptual algorithm involves multiple passes over the episode, one at each horizon, each generating a different sequence of weight vectors.

- Let's distinguish between the weight vectors computed at the different horizons by seeing the first 3 sequences:

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

where

$\mathbf{w}_t^h \equiv$ weights used to generate the value at time t in the sequence up to horizon h

$\mathbf{w}_0^h \equiv$ 1st weight vector in each sequence that is inherited from the previous episode

$\mathbf{w}_n^h \equiv$ last weight vector in each sequence or the ultimate weight-vector sequence

- The general form of the **online λ-return update** for $0 \leq t < h \leq T$ is:

$$\boxed{\mathbf{w}_{t+1}^h \doteq \mathbf{w}_t^h + \alpha[G_{t:h}^\lambda - \hat{v}(S_t, \mathbf{w}_t^h)]\, \nabla \hat{v}(S_t, \mathbf{w}_t^h)}$$

$$\mathbf{w}_t \doteq \mathbf{w}_t^t$$

12.5. True Online TD(λ)

- The original ideal online λ-return algorithm shown in Section 12.4 is very complex so we use online TD(λ) to approximate it.
- We'll use eligibility trace to invert the forward view, online λ-return to an efficient backward view algorithm. This is called the **True Online TD(λ)**.
- It uses a simple strategy 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 diagonals of online λ-return weight matrix).

$$
\begin{bmatrix}
\mathbf{w}_0^0 & & & & & \\
\mathbf{w}_0^1 & \mathbf{w}_1^1 & & & & \\
\mathbf{w}_0^2 & \mathbf{w}_1^2 & \mathbf{w}_2^2 & & & \\
\mathbf{w}_0^3 & \mathbf{w}_1^3 & \mathbf{w}_2^3 & \mathbf{w}_3^3 & & \\
\vdots & \vdots & \vdots & \vdots & \ddots & \\
\mathbf{w}_0^T & \mathbf{w}_1^T & \mathbf{w}_2^T & \mathbf{w}_3^T & \cdots & \mathbf{w}_T^T
\end{bmatrix}
\longrightarrow
\begin{bmatrix}
\mathbf{w}_0^0 & & & & & \\
& \mathbf{w}_1^1 & & & & \\
& & \mathbf{w}_2^2 & & & \\
& & & \mathbf{w}_3^3 & & \\
& & & & \ddots & \\
& & & & & \mathbf{w}_T^T
\end{bmatrix}
$$

Online λ-return True Online TD(λ)

- For the linear case in which $\hat{v}(s, \mathbf{w}) = \mathbf{w}^T \mathbf{x}(s)$, the true online TD($\lambda$) algorithm is:

$$\boxed{\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\, \delta_t\, \mathbf{z}_t + \alpha(\mathbf{w}_t^T \mathbf{x}_t - \mathbf{w}_{t-1}^T \mathbf{x}_t)(\mathbf{z}_t - \mathbf{x}_t)}$$

where

$$
\begin{aligned}
\mathbf{w}_t &\doteq \mathbf{w}_t^t \\
\mathbf{x}_t &\doteq \mathbf{x}(S_t) \\
\mathbf{z}_t &\doteq \gamma\lambda \mathbf{z}_{t-1} + (1 - \alpha\gamma\lambda\, \mathbf{z}_{t-1}^T \mathbf{x}_t)\, \mathbf{x}_t
\end{aligned}
$$

- The per-step computational complexity of true online TD(λ) is the same as TD(λ), $O(d)$.
- $\mathbf{z}_t$ used in true online TD(λ) is called a **dutch trace**, unlike that of TD(λ) which is called an **accumulating trace**.

- Earlier work used a 3rd kind of trace called the **replacing trace**, defined only for the tabular case or for binary feature vectors (tile coding). It is defined:

$$\tilde{z}_{i,t} \doteq \left\{ \begin{array}{ll} 1 & \text{if } x_{i,t} = 1 \\ \gamma\lambda\, z_{i,t-1} & \text{otherwise} \end{array} \right\}$$

- Nowadays, dutch traces usually perform better than replacing traces and have a clearer theoretical basis.
- Accumulating traces remain of interest for nonlinear function approximations where dutch traces are unavailable.

12.6. Dutch Traces in Monte Carlo Learning

- Eligibility traces have nothing to do with TD learning despite their close historical association.
- Eligibility traces arise even in Monte Carlo learning.
- Using dutch traces, we can invert the forward view MC algorithm to an equivalent, yet computationally cheaper backward view algorithm.
- This is the only equivalence of forward and backward view that is explicitly demonstrated in this book.
- The linear, gradient MC prediction algorithm makes the following sequence of updates, one for each time step of the episode:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\big[G - \mathbf{w}_t^T\mathbf{x}_t\big]\,\mathbf{x}_t, \quad 0 \le t < T$$

- For simplicity, assume that the return G is a single reward at the end of the episode (hence no subscript by time) and that there is no discounting.
- This is known as the **Least Mean Square (LMS)** rule.
- We introduce an additional vector memory, the **eligibility trace**, that keeps in it a summary of all the feature vectors seen so far. The overall update will be the same as the MC updates' sequence shown above and is:

$$\begin{aligned} \mathbf{w}_T &= \mathbf{w}_{T-1} + \alpha(G - \mathbf{w}_{T-1}^T\mathbf{x}_{T-1})\,\mathbf{x}_{T-1} \\ &= \mathbf{w}_{T-1} + \alpha\mathbf{x}_{T-1}(-\mathbf{x}_{T-1}^T\mathbf{w}_{T-1}) + \alpha G\mathbf{x}_{T-1} \\ &= (\mathbf{I} - \alpha\mathbf{x}_{T-1}\mathbf{x}_{T-1}^T)\,\mathbf{w}_{T-1} + \alpha G\mathbf{x}_{T-1} \\ &= \mathbf{F}_{T-1}\,\mathbf{w}_{T-1} + \alpha G\mathbf{x}_{T-1} \end{aligned}$$

where

$$\mathbf{F}_t \doteq \mathbf{I} - \alpha\mathbf{x}_t\mathbf{x}_t^T \equiv \text{a forgetting or fading matrix}$$

$$\therefore \quad \mathbf{w}_{T-1} = \mathbf{F}_{T-2}\,\mathbf{w}_{T-2} + \alpha G\mathbf{x}_{T-2}$$

Now recursing:

$$\begin{aligned} \mathbf{w}_T &= \mathbf{F}_{T-1}\,\mathbf{w}_{T-1} + \alpha G\mathbf{x}_{T-1} \\ &= \mathbf{F}_{T-1}(\mathbf{F}_{T-2}\,\mathbf{w}_{T-2} + \alpha G\mathbf{x}_{T-2}) + \alpha G\mathbf{x}_{T-1} \\ &= \mathbf{F}_{T-1}\mathbf{F}_{T-2}\,\mathbf{w}_{T-2} + \alpha G(\mathbf{F}_{T-1}\mathbf{x}_{T-2} + \mathbf{x}_{T-1}) \\ &= \mathbf{F}_{T-1}\mathbf{F}_{T-2}(\mathbf{F}_{T-3}\,\mathbf{w}_{T-3} + \alpha G\,\mathbf{x}_{T-3}) + \alpha G(\mathbf{F}_{T-1}\,\mathbf{x}_{T-2} + \mathbf{x}_{T-1}) \\ &= \mathbf{F}_{T-1}\mathbf{F}_{T-2}\mathbf{F}_{T-3}\,\mathbf{w}_{T-3} + \alpha G(\mathbf{F}_{T-1}\mathbf{F}_{T-2}\,\mathbf{x}_{T-3} + \mathbf{F}_{T-1}\,\mathbf{x}_{T-2} + \mathbf{x}_{T-1}) \\ &\;\;\vdots \\ &= \underbrace{\mathbf{F}_{T-1}\mathbf{F}_{T-2}\cdots\mathbf{F}_0\,\mathbf{w}_0}_{\mathbf{a}_{T-1}} + \alpha G\,\underbrace{\sum_{k=0}^{T-1}\mathbf{F}_{T-1}\mathbf{F}_{T-2}\cdots\mathbf{F}_{k+1}\,\mathbf{x}_k}_{\mathbf{z}_{T-1}} \\ &= \mathbf{a}_{T-1} + \alpha G\,\mathbf{z}_{T-1} \end{aligned}$$

where

$\mathbf{a}_{T-1}$ & $\mathbf{z}_{T-1} \equiv$ values at time $T-1$ of 2 auxiliary memory vectors that can be updated incrementally w/o knowledge of G and with $O(d)$ complexity per time step

$\mathbf{z}_t \equiv$ dutch-style eligibility trace, initialized to $\mathbf{z}_0 = \mathbf{x}_0$

- The $\mathbf{z}_t$ vector is in fact a dutch-style eligibility trace, initialized to $\mathbf{z}_0 = \mathbf{x}_0$, that can be updated according to:

$$\begin{aligned}
\mathbf{z}_t &= \sum_{k=0}^{t} \mathbf{F}_t \mathbf{F}_{t-1} \cdots \mathbf{F}_{k+1} \mathbf{x}_k, \quad 1 \leq t < T \\
&= \sum_{k=0}^{t-1} \mathbf{F}_t \mathbf{F}_{t-1} \cdots \mathbf{F}_{k+1} \mathbf{x}_k + \mathbf{x}_t \\
&= \mathbf{F}_t \sum_{k=0}^{t-1} \mathbf{F}_{t-1} \mathbf{F}_{t-2} \cdots \mathbf{F}_{k+1} \mathbf{x}_k + \mathbf{x}_t \\
&= \mathbf{F}_t \, \mathbf{z}_{t-1} + \mathbf{x}_t \\
&= (\mathbf{I} - \alpha \mathbf{x}_t \mathbf{x}_t^T) \, \mathbf{z}_{t-1} + \mathbf{x}_t \\
&= \mathbf{z}_{t-1} - \alpha (\mathbf{z}_{t-1}^T \mathbf{x}_t) \, \mathbf{x}_t + \mathbf{x}_t \\
&\boxed{= \mathbf{z}_{t-1} + (1 - \alpha \, \mathbf{z}_{t-1}^T \mathbf{x}_t) \, \mathbf{x}_t}
\end{aligned}$$

which is the dutch trace for $\gamma\lambda = 1$.

- The $\mathbf{a}_t$ auxiliary vector is initialized to $\mathbf{a}_0 = \mathbf{w}_0$ and then updated according to:

$$\begin{aligned}
\mathbf{a}_t &= \mathbf{F}_t \mathbf{F}_{t-1} \cdots \mathbf{F}_0 \, \mathbf{w}_0, \quad 1 \leq t < T \\
&= \mathbf{F}_t \, \mathbf{a}_{t-1} \\
&= \mathbf{a}_{t-1} - \alpha \mathbf{x}_t \mathbf{x}_t^T \mathbf{a}_{t-1} \\
&\boxed{= \mathbf{a}_{t-1} (\mathbf{I} - \alpha \mathbf{x}_t \mathbf{x}_t^T)}
\end{aligned}$$

12.6.1. Takeaways

- The auxiliary vectors, $\mathbf{a}_t$ and $\mathbf{z}_t$, are updated on each time step $t < T$ and then, at time T when G is observed, they are used to compute:

$$\boxed{\mathbf{w}_T = \mathbf{a}_{T-1} + \alpha G \, \mathbf{z}_{T-1}}$$

- The time and memory complexity per step is $O(d)$.
- This is surprising and intriguing since ET is working in a non-TD setting (ET arise where long-term predictions are needed to be learned efficiently).

12.7. Sarsa(λ)

- Now let's extend eligibility traces to action-value methods.
- First, let's recall the action-value form of the n**-step** return:

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

$$\text{with} \quad G_{t:t+n} = G_t \quad \text{if } t + n \geq T$$

- With this, for $t = 0, ..., T-1$, let's form the action-value form of the **off-line λ-return** algorithm which uses $\hat{q}$ rather than $\hat{v}$:

$$\boxed{\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha [G_t^\lambda - \hat{q}(S_t, A_t, \mathbf{w}_t)] \, \nabla \hat{q}(S_t, A_t, \mathbf{w}_t)}$$

where

$$G_t^\lambda \doteq G_{t:\infty}^\lambda$$

- For the forward view shown in the figure below, which is similar to TD(λ), the updates are:
  - **1st update:** one full-step lookahead
  - **2nd update:** two-step lookahead
  - **Final update:** complete return

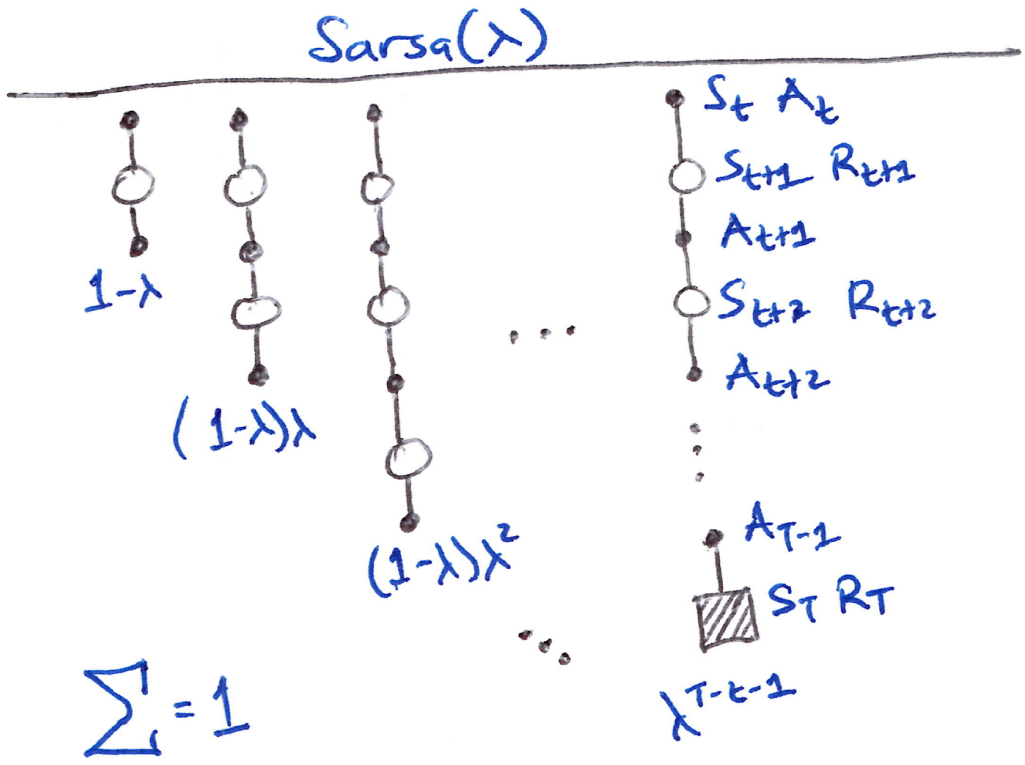

Figure 12.7. **Backup Diagram for Sarsa(λ).** Action-value analogue of TD(λ), geometrically averaging 1-step, 2-step, multi-step, and full-episode state-action returns.

- The weighting of each n-step update in the λ-return is same as in TD(λ) and λ-return.
- The forward view TD for action values, Sarsa(λ), has the same update rule as TD(λ):

$$\boxed{\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\, \delta_t\, \mathbf{z}_t}$$

- The action-value form of the TD error is used:

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

- The action-value form of the eligibility trace:

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

- There exists an action-value version of our ideal TD method, the online λ-return algorithm and its efficient implementation as true online TD(λ):
  - Section 12.4: Everything there holds here except for using the action-value form of the n-step return, $G_{t:t+n}$.

  - Sections 12.5 & 12.6: Everything holds here except for using state-action feature vectors $\mathbf{x}_t = \mathbf{x}(S_t, A_t)$ instead of state feature vectors $\mathbf{x}_t = \mathbf{x}(S_t)$.
  - The resulting efficient backward algorithm obtained from using the eligibility trace to invert the action-value form of the forward view, online λ-return is called the **True Online Sarsa(λ)**.
- There is also a truncated version of Sarsa(λ), called **Forward Sarsa(λ)**, which appears to be a model-free, control method for use in conjunction with multi-layer ANNs.

12.8. Variable λ and γ

- To get the most general forms of the final TD algorithms, it is vital to generalize the degree of bootstrapping and discounting beyond constant parameters to functions dependent on the state and action:

$$\begin{aligned} \lambda_t &\doteq \lambda(S_t, A_t), \qquad & \lambda : S \times A \to [0,1] \\ \gamma_t &\doteq \gamma(S_t), & \gamma : S \to [0,1] \end{aligned}$$

- λ_t is the termination function and is significant because it changes the return G_t, which is now more generally defined as:

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma_{t+1} G_{t+1} \\ &= R_{t+1} + \gamma_{t+1} R_{t+2} + \gamma_{t+1}\gamma_{t+2} R_{t+3} + \gamma_{t+1}\gamma_{t+2}\gamma_{t+3} R_{t+4} + ... \\ &= \sum_{k=t}^{\infty} \left(\prod_{i=t+1}^{k} \gamma_i \right) R_{k+1} \end{aligned}$$

where

$$\prod_{k=t}^{\infty} \gamma_k = 0 \text{ with probability 1 for all } t, \text{ to assure the sums are finite}$$

- This general return G_t definition enables episodic settings to become a single stream of experience, without special terminal states, start distributions, or termination times.
  - A terminal state just becomes a state with $\gamma(s) = 0$ that transitions to the start distribution.
- Generalization to variable bootstrapping yields a new state-based λ-return:

$$\boxed{G_t^{\lambda s} \doteq R_{t+1} + \gamma_{t+1}\big[(1-\lambda_{t+1})\,\hat{v}(S_{t+1}, \mathbf{w}_t) + \lambda_{t+1}\, G_{t+1}^{\lambda s}\big]}$$

- Action-based λ-return is either the **Sarsa form**:

$$\boxed{G_t^{\lambda a} \doteq R_{t+1} + \gamma_{t+1}\big[(1-\lambda_{t+1})\,\hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) + \lambda_{t+1}\, G_{t+1}^{\lambda a}\big]}$$

- or the **Expected Sarsa form**:

$$\boxed{G_t^{\lambda a} \doteq R_{t+1} + \gamma_{t+1}\Big[(1-\lambda_{t+1})\,\bar{V}_t(S_{t+1}) + \lambda_{t+1}\, G_{t+1}^{\bar{\lambda} a}\Big]}$$

where

$$\bar{V}_t(s) \doteq \sum_a \pi(a|s)\,\hat{q}(s, a, \mathbf{w}_t)$$

12.8.1. Superscripts notation for i in $G_t^{\lambda i}$

"s" : bootstraps from state values

"a" : bootstraps from action values

12.9. Off-Policy Traces with Control Variates

- To generalize to off-policy, we need to incorporate importance sampling using eligibility traces.
- Let's focus on the bootstrapping generalization of per-decision importance sampling with control variates (Section 7.4).
- The new state-based λ-return in Section 12.8 generalizes, after the off-policy, control variate, n-step return (ending at horizon h) model to:

$$\boxed{G_t^{\lambda s} \doteq \rho_t(R_{t+1} + \gamma_{t+1}[(1-\lambda_{t+1})\,\hat{v}(S_{t+1},\mathbf{w}_t) + \lambda_{t+1}\,G_{t+1}^{\lambda s}]) + (1-\rho_t)\,\hat{v}(S_t,\mathbf{w}_t)}$$

where

$$\rho_t = \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$$

- The final λ-return can be approximated in terms of sums of the state-based TD error δ_t^s, with the approximation becoming exact if the approximate value function does not change:

$$G_t^{\lambda s} \approx \hat{v}(S_t,\mathbf{w}_t) + \rho_t \sum_{k=t}^{\infty} \delta_k^s \prod_{i=t+1}^{k} \gamma_i\lambda_i\rho_i$$
$$\delta_t^s \doteq R_{t+1} + \gamma_{t+1}\,\hat{v}(S_{t+1},\mathbf{w}_t) - \hat{v}(S_t,\mathbf{w}_t)$$

- The forward view update of the approximate λ-return is:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha[G_t^{\lambda s} - \hat{v}(S_t,\mathbf{w}_t)]\,\nabla\hat{v}(S_t,\mathbf{w}_t)$$
$$\boxed{\approx \mathbf{w}_t + \alpha\rho_t\left(\sum_{k=t}^{\infty}\delta_k^s\prod_{i=t+1}^{k}\gamma_i\lambda_i\rho_i\right)\nabla\hat{v}(S_t,\mathbf{w}_t)}$$

- We're interested in the equivalence (approximately) between the forward-view update summed over time and a backward-view update summed over time. The equivalence is approximate because we ignore changes in the value function.
- The sum of the forward-view update over time is:

$$\begin{aligned}
\sum_{t=0}^{\infty}(\mathbf{w}_{t+1}-\mathbf{w}_t) &\approx \sum_{t=0}^{\infty}\sum_{k=t}^{\infty}\alpha\rho_t\,\delta_k^s\nabla\hat{v}(S_t,\mathbf{w}_t)\prod_{i=t+1}^{k}\gamma_i\lambda_i\rho_i \\
&= \sum_{k=0}^{\infty}\sum_{t=0}^{k}\alpha\rho_t\nabla\hat{v}(S_t,\mathbf{w}_t)\,\delta_k^s\prod_{i=t+1}^{k}\gamma_i\lambda_i\rho_i \\
&\left(\text{using the summation rule: } \sum_{t=x}^{y}\sum_{k=t}^{y} = \sum_{k=x}^{y}\sum_{t=x}^{k}\right) \\
&= \sum_{k=0}^{\infty}\alpha\,\delta_k^s\sum_{t=0}^{k}\rho_t\nabla\hat{v}(S_t,\mathbf{w}_t)\prod_{i=t+1}^{k}\gamma_i\lambda_i\rho_i
\end{aligned}$$

- If the entire expression from the 2nd sum on could be written and updated incrementally as an eligibility trace, then the sum of the forward-view update over time would be in the form of the sum of a backward-view TD update.
  - That is, if this expression was the trace at time k, then we could update it from its value at time $k-1$ by:

$$\begin{aligned}\mathbf{z}_k &= \sum_{t=0}^{k} \rho_t \nabla \hat{v}(S_t, \mathbf{w}_t) \prod_{i=t+1}^{k} \gamma_i \lambda_i \rho_i \\ &= \sum_{t=0}^{k-1} \rho_t \nabla \hat{v}(S_t, \mathbf{w}_t) \prod_{i=t+1}^{k} \gamma_i \lambda_i \rho_i + \rho_k \nabla \hat{v}(S_k, \mathbf{w}_k) \\ &= \gamma_k \lambda_k \rho_k \underbrace{\sum_{t=0}^{k-1} \rho_t \nabla \hat{v}(S_t, \mathbf{w}_t) \prod_{i=t+1}^{k-1} \gamma_i \lambda_i \rho_i}_{\mathbf{z}_{k-1}} + \rho_k \nabla \hat{v}(S_k, \mathbf{w}_k)\end{aligned}$$

$$\boxed{\mathbf{z}_k = \rho_k[\gamma_k \lambda_k \, \mathbf{z}_{k-1} + \nabla \hat{v}(S_k, \mathbf{w}_k)]}$$

- If we change the index from k to t of the $\mathbf{z}_k$ equation above, we get the **general accumulating trace** update for state values:

$$\boxed{\mathbf{z}_t \doteq \rho_t[\gamma_t \lambda_t \, \mathbf{z}_{t-1} + \nabla \hat{v}(S_t, \mathbf{w}_t)]}$$

- This eligibility trace combined with the usual semi-gradient TD(λ) parameter-update rule (Section 12.2) forms a **general TD(λ)** algorithm that can be applied to either on-policy or off-policy data:
  - In on-policy, the algorithm is exactly TD(λ) because $\rho_t = 1$ always and the ET above becomes the usual accumulating trace for variable λ and γ:

$$\mathbf{z}_t \doteq \gamma_t \lambda_t \, \mathbf{z}_{t-1} + \nabla \hat{v}(S_t, \mathbf{w}_t)$$

- In off-policy, the algorithm stays as it is, although not guaranteed to be stable as a semi-gradient method.
- For off-policy, we'll consider extensions that guarantee stability in the next few sections.
- Let's derive the off-policy ET for **action-value** methods and corresponding general Sarsa(λ) algorithms.
  - Starting with either recursive general action-based λ-return of Sarsa or Expected Sarsa, $G_t^{\lambda a}$, in Section 12.8 (Expected Sarsa works out to be simpler), we can extend the Expected Sarsa $G_t^{\lambda a}$ to the off-policy case after the off-policy model of action-based, off-policy, control variate, n-step return:

$$\boxed{\begin{aligned}G_t^{\lambda a} &\doteq R_{t+1} + \gamma_{t+1}\big([1 - \lambda_{t+1}]\, \bar{V}_t(S_{t+1}) + \lambda_{t+1}\big[\rho_{t+1} G_{t+1}^{\lambda a} + \bar{V}_t(S_{t+1}) - \rho_{t+1}\, \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t)\big]\big) \\ &= R_{t+1} + \gamma_{t+1}\big(\bar{V}_t(S_{t+1}) + \lambda_{t+1}\rho_{t+1}\big[G_{t+1}^{\lambda a} - \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t)\big]\big)\end{aligned}}$$

where

$$\bar{V}_t(S_{t+1}) = \sum_a \pi(a|S_{t+1})\, \hat{q}(S_{t+1}, a, \mathbf{w}_t)$$

- The λ-return, approximately as the sum of TD errors, is:

$$\begin{aligned}G_t^{\lambda a} &\approx \hat{q}(S_t, A_t, \mathbf{w}_t) + \sum_{k=t}^{\infty} \delta_k^a \prod_{i=t+1}^{k} \gamma_i \lambda_i \rho_i \\ \delta_t^a &= R_{t+1} + \gamma_{t+1} \bar{V}(S_{t+1}) - \hat{q}(S_t, A_t, \mathbf{w}_t)\end{aligned}$$

- Using steps analogous to those for the state case earlier in this section, write a forward-view update based on action-based λ-return $G_t^{\lambda a}$ above, then transform the sum of the updates using the summation rule and finally derive the eligibility trace for action values:

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

- This ET combined with the action-based, expected TD error δ_t^a and the usual semi-gradient TD(λ) parameter-update rule (Section 12.2) forms an elegant, efficient **Expected Sarsa(λ)** algorithm that can be applied to either on-policy or off-policy data:
  - **On-policy case:** The algorithm becomes the Sarsa(λ) algorithm given constant λ and γ, and the usual state-action TD error:

$$\rho_t = 1, \quad \nabla\lambda_t = \nabla\gamma_t = 0$$

$$\boxed{\mathbf{z}_t \doteq \gamma\lambda\,\mathbf{z}_{t-1} + \nabla\hat{q}(S_t, A_t, \mathbf{w}_t)}$$

- At $\lambda = 1$, these algorithms become closely related to corresponding Monte Carlo algorithms.
- No episode-by-episode equivalence of updates exist, only of their expectations, even under the most favorable conditions.
  - Methods have been proposed recently [**Sutton, Mahmood, Precup & van Hasselt, 2014**] that do achieve an exact equivalence.
  - These methods require an additional vector of **"provisional weights"** that keep track of executed updates but may need to be retracted/emphasized depending on future actions taken.
  - The state and state-action versions of these methods are called **PTD(λ) and PQ(λ)** respectively, where the 'P' stands for Provisional.
- If $\lambda < 1$, then all these off-policy algorithms involve bootstrapping and **the deadly triad** applies, meaning that they can be guaranteed stable only for the tabular case, state aggregation and other limited forms of function approximation.
- Recall the challenge of off-policy learning has 2 parts. Off-policy eligibility traces deal effectively with the 1st part, correcting for the expected value of the targets, but not with the 2nd part that has to do with distribution of updates (matching off-policy to on-policy).
- Algorithmic strategies for handling the 2nd part of the off-policy learning challenge with eligibility traces are summarized in Section 12.11.

12.10. Watkins's Q(λ) to Tree-Backup(λ)

12.10.1. Watkins's Q(λ)

- Watkins's Q(λ) is the original method for extending Q-learning to eligibility traces.
- It involves decaying the ET in the usual way as long as a greedy action was taken, then cuts the traces to 0 after the 1st non-greedy action.

12.10.2. Tree-Backup(λ)

- Let's look at the eligibility trace version of Tree Backup, which is called **Tree-Backup(λ)** or **TB(λ)**.
- TB(λ) is the **true successor** to Q-learning because it has no importance sampling.
- TB(λ) concept is straightforward:
  - The tree-backup updates of each length (Section 7.5) are weighted dependent on the bootstrapping parameter λ.
  - Using the recursive form of the action-based λ-return for Expected Sarsa and then expanding the bootstrapping target case after the model of tree-backup n-step return (Section 7.5):

$$\boxed{\begin{aligned} G_t^{\lambda a} &\doteq R_{t+1} + \gamma_{t+1}\left([1-\lambda_{t+1}]\,\bar{V}_t(S_{t+1}) + \lambda_{t+1}\left[\sum_{a\neq A_{t+1}} \pi(a|S_{t+1})\,\hat{q}(S_{t+1}, a, \mathbf{w}_t) + \pi(A_{t+1}|S_{t+1})G_{t+1}^{\lambda a}\right]\right) \\ &= R_{t+1} + \gamma_{t+1}(\bar{V}_t(S_{t+1}) + \lambda_{t+1}\pi(A_{t+1}|S_{t+1})[G_{t+1}^{\lambda a} - \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t)]) \end{aligned}}$$

- $G_t^{\lambda a}$ can be approximated (ignoring changes in approx. value function) as a sum of TD errors:

$$G_t^{\lambda a} \approx \hat{q}(S_t, A_t, \mathbf{w}_t) + \sum_{k=t}^{\infty} \delta_k^a \prod_{i=t+1}^{k} \gamma_i\lambda_i\pi(A_i|S_i)$$

$$\delta_t^a = R_{t+1} + \gamma_{t+1}\bar{V}_t(S_{t+1}) - \hat{q}(S_t, A_t, \mathbf{w}_t)$$

- As always, using same steps as in the previous section, we get a special eligibility trace update involving the target-policy probabilities of the selected actions:

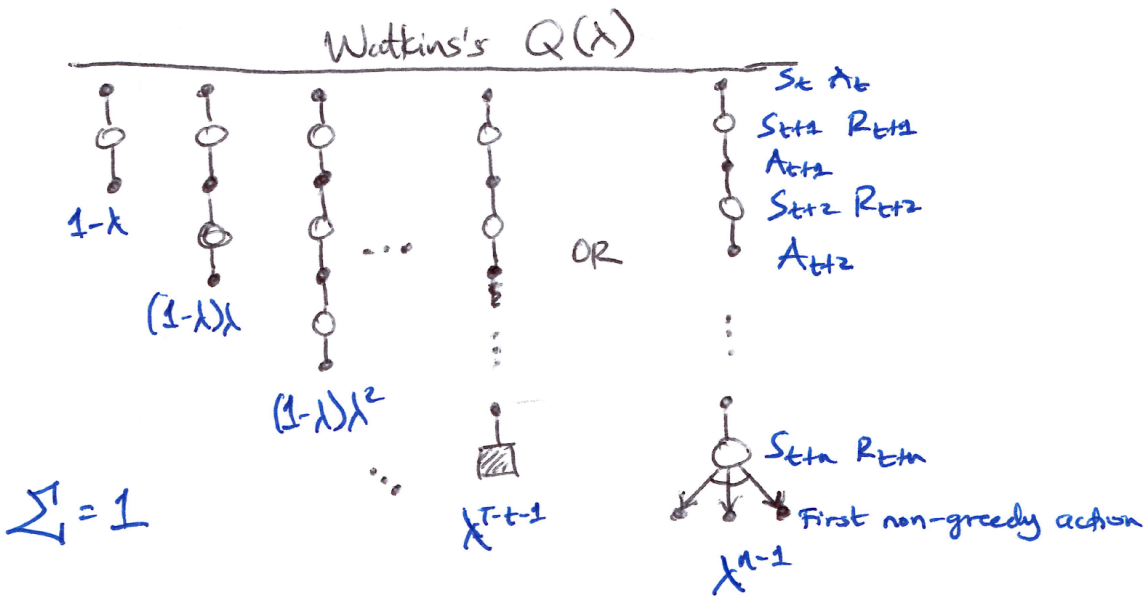

Figure 12.8. **Backup Diagram for Watkins's Q(λ).** Multi-step backups cut off as soon as an exploratory (nongreedy) action occurs or the episode ends.

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

- The ET above combined with the usual semi-gradient TD(λ) parameter-update rule defines the TB(λ) algorithm.
- Like all semi-gradient algorithms, TB(λ) is not guaranteed to be stable when used with off-policy data and a powerful function approximator **(the deadly triad).**

12.11. Stable Off-Policy Methods with Traces

- Let's look at 4 of the most important methods that achieve stable off-policy methods/training using eligibility traces.
- All 4 are based on either the **Gradient-TD or Emphatic TD** and linear function approximation.

12.11.1. GTD(λ)

- Analogous to TDC, and aims to learn a parameter $\mathbf{w}_t$ such that $\hat{v}(s, \mathbf{w}) \doteq \mathbf{w}_t^T \mathbf{x}(s) \approx v_\pi(s)$ even from data that is due to following another policy b. Its update is:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha\, \delta_t^s\, \mathbf{z}_t - \alpha\gamma_{t+1}(1-\lambda_{t+1})(\mathbf{z}_t^T \mathbf{v}_t)\, \mathbf{x}_{t+1}$$
$$\mathbf{v}_{t+1} \doteq \mathbf{v}_t + \beta\, \delta_t^s\, \mathbf{z}_t - \beta(\mathbf{v}_t^T \mathbf{x}_t)\, \mathbf{x}_t$$

where

$\mathbf{v} \in \mathbb{R}^d \equiv$ a vector of the same dimension as $\mathbf{w}$, initialized to $\mathbf{v}_0 = \mathbf{0}$

$\beta > 0 \equiv$ a 2nd step-size parameter

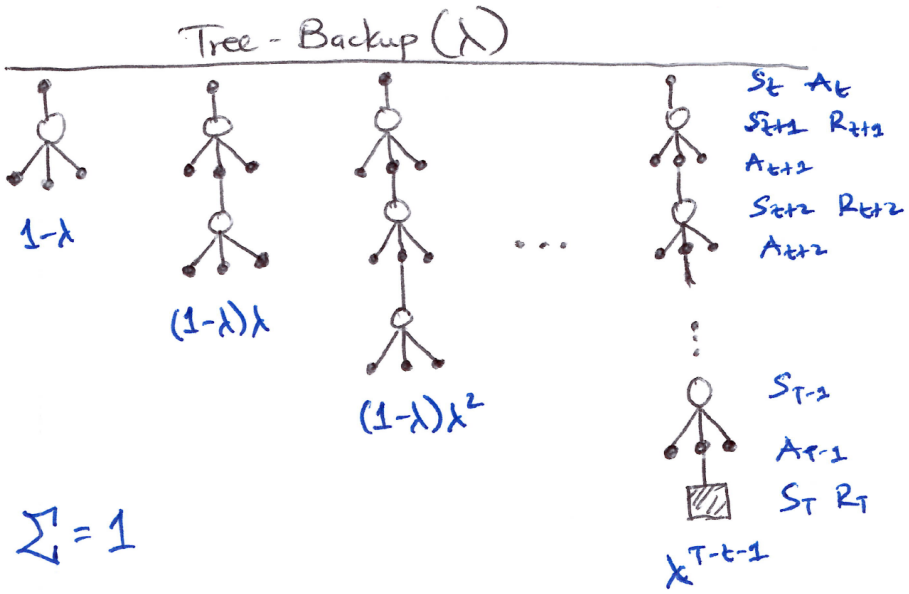

Figure 12.9. **Backup Diagram for Tree-Backup(λ).** Expected multi-step tree backups of every horizon combined with standard λ-decay weights, requiring no importance sampling.

12.11.2. GQ(λ)

- Gradient-TD algorithm for action values with eligibility traces.
- GQ(λ) aims to learn $\mathbf{w}_t$ such that $\hat{q}(s,a,\mathbf{w}_t) \doteq \mathbf{w}_t^T \mathbf{x}(s,a) \approx q_\pi(s,a)$ from off-policy data.
- If the target policy is ε-greedy, or otherwise biased towards the greedy policy for $\hat{q}$, then GQ(λ) can be used as a control algorithm.
- GQ(λ) update is:

$$\begin{aligned}
\mathbf{w}_{t+1} &\doteq \mathbf{w}_t + \alpha\, \delta_t^a\, \mathbf{z}_t - \alpha\gamma_{t+1}(1-\lambda_{t+1})(\mathbf{z}_t^T \mathbf{v}_t)\, \bar{\mathbf{x}}_{t+1} \\
\bar{\mathbf{x}}_t &\doteq \sum_a \pi(a|S_t)\, \mathbf{x}(S_t,a) \\
\delta_t^a &\doteq R_{t+1} + \gamma_{t+1}\, \mathbf{w}_t^T \bar{\mathbf{x}}_{t+1} - \mathbf{w}_t^T \mathbf{x}_t \\
\mathbf{z}_t &\doteq \gamma_t \lambda_t \rho_t\, \mathbf{z}_{t-1} + \nabla \hat{q}(S_t, A_t, \mathbf{w}_t)
\end{aligned}$$

where

$\bar{\mathbf{x}}_t \equiv$ average feature vector for S_t under the target policy

$\delta_t^a \equiv$ expectation form of the TD error

12.11.3. HTD(λ)

- Hybrid TD(λ) state-value algorithm combines aspects of GTD(λ) and TD(λ).
- HTD(λ) is a strict generalization of TD(λ) to the off-policy setting, meaning it reduces exactly to TD(λ) when the behavior and target policies coincide – a property GTD(λ) does not share:

$$b(A_t|S_t) = \pi(A_t|S_t), \quad \rho_t = 1 \implies \text{HTD}(\lambda) = \text{TD}(\lambda)$$

- HTD(λ) is defined by:

$$
\begin{aligned}
\mathbf{w}_{t+1} &\doteq \mathbf{w}_t + \alpha\,\delta_t^s\,\mathbf{z}_t + \alpha\Big[\big(\mathbf{z}_t - \mathbf{z}_t^b\big)^T\,\mathbf{v}_t\Big]\big(\mathbf{x}_t - \gamma_{t+1}\mathbf{x}_{t+1}\big) \\
\mathbf{v}_{t+1} &\doteq \mathbf{v}_t + \beta\,\delta_t^s\,\mathbf{z}_t - \beta(\mathbf{z}_t^T\mathbf{v}_t)(\mathbf{x}_t - \gamma_{t+1}\mathbf{x}_{t+1})\,, && \mathbf{v}_0 \doteq \mathbf{0} \\
\mathbf{z}_t &\doteq \rho_t(\gamma_t\lambda_t\,\mathbf{z}_{t-1} + \mathbf{x}_t)\,, && \mathbf{z}_{-1} \doteq \mathbf{0} \\
\mathbf{z}_t^b &\doteq \gamma_t\lambda_t\,\mathbf{z}_{t-1}^b + \mathbf{x}_t, && \mathbf{z}_{-1}^b \doteq \mathbf{0}
\end{aligned}
$$

- We get
  - a 2nd set of weights, $\mathbf{v}_t$.
  - a 2nd set of eligibility traces, $\mathbf{z}_t^b$, **conventional accumulating traces** for the behavior policy.

$$
\begin{aligned}
\mathbf{z}_t^b = \mathbf{z}_t \text{ if all } \rho_t = 1 &\implies \big(\mathbf{z}_t - \mathbf{z}_t^b\big)^T = \mathbf{0} \\
&\implies \mathbf{w}_{t+1} = \mathbf{w}_t + \alpha\,\delta_t^s\,\mathbf{z}_t \quad \text{(TD}(\lambda)\text{)}
\end{aligned}
$$

12.11.4. Emphatic TD(λ)

- Extension of one-step Emphatic TD (Sections 9.11 & 11.8) to eligibility traces.
- The resulting algorithm:
  - (+) retains strong off-policy convergence guarantees
  - (+) enables any degree of bootstrapping
  - (-) has high variance
  - (-) potentially slow convergence
- Emphatic TD(λ) is defined by:

$$
\begin{aligned}
\mathbf{w}_{t+1} &\doteq \mathbf{w}_t + \alpha\,\delta_t\,\mathbf{z}_t \\
\delta_t &\doteq R_{t+1} + \gamma_{t+1}\,\mathbf{w}_t^T\mathbf{x}_{t+1} - \mathbf{w}_t^T\mathbf{x}_t \\
\mathbf{z}_t &\doteq \rho_t(\gamma_t\lambda_t\,\mathbf{z}_{t-1} + M_t\mathbf{x}_t)\,, && \mathbf{z}_{-1} \doteq \mathbf{0} \\
M_t &\doteq \lambda_t\mathcal{I}_t + (1-\lambda_t)F_t \\
F_t &\doteq \rho_{t-1}\gamma_t F_{t-1} + \mathcal{I}_t, && F_0 \doteq i(S_0)
\end{aligned}
$$

where

$M_t \geq 0 \equiv$ emphasis

$F_t \geq 0 \equiv$ followon trace

$\mathcal{I}_t \geq 0 \equiv$ interest

- In the on-policy case ($\rho_t = 1$ for all t), Emphatic TD(λ) is similar to conventional TD(λ), but still significantly different:
  - Emphatic TD(λ) is guaranteed to converge for all state-dependent λ functions; TD(λ) is not (TD(λ) is guaranteed only for constant λ).
  - See Yu's counterexample **[Ghassian, Rafiee & Sutton, 2016]**.

12.12. Implementation Issues

- **Naive implementation seems expensive:** Updating eligibility traces for every state at every time step appears computationally costly on serial computers.
- **Practical optimization:** Most ET are nearly 0; only recently visited states have significant traces, so implementations can track and update only these few states.
- **Computational cost:** With this optimization, tabular methods with traces are only a few times more expensive than one-step methods.
- **Function approximation reduces overhead:** When using neural networks, ET typically only double memory and computation per step (much less overhead than in tabular case).
- **Tabular is the worst case:** The tabular setting represents the highest computational complexity for ET relative to simpler methods.

12.13. Conclusions

- **Eligibility traces** provide an efficient, incremental way to interpolate between TD and MC methods.
- ET offer advantages over n-step methods in terms of **generality and computational trade-offs**.
- Empirically, **an intermediate mix works best:** ET should move towards MC but not all the way since pure MC performance degrades sharply.
- ET are the **first defense against long-delayed rewards and non-Markov tasks**, used with TD methods to make them behave more like MC methods without full bootstrapping.
- Use traces **when data is scarce and online learning is required**, as they provide faster learning per sample despite higher computational cost per step.
- **Avoid traces in offline settings** with cheap abundant data (maximum data processing speed matters more than learning efficiency per sample).
- **True online methods** achieve ideal λ-return performance while maintaining $O(d)$ computational efficiency.
- Forward-to-backward view derivations provide **computationally efficient, mechanistic**, practical implementations of theory.

13. Policy Gradient Methods

- Almost all the algorithms/methods covered so far have been **action-value methods** (except gradient-bandit algorithms, Section 2.8).
- Action-value methods learn the values of actions and then derive the policy thereafter to select actions based on their estimated action values.
- Here, we explicitly learn a **parametrized policy** that can select actions without consulting a value function.
- A value function is not required for action selection, but still could be used to learn the policy parameters θ.
- The parametrized policy $\pi(a|s,\theta)$ is now the probability that action a is taken at time t given that the environment is in state s at time t with parameter $\theta \in \mathbb{R}^{d'}$:

$$\pi(a|s,\theta) = \Pr\{A_t = a | S_t = s,\ \theta_t = \theta\}$$

- We will consider methods for learning θ based on the gradient of some scalar performance measure $J(\theta)$ w.r.t. θ. The goal is to **maximize** performance, hence the use of gradient ascent:

$$\theta_{t+1} = \theta_t + \alpha \widehat{\nabla J(\theta_t)}$$

where

$\widehat{\nabla J(\theta_t)} \in \mathbb{R}^{d'} \equiv$ a stochastic estimate whose expectation approximates the gradient of the performance measure J w.r.t. θ_t

- This general methodology applies to **policy gradient methods**.
- Methods that learn approximations to both policy & value functions are often called **Actor-Critic methods**:
  - **Actor** refers to the learned policy.
  - **Critic** refers to the learned value function (state-value function).

13.1. Policy Approximation & Its Advantages

- In policy gradient methods, the policy can be parametrized in any way, as long as $\pi(a|s,\theta)$ is differentiable w.r.t. θ (essentially the partial derivatives' column vector exists and is finite):

$$\nabla\pi(a|s,\theta) \text{ exists and is finite for all } s \in S,\ a \in A(s),\ \theta \in \mathbb{R}^{d'}$$

- If the action-space is **discrete** and not too large, then a natural & common kind of parametrization is to form parametrized **numerical preferences** $h(s,a,\theta) \in \mathbb{R}$ for each (s,a) 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:

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

- This kind of policy parametrization is called **softmax in action preferences**.
- The action preferences can be parametrized arbitrarily, they can be computed by a deep artificial neural network (ANN) or could simply be **linear** in features:

$$h(s,a,\theta) = \theta^T \mathbf{x}(s,a)$$

Advantages of softmax in action preferences policy parametrization:

- The approx. policy can approach a **deterministic policy**, unlike ε-greedy action selection.
- It enables the selection of actions with **arbitrary probabilities**, which is useful for cases that require near-optimal stochastic policy (e.g. significant function approximation).
  - E.g. useful in environments with imperfect information (e.g., card games) where it is optimal to act stochastically, such as when **bluffing in Poker**; it is important to do so randomly to unnerve/confuse an opponent.
- The policy from policy parametrization may be a **simpler function to approximate** than the action-value function.
- Often the most important reason for using a policy-based learning method is that it is a good way to **inject prior knowledge** about the desired form of the policy into the RL system.

13.2. The Policy Gradient Theorem

- Policy-gradient methods have stronger convergence guarantees compared to action-value methods due to **smooth continuity** in change of action probabilities as a function of the learned parameter (gradient ascent).
- Let's consider the **episodic** performance measure, which is the value of the start state of the episode:

$$J(\theta) \doteq v_{\pi_\theta}(s_0)$$

where

$v_{\pi_\theta} \equiv$ the true value function for π_θ, the policy determined by θ

$s_0 \equiv$ some non-random state

- How can we estimate the performance gradient w.r.t. the policy parameter when the gradient depends on the unknown effect of policy changes on the state distribution?
  - The theoretical answer is the **policy gradient theorem**.
- The policy gradient theorem, which provides an analytic expression for the performance gradient w.r.t. the policy parameter, for the episodic case establishes that:

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

where

$\nabla J(\theta) \equiv$ column vectors of partial derivatives w.r.t. the components of θ

$\mu \equiv$ the on-policy distribution under the policy π

(in the episodic case, proportionality constant is the average length of an episode;
in the continuing case it is 1)

i Proof: Policy Gradient Theorem (Episodic Case)

Let's prove the policy gradient theorem from first principles using elementary calculus.
Note: To keep the notation simple, we leave it implicit in all cases that $\pi = f(\theta)$ and all gradients $\nabla[...]$ are also implicit w.r.t. θ.

$$\begin{aligned}
\nabla v_\pi(s) &= \nabla \left[\sum_a \pi(a|s)\, q_\pi(s,a) \right], \quad \text{for all } s \in \mathcal{S} \\
&= \sum_a [\nabla \pi(a|s)\, q_\pi(s,a) + \pi(a|s)\, \nabla q_\pi(s,a)] \\
&= \sum_a \left[\nabla \pi(a|s)\, q_\pi(s,a) + \pi(a|s)\, \nabla \sum_{s',r} p(s',r|s,a)(r + v_\pi(s')) \right] \\
&= \sum_a \left[\nabla \pi(a|s)\, q_\pi(s,a) + \pi(a|s) \sum_{s'} p(s'|s,a)\, \nabla v_\pi(s') \right] \\
&= \sum_a \left[\nabla \pi(a|s)\, q_\pi(s,a) + \pi(a|s) \sum_{s'} p(s'|s,a) \sum_{a'} \left[\nabla \pi(a'|s')\, q_\pi(s',a') \right.\right. \\
&\qquad \left.\left. + \pi(a'|s') \sum_{s''} p(s''|s',a')\, \nabla v_\pi(s'') \right] \right]
\end{aligned}$$

Unrolling this recursion:

$$\boxed{\nabla v_\pi(s) = \sum_{x \in \mathcal{S}} \sum_{k=0}^{\infty} \Pr(s \to x, k, \pi) \sum_a \nabla \pi(a|x)\, q_\pi(x,a)}$$

$\Pr(s \to x, k, \pi) \equiv$ probability of transitioning from state s to state x in k steps under policy π

Then:

$$\begin{aligned}
\nabla J(\theta) &= \nabla v_\pi(s_0) \\
&= \sum_s \left(\sum_{k=0}^{\infty} \Pr(s_0 \to s, k, \pi) \right) \sum_a \nabla \pi(a|s)\, q_\pi(s,a) \\
&= \sum_s \eta(s) \sum_a \nabla \pi(a|s)\, q_\pi(s,a) \\
&= \sum_{s'} \eta(s') \sum_s \frac{\eta(s)}{\sum_{s'} \eta(s')} \sum_a \nabla \pi(a|s)\, q_\pi(s,a) \\
&= \sum_{s'} \eta(s') \sum_s \mu(s) \sum_a \nabla \pi(a|s)\, q_\pi(s,a)
\end{aligned}$$

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

13.3. REINFORCE: Monte Carlo Policy Gradient

- REINFORCE is our first policy gradient algorithm.
- The goal/strategy is to find a way to get samples such that the expectation of the sample gradient is proportional to the actual performance gradient as a function of the parameters.
- The sample gradients need to only be proportional to the performance gradient because any proportionality constant can be absorbed into the step size α.
- The policy gradient theorem gives **an exact expression proportional to the gradient**; so all that is needed is a way of sampling whose expectation equals or approximates this expression.

- Recall the RHS of the policy gradient theorem is a sum over states weighted by how often the states occur under the target policy π:

$$\begin{aligned}\nabla J(\theta) &\propto \sum_s \mu(s) \sum_a q_\pi(s,a)\, \nabla\pi(a|s,\theta) \\ &= \mathbb{E}_\pi\left[\sum_a q_\pi(S_t,a)\, \nabla\pi(a|S_t,\theta)\right]\end{aligned}$$

- We get REINFORCE by replacing the sum over the random variable's possible values by an expectation under π, and then sampling the expectation:

$$\begin{aligned}\nabla J(\theta) &\propto \mathbb{E}_\pi\left[\sum_a \pi(a|S_t,\theta)\, q_\pi(S_t,a)\, \frac{\nabla\pi(a|S_t,\theta)}{\pi(a|S_t,\theta)}\right] \\ &= \mathbb{E}_\pi\left[q_\pi(S_t,A_t)\, \frac{\nabla\pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)}\right] \quad \text{(replacing } a \text{ by the sample } A_t \sim \pi) \\ &= \mathbb{E}_\pi\left[G_t\, \frac{\nabla\pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)}\right] \quad \text{(because } \mathbb{E}_\pi[G_t|S_t,A_t] = q_\pi(S_t,A_t))\end{aligned}$$

- The last expression is the required expression; a quantity that can be sampled on each time step whose expectation is proportional to the gradient. This leads to the **REINFORCE update**:

$$\boxed{\theta_{t+1} \doteq \theta_t + \alpha G_t \frac{\nabla\pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)}}$$

- This update is intuitively coherent:
  - The gradient term represents the direction in parameter space that most increases the probability of repeating/taking again the same action A_t in the future.
  - The update moves the parameter the most in the directions that favour actions that yield the highest return.
  - The update ensures a balancing act (lower impact) of frequently selected actions with lower returns.
- REINFORCE has good convergence properties but may be of high variance and slow learning as a MC method.

13.3.1. Pseudocode: REINFORCE (Episodic) for estimating π_*

Input: a differentiable policy parametrization $\pi(a|s,\theta)$

Algorithm parameter: step size $\alpha > 0$

Initialize policy parameter $\theta \in \mathbb{R}^{d'}$ (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 $\pi(\cdot|\cdot,\theta)$

Loop for each step of the episode $t = 0, 1, 2, \ldots, T-1$:

$G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$

$\theta \leftarrow \theta + \alpha\gamma^t G\, \nabla \ln \pi(A_t|S_t,\theta)$

13.4. REINFORCE with Baseline

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

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a [q_\pi(s,a) - b(s)]\, \nabla\pi(a|s,\theta)$$

- The baseline can be any function that is not dependent on the action a:

$$\sum_a b(s)\,\nabla\pi(a|s,\theta) = b(s)\,\nabla\sum_a \pi(a|s,\theta) = b(s)\,\nabla 1 = 0$$

- Now the REINFORCE update with baseline is:

$$\boxed{\theta_{t+1} \doteq \theta_t + \alpha[G_t - b(S_t)]\,\frac{\nabla\pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)}}$$

- The Baseline REINFORCE update **reduces the variance** despite leaving the expected update value unchanged, which speeds up learning.
- One natural choice for the baseline is an estimate of the state value $\hat{v}(S_t, \mathbf{w})$.
- Since REINFORCE is a Monte Carlo method for learning the policy parameter θ, it's natural to also use a Monte Carlo method to learn the state-value weights $\mathbf{w}$.

13.4.1. Pseudocode: REINFORCE with Baseline (Episodic), for estimating $\pi_\theta \approx \pi_*$

Input: a differentiable policy parametrization $\pi(a|s,\theta)$

Input: a differentiable state-value function parametrization $\hat{v}(s,\mathbf{w})$

Algorithm parameters: step sizes $\alpha^\theta > 0,\ \alpha^{\mathbf{w}} > 0$

Initialize policy parameter $\theta \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^d$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):

Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$ following $\pi(\cdot|\cdot,\theta)$

Loop for each step of the episode $t = 0, 1, \ldots, T-1$:

$$G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$$

$$\delta \leftarrow G - \hat{v}(S_t,\mathbf{w})$$

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}}\,\delta\,\nabla\hat{v}(S_t,\mathbf{w})$$

$$\theta \leftarrow \theta + \alpha^\theta \gamma^t\,\delta\,\nabla \ln\pi(A_t|S_t,\theta)$$

- This algorithm has 2 step sizes α^θ and $\alpha^{\mathbf{w}}$.
- Choosing $\alpha^{\mathbf{w}}$ is relatively easy; in the linear case a good rule of thumb is (see Section 9.6):

$$\alpha^{\mathbf{w}} = \frac{0.1}{\mathbb{E}\left[\|\nabla\hat{v}(S_t,\mathbf{w})\|_\mu^2\right]}$$

- Choosing α^θ is much less clear since its best value depends on the range of variation of the rewards and on the policy parametrization.

13.5. Actor-Critic Methods

- REINFORCE with baseline cannot be used to evaluate actions because its state-value function **only estimates the value of the 1st state** of each state transition (1st state to 2nd state), which serves as a baseline of what to expect for the subsequent return to be, prior to the actual transition's action.
- **Actor-critic methods, however, apply the state-value function to the 2nd state** of the transition thereby estimating its value and thus **evaluating the action**.
- The policy is the **actor** that maps states to actions, while the state-value function used to assess actions in this way is the **critic**.
- The estimated value of the 2nd state, when discounted & added to the reward, yields the **one-step return**, $G_{t:t+1}$.

13.5.1. One-Step Actor-Critic Methods

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

$$\begin{aligned}\theta_{t+1} &\doteq \theta_t + \alpha[G_{t:t+1} - \hat{v}(S_t, \mathbf{w})] \frac{\nabla \pi(A_t|S_t, \theta_t)}{\pi(A_t|S_t, \theta_t)} \\ &= \theta_t + \alpha[R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w})] \frac{\nabla \pi(A_t|S_t, \theta_t)}{\pi(A_t|S_t, \theta_t)} \\ &= \theta_t + \alpha\, \delta_t \frac{\nabla \pi(A_t|S_t, \theta_t)}{\pi(A_t|S_t, \theta_t)}\end{aligned}$$

- The semi-gradient TD(0) could serve as the natural state-value function learning method.
- **PROS:** simple, fully online & incremental.
- It is analogous to TD(0), Sarsa(0) & Q-learning.

13.5.2. Pseudocode: One-Step Actor-Critic (Episodic), for estimating $\pi_\theta \approx \pi_*$

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

13.5.3. Actor-Critic with Eligibility Traces

- We generalize to the forward view of n-step methods and then to a λ-return algorithm.
- We replace the one-step return $G_{t:t+1}$ by $G_{t:t+n}$ or G_t^λ respectively for either n-step or λ-return.
- The backward view of the λ-return algorithm uses **eligibility traces** for the actor and the critic.

13.5.4. Pseudocode: Actor-Critic with ET (Episodic), for estimating $\pi_\theta \approx \pi_*$

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

13.6. Policy Gradient for Continuing Problems

- From Section 10.3 on continuing problems, lack of episode boundaries requires a new performance measure definition in terms of the **average rate of reward per time step**:

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

  where

$$\mu(s) \doteq \lim_{t\to\infty}\Pr\{S_t = s|A_{0:t}\sim\pi\} \equiv \text{steady state distribution under } \pi,$$

  assumed to exist and be independent of S_0 **[ergodicity assumption]**

$$\sum_s \mu(s)\sum_a \pi(a|s,\theta)\,p(s'|s,a) = \mu(s') \quad \text{for all } s' \in S \quad \text{(ergodicity)}$$

- In the continuing case, we define values $v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t = s]$ and $q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$ w.r.t. the **differential return**:

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

Proof: Policy Gradient Theorem (Continuing Case)

Leave the notation implicit in all cases that $\pi = f(\theta)$ and that the gradients $\nabla[...]$ are w.r.t. θ. In the continuing case $J(\theta) = r(\pi)$, and v_π & q_π denote values w.r.t. the **differential return**.

$$\begin{aligned}
\nabla v_\pi(s) &= \nabla \left[\sum_a \pi(a|s)\, q_\pi(s,a) \right], \quad \text{for all } s \in \mathcal{S} \\
&= \sum_a [\nabla \pi(a|s)\, q_\pi(s,a) + \pi(a|s)\, \nabla q_\pi(s,a)] \\
&= \sum_a \left[\nabla \pi(a|s)\, q_\pi(s,a) + \pi(a|s)\, \nabla \sum_{s',r} p(s',r|s,a)(r - r(\theta) + v_\pi(s')) \right] \\
&= \sum_a \left[\nabla \pi(a|s)\, q_\pi(s,a) + \pi(a|s) \left[-\nabla r(\theta) + \sum_{s'} p(s'|s,a)\, \nabla v_\pi(s') \right] \right] \\
\nabla r(\theta) &= \sum_a \left[\nabla \pi(a|s)\, q_\pi(s,a) + \pi(a|s) \sum_{s'} p(s'|s,a)\, \nabla v_\pi(s') \right] - \nabla v_\pi(s)
\end{aligned}$$

Since $\nabla J(\theta)$ does not depend on s, we sum over all $s \in \mathcal{S}$ weighted by $\mu(s)$ (because $\sum_s \mu(s) = 1$):

$$\begin{aligned}
\nabla J(\theta) &= \sum_s \mu(s) \left(\sum_a \left[\nabla \pi(a|s)\, q_\pi(s,a) + \pi(a|s) \sum_{s'} p(s'|s,a)\, \nabla v_\pi(s') \right] - \nabla v_\pi(s) \right) \\
&= \sum_s \mu(s) \sum_a \nabla \pi(a|s)\, q_\pi(s,a) \\
&\quad + \sum_{s'} \underbrace{\sum_s \mu(s) \sum_a \pi(a|s)\, p(s'|s,a)}_{\mu(s')}\, \nabla v_\pi(s') - \sum_s \mu(s)\, \nabla v_\pi(s) \\
&= \sum_s \mu(s) \sum_a \nabla \pi(a|s)\, q_\pi(s,a) + \sum_{s'} \mu(s')\, \nabla v_\pi(s') - \sum_s \mu(s)\, \nabla v_\pi(s) \\
&= \sum_s \mu(s) \sum_a \nabla \pi(a|s)\, q_\pi(s,a) \qquad \text{Q.E.D.}
\end{aligned}$$

13.6.1. Pseudocode: Actor-Critic with ET (Continuing), for estimating $\pi_\theta \approx \pi_*$

Inputs: $\pi(a|s,\theta)$, $\hat{v}(s,\mathbf{w})$

Parameters: $\lambda^{\mathbf{w}} \in [0,1]$, $\lambda^{\theta} \in [0,1]$, $\alpha^{\mathbf{w}} > 0$, $\alpha^{\theta} > 0$, $\alpha^{\bar{R}} > 0$

Initialize $\bar{R} \in \mathbb{R}$ (e.g., to 0), $\mathbf{w} \in \mathbb{R}^d$ & $\theta \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$), $S \in \mathcal{S}$ (e.g., to s_0)

$\mathbf{z}^{\mathbf{w}} \leftarrow \mathbf{0}$; $\mathbf{z}^{\theta} \leftarrow \mathbf{0}$

Loop forever (for each time step):

- $A \sim \pi(\cdot|S,\theta)$
- Take action A, observe S', R
- $\delta \leftarrow R - \bar{R} + \hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$
- $\bar{R} \leftarrow \bar{R} + \alpha^{\bar{R}}\, \delta$
- $\mathbf{z}^{\mathbf{w}} \leftarrow \lambda^{\mathbf{w}} \mathbf{z}^{\mathbf{w}} + \nabla \hat{v}(S,\mathbf{w})$
- $\mathbf{z}^{\theta} \leftarrow \lambda^{\theta} \mathbf{z}^{\theta} + \nabla \ln \pi(A|S,\theta)$
- $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}}\, \delta\, \mathbf{z}^{\mathbf{w}}$
- $\theta \leftarrow \theta + \alpha^{\theta}\, \delta\, \mathbf{z}^{\theta}$
- $S \leftarrow S'$

13.7. Policy Parametrization for Continuous Actions

- Policy gradient methods enable us to handle large (and even continuous) action spaces by learning the statistics of the probability distribution instead of computing learned probabilities for each of the many actions.
- The probability density function for the normal distribution is conventionally written as:

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

where

$\mu \equiv$ mean of the normal distribution

$\sigma \equiv$ standard deviation of the normal distribution

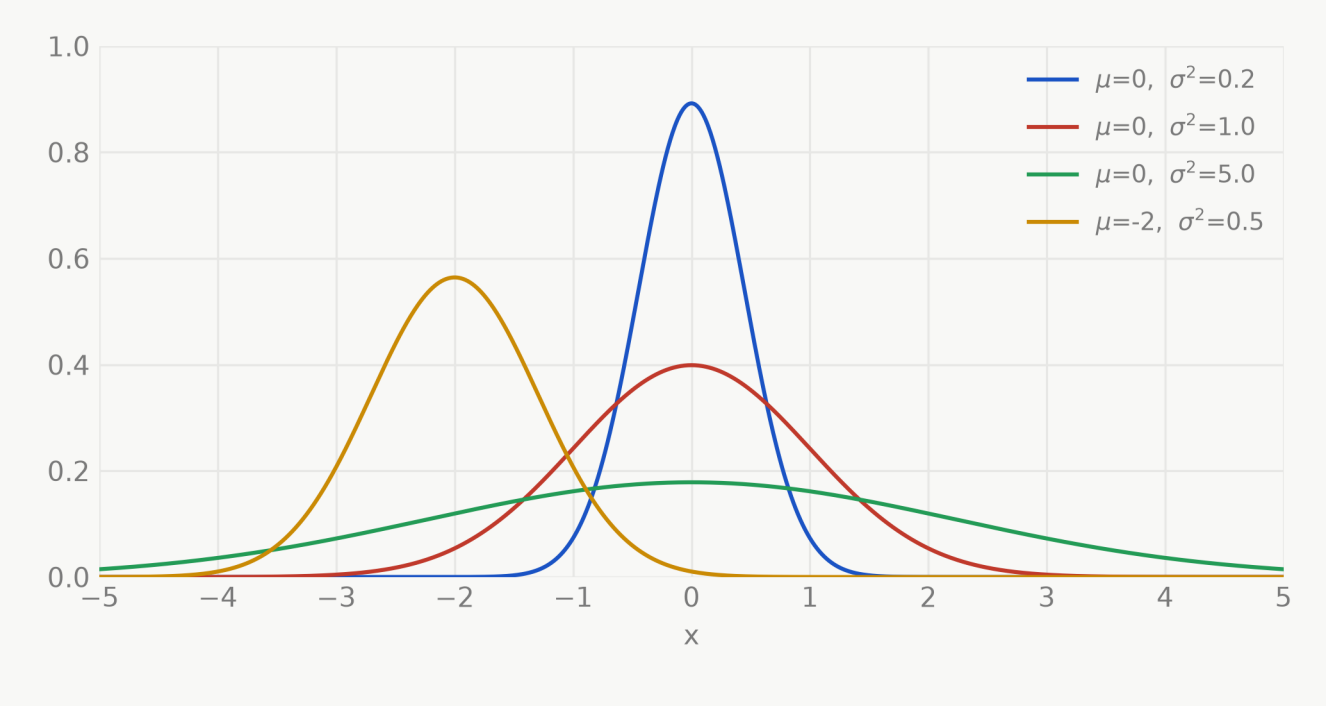

Figure 13.1. **Probability Density Function (PDF).** The probability density functions for several different means and standard deviations are shown above.

- $p(x)$ is the **density** of the probability at x, and not the probability. It can be greater than 1; it is the total area under the graph that must sum to 1.
- To get the probability of x falling within a range, take the integral under $p(x)$ for that specific range of x values.
- The policy parametrization, defining the policy as the normal probability density over a real-valued scalar action a with mean & standard deviation given by state-dependent, parametric function approximators, is as follows:

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

where

$\mu : S \times \mathbb{R}^{d'} \to \mathbb{R} \equiv$ mean function approximator

$\sigma : S \times \mathbb{R}^{d'} \to \mathbb{R}^+ \equiv$ standard deviation function approximator

- The approximators need a representation form, so we split the policy's parameter vector into 2 parts, $\theta = [\theta_\mu, \theta_\sigma]^T$, one for mean approximation and the other for standard deviation approximation.

- The mean can be approximated as a linear function while the standard deviation can be approximated as the exponential of a linear function:

$$\mu(s, \theta) \doteq \theta_\mu^T \mathbf{x}_\mu(s)$$
$$\sigma(s, \theta) \doteq \exp\left(\theta_\sigma^T \mathbf{x}_\sigma(s)\right)$$

13.8. Summary

- **Parametrized policy methods** select actions directly from $\pi(a \mid s, \theta)$ rather than deriving a policy indirectly from action-value estimates, adjusting θ along estimated ascent directions of performance $J(\theta)$.
- Key advantages over ε-greedy value-based control include representing genuinely stochastic optimal policies, smoothly annealing toward deterministic behavior, handling continuous action spaces naturally, and enjoying stronger convergence guarantees via the **policy gradient theorem**.
- **REINFORCE** pairs the policy gradient theorem with full Monte Carlo returns; subtracting a learned state-value **baseline** cuts gradient variance substantially without adding bias.
- When a learned value function bootstraps to evaluate transitions directly, it acts as a **critic** guiding the policy **actor** (**actor-critic methods**). Bootstrapping adds some bias, but the sharp drop in variance typically accelerates learning much as TD improves upon Monte Carlo.

A. Figure Gallery

All 42 figures in these notes, listed by chapter and numbered as they appear in the text.

Chapter	Figures
3	Finite Markov Decision Processes: 3.1, 3.2, 3.3
4	Dynamic Programming: 4.1
5	Monte Carlo Methods: 5.1
6	Temporal-Difference Learning: 6.1, 6.2, 6.3, 6.4, 6.5
7	n-step Bootstrapping: 7.1, 7.2, 7.3, 7.4, 7.5
8	Planning & Learning with Tabular Methods: 8.1, 8.2, 8.3, 8.4, 8.5, 8.6
9	On-Policy Prediction with Approximation: 9.1, 9.2, 9.3, 9.4
11	Off-Policy Methods with Approximation: 11.1, 11.2, 11.3, 11.4, 11.5, 11.6, 11.7
12	Eligibility Traces: 12.1, 12.2, 12.3, 12.4, 12.5, 12.6, 12.7, 12.8, 12.9
13	Policy Gradient Methods: 13.1

Notation Index

The book uses the textbook's notation throughout. Each symbol below links to the section where it is defined; the last column counts the sections that use it, so a symbol with a single hit is probably described only in passing.

Symbol	Meaning	First defined	Sections using it
S_t, A_t	state and action at time t	§1.5	57
R_{t+1}	reward received after step t	§2.1	49
G_t	return: discounted sum of future rewards	§3.3	32
γ	discount rate	§3.3	57
$\pi(a \mid s)$	policy: probability of each action in a state	§2.2	61
$v_\pi(s), q_\pi(s,a)$	state-value and action-value under π	§3.5	32
$v_*(s), q_*(s,a)$	optimal state-value and action-value	§2.1	13
$p(s', r \mid s, a)$	dynamics: transition probability and reward	§3.1	14
$b(a \mid s)$	behaviour policy (off-policy)	§5.5	10
ε	exploration rate in ε-greedy	§2.2	21
α	step size (learning rate)	§1.5	45
δ_t	TD error	§6.1	19
$\rho_{t:T-1}$	importance-sampling ratio	§5.5	15
$\mathbf{w}$	weight vector of a function approximator	§9.1	33
$\hat{v}(s, \mathbf{w})$	approximated value with weights $\mathbf{w}$	§9.1	29
$\mathbf{x}(s)$	feature vector for state s	§9.4	15
$\mathbf{z}_t$	eligibility trace	§12.2	9
λ	trace decay rate	§1.7	17
n	number of steps in n-step methods	§2.4	30
θ	policy parameters	§13.1	10
∇	gradient with respect to the parameters	§9.3	26
μ	on-policy state distribution $\mu(s)$, or continuous action mean	§5.4	19
σ	sampling-vs-expectation degree in $Q(\sigma)$, or std. dev.	§7.6	4
$\mathbb{E}[\cdot]$	expectation	§2.1	32
$\doteq$	"is defined as"	§2.1	59

Generated by `scripts/build-notation.cjs` from the chapter sources; re-run it after adding symbols.