17 Model-Free Methods

In contrast with model-based methods, *model-free reinforcement learning* does not require building explicit representations of the transition and reward models.[1] The model-free methods discussed in this chapter model the action value function directly. Avoiding explicit representations is attractive, especially when the problem is high dimensional. This chapter begins by introducing incremental estimation of the mean of a function, which plays an important role in estimating the mean of returns. We then discuss some common model-free algorithms and methods for handling delayed reward more efficiently. Finally, we discuss how to use function approximation to generalize from our experience.

17.1 Incremental Estimation of the Mean

Many model-free methods *incrementally estimate* the action value function $Q(s, a)$ from samples. For the moment, suppose we are only concerned with the expectation of a single variable X from m samples:

$$\hat{x}_m = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} \tag{17.1}$$

where $x^{(1)}, \ldots x^{(m)}$ are our samples. We can derive an incremental update:

$$\begin{aligned}
\hat{x}_m &= \frac{1}{m}\left(x^{(m)} + \sum_{i=1}^{m-1} x^{(i)}\right) && (17.2)\\
&= \frac{1}{m}\left(x^{(m)} + (m-1)\hat{x}_{m-1}\right) && (17.3)\\
&= \hat{x}_{m-1} + \frac{1}{m}\left(x^{(m)} - \hat{x}_{m-1}\right) && (17.4)
\end{aligned}$$

[1] Many of the topics in this chapter are covered in greater depth by R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. MIT Press, 2018.

We can rewrite this equation with the introduction of a *learning rate* function $\alpha(m)$:

$$\hat{x}_m = \hat{x}_{m-1} + \alpha(m)\left(x^{(m)} - \hat{x}_{m-1}\right) \tag{17.5}$$

The learning rate can be a function other than $1/m$. To ensure convergence, we generally select $\alpha(m)$ such that we have $\sum_{m=1}^{\infty} \alpha(m) = \infty$ and $\sum_{m=1}^{\infty} \alpha^2(m) < \infty$. The first condition ensures that the steps are sufficiently large, and the second condition ensures that the steps are sufficiently small.[2]

If the learning rate is constant, which is common in reinforcement learning applications, then the weights of older samples decay exponentially at the rate $(1 - \alpha)$. With a constant learning rate, we can update our estimate after observing x using the following rule:

$$\hat{x} \leftarrow \hat{x} + \alpha(x - \hat{x}) \tag{17.6}$$

Algorithm 17.1 provides an implementation. An example of several learning rates is shown in example 17.1.

The update rule above will appear again in later sections and is related to stochastic gradient descent. The magnitude of the update is proportional to the difference between the sample and the previous estimate. The difference between the sample and previous estimate is called the *temporal difference error*.

17.2 Q-Learning

Q-learning (algorithm 17.2) involves applying incremental estimation of the action value function $Q(s, a)$.[3] The update is derived from the action value form of the Bellman equation:

$$\begin{aligned} Q(s,a) &= R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U(s') & (17.7) \\ &= R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) \max_{a'} Q(s',a') & (17.8) \end{aligned}$$

Instead of using T and R, we can rewrite the equation above in terms of an expectation over samples of reward r and the next state s':

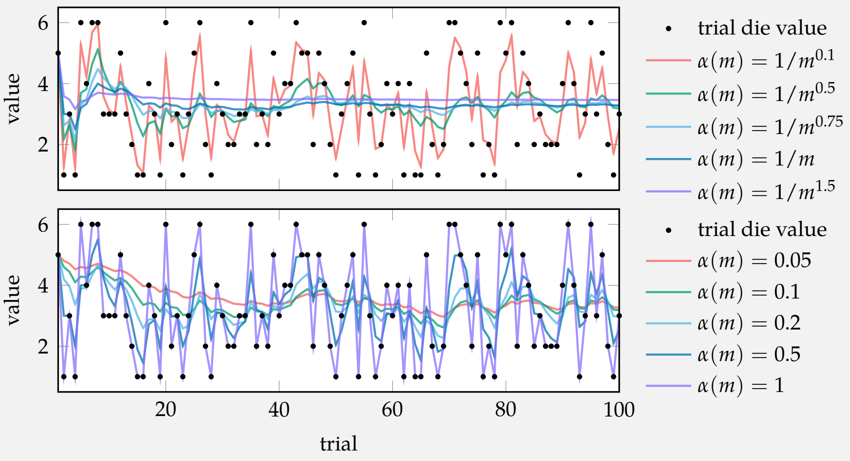
$$Q(s,a) = \mathbb{E}_{r,s'}[r + \gamma \max_{a'} Q(s',a')] \tag{17.9}$$

[2] For a discussion on convergence and application to some of the other algorithms discussed in this chapter, see T. Jaakkola, M. I. Jordan, and S. P. Singh, ''On the Convergence of Stochastic Iterative Dynamic Programming Algorithms,'' *Neural Computation*, vol. 6, no. 6, pp. 1185–1201, 1994.

[3] C. J. C. H. Watkins, ''Learning from Delayed Rewards,'' Ph.D. dissertation, University of Cambridge, 1989.

Consider estimating the expected value obtained when rolling a fair six-sided die. Below is a plot of the incremental estimates using different learning rate functions. As we can see, convergence is not guaranteed if $\alpha(m)$ decays too quickly, and is slow if $\alpha(m)$ does not decay quickly enough.

For constant values of $\alpha \in (0, 1]$, the mean estimate will continue to fluctuate. Larger values of constant α fluctuate wildly, whereas lower values take longer to converge.



Example 17.1. The effect of decaying the learning rate with functions of the form $\alpha(m) = 1/m^{\gamma}$.

```
mutable struct IncrementalEstimate
    μ # mean estimate
    α # learning rate function
    m # number of updates
end

function update!(model::IncrementalEstimate, x)
    model.m += 1
    model.μ += model.α(model.m) * (x - model.μ)
    return model
end
```

Algorithm 17.1. A type for maintaining an incremental estimate of the mean of a random variable. The associated type maintains a current mean value `μ`, a learning rate function `α`, and an iteration count `m`. Calling `update!` with a new value `x` updates the estimate.

We can use equation (17.6) to produce an incremental update rule to estimate the action value function:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right) \tag{17.10}$$

Our choice of actions affects which states we end up in, and therefore our ability to accurately estimate $Q(s,a)$. In order to guarantee convergence of our action value function, we need to adopt some form of exploration policy, such as ϵ-greedy or softmax, just as we did for our model-based methods in the previous chapter. Example 17.2 shows how to run a simulation with the Q-learning update rule and an exploration policy. Figure 17.1 illustrates this process on the hex world problem.

```julia
mutable struct QLearning
    𝒮 # state space (assumes 1:nstates)
    𝒜 # action space (assumes 1:nactions)
    γ # discount
    Q # action value function
    α # learning rate
end

lookahead(model::QLearning, s, a) = model.Q[s,a]

function update!(model::QLearning, s, a, r, s′)
    γ, Q, α = model.γ, model.Q, model.α
    Q[s,a] += α*(r + γ*maximum(Q[s′,:]) - Q[s,a])
    return model
end
```

Algorithm 17.2. The Q-learning update for model-free reinforcement learning, which can be applied to problems with unknown transition and reward functions. The update modifies `Q`, which is a matrix of state-action values. This update function can be used together with an exploration strategy, such as ϵ-greedy in the simulate function in algorithm 15.9.

17.3 *Sarsa*

Sarsa (algorithm 17.3) is an alternative to Q-learning.[4] It derives its name from the fact that it uses (s,a,r,s',a') to update the Q function at each step. It uses the actual next action a' to update Q instead of maximizing over all possible actions:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\big(r + \gamma Q(s',a') - Q(s,a)\big) \tag{17.11}$$

When run with a suitable exploration strategy, the a' above will converge to $\arg\max_{a'} Q(s',a')$, which is what is used in the Q-learning update.

[4] This approach was suggested by a different name in G. A. Rummery and M. Niranjan, "On-Line Q-Learning Using Connectionist Systems," Cambridge University, Tech. Rep. CUED/F-INFENG/TR 166, 1994.

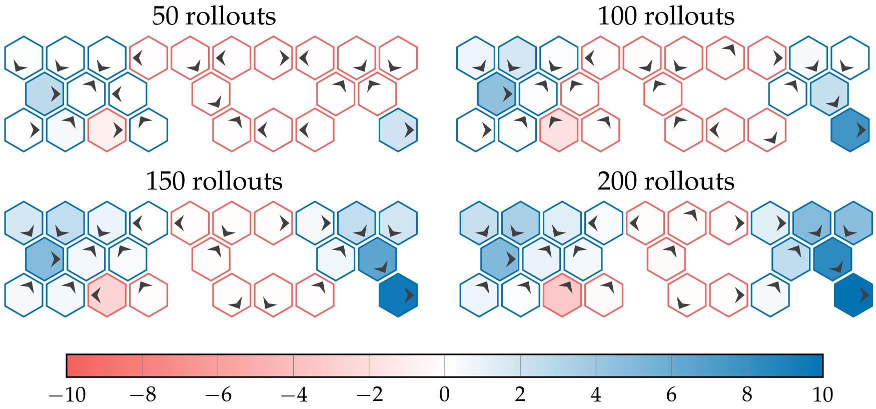


Figure 17.1. *Q*-learning used to iteratively learn an action value function for the hex world problem. Each state is colored according to the expected value of the best action in that state according to *Q*. Actions are similarly the best expected actions. *Q*-learning was run with $\alpha = 0.1$ and 10 steps per rollout.

Example 17.2. An example of how to use an exploration strategy with *Q*-learning in simulation. The parameter settings are notional.

Suppose we want to apply *Q*-learning to an MDP problem $\mathcal{P}$. We can construct an exploration policy, such as the ϵ-greedy policy implemented in algorithm 16.3 from the previous chapter. The *Q*-learning model comes from algorithm 17.2 and the simulate function is implemented in algorithm 15.9.

```
Q = zeros(length(𝒫.𝒮), length(𝒫.𝒜))
α = 0.2 # learning rate
model = QLearning(𝒫.𝒮, 𝒫.𝒜, 𝒫.γ, Q, α)
ϵ = 0.1 # probability of random action
α = 1.0 # exploration decay factor
π = EpsilonGreedyExploration(ϵ, α)
k = 20 # number of steps to simulate
s = 1  # initial state
simulate(𝒫, model, π, k, s)
```

Sarsa is referred to as a type of *on-policy* reinforcement learning method because it attempts to directly estimate the value of the exploration policy as it follows it. In contrast, Q-learning is an *off-policy* method because it attempts to find the value of the optimal policy while following the exploration strategy. Although Q-learning and Sarsa both converge to an optimal strategy, the speed of convergence depends on the application. Sarsa is run on the hex world problem in figure 17.2.

```
mutable struct Sarsa
    𝒮 # state space (assumes 1:nstates)
    𝒜 # action space (assumes 1:nactions)
    γ # discount
    Q # action value function
    α # learning rate
    ℓ # last experience tuple (s,a,r)
end

lookahead(model::Sarsa, s, a) = model.Q[s,a]

function update!(model::Sarsa, s, a, r, s′)
    if model.ℓ != nothing
        γ, Q, α, ℓ = model.γ, model.Q, model.α,  model.ℓ
        model.Q[ℓ.s,ℓ.a] += α*(ℓ.r + γ*Q[s,a] - Q[ℓ.s,ℓ.a])
    end
    model.ℓ = (s=s, a=a, r=r)
    return model
end
```

Algorithm 17.3. The Sarsa update for model-free reinforcement learning. We update the matrix `Q` containing the state-action values, `α` is a constant learning rate, and `ℓ` is the last experience tuple.

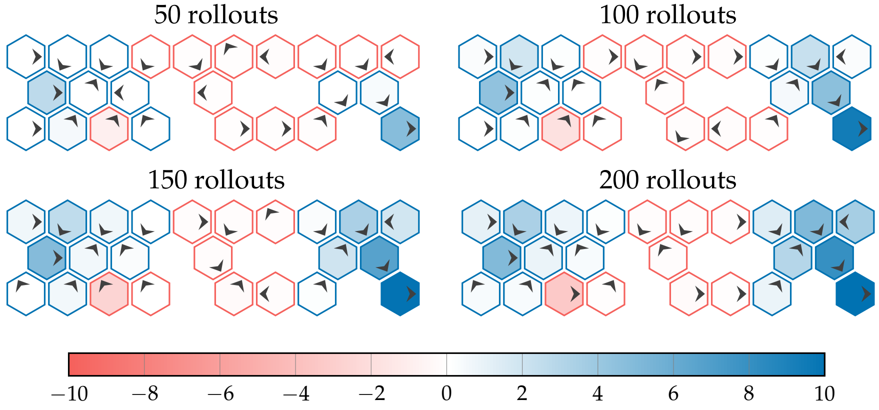


Figure 17.2. Sarsa used to iteratively learn an action value function for the hex world problem in a manner otherwise identical to figure 17.1. We find that Sarsa is slower to converge to the true action value function.

17.4 *Eligibility Traces*

One of the disadvantages of Q-learning and Sarsa is that learning can be very slow, especially with *sparse rewards*. For example, suppose the environment has a single goal state that provides a large reward, and the reward is zero at all other states. After some amount of random exploration in the environment, we reach the goal state. Regardless of whether we use Q-learning or Sarsa, we only update the action value of the state immediately preceding the goal state. The values at all other states leading up to the goal remain at zero. A large amount of exploration is required to slowly propagate non-zero values to the remainder of the state space.

Q-learning and Sarsa can be modified to propagate reward backward to the states and actions leading to the source of the reward using *eligibility traces*.[5] The credit is decayed exponentially so that states closer to the reward are assigned larger values. It is common to use $0 \leq \lambda \leq 1$ as the exponential decay parameter. Versions of Q-learning and Sarsa with eligibility traces are often called $\text{Q}(\lambda)$ and $\text{Sarsa}(\lambda)$.[6]

A version of $\text{Sarsa}(\lambda)$ is implemented in algorithm 17.4. The algorithm maintains an exponentially decaying visit count $N(s, a)$ for all state-action pairs. When action a is taken in state s, $N(s, a)$ is incremented by 1. The Sarsa temporal difference update is then partially applied to every state-action pair according to this decaying visit count.

Let δ denote the Sarsa temporal difference update:

$$\delta = r + \gamma Q(s', a') - Q(s, a) \tag{17.12}$$

Every entry in the action value function is then updated according to:

$$Q(s, a) \leftarrow Q(s, a) + \alpha\delta N(s, a) \tag{17.13}$$

The visit counts are then decayed using both the discount factor and the exponential decay parameter:

$$N(s, a) \leftarrow \gamma\lambda N(s, a) \tag{17.14}$$

Although the impact of eligibility traces is especially pronounced in environments with sparse reward, the algorithm can speed learning in general environments where reward is more distributed.

[5] Eligibility traces were proposed in the context of temporal difference learning by R. Sutton, ''Learning to Predict by the Methods of Temporal Differences,'' *Machine Learning*, vol. 3, no. 1, pp. 9–44, 1988.

[6] These algorithms were introduced by C. J. C. H. Watkins, ''Learning from Delayed Rewards,'' Ph.D. dissertation, University of Cambridge, 1989. and J. Peng and R. J. Williams, ''Incremental Multi-Step Q-Learning,'' *Machine Learning*, vol. 22, no. 1–3, pp. 283–290, 1996.

```
mutable struct SarsaLambda
    𝒮 # state space (assumes 1:nstates)
    𝒜 # action space (assumes 1:nactions)
    γ # discount
    Q # action value function
    N # trace
    α # learning rate
    λ # trace decay rate
    ℓ # last experience tuple (s,a,r)
end

lookahead(model::SarsaLambda, s, a) = model.Q[s,a]

function update!(model::SarsaLambda, s, a, r, s′)
    if model.ℓ != nothing
        γ, λ, Q, α, ℓ = model.γ, model.λ, model.Q, model.α, model.ℓ
        model.N[ℓ.s,ℓ.a] += 1
        δ = ℓ.r + γ*Q[s,a] - Q[ℓ.s,ℓ.a]
        for s in model.𝒮
            for a in model.𝒜
                model.Q[s,a] += α*δ*model.N[s,a]
                model.N[s,a] *= γ*λ
            end
        end
    else
        model.N[:,:] .= 0.0
    end
    model.ℓ = (s=s, a=a, r=r)
    return model
end
```

Algorithm 17.4. The Sarsa(λ) update, which uses eligibility traces to propagate reward back in time to speed learning of sparse rewards. The matrix `Q` contains the state-action values, the matrix `N` contains exponentially decaying state-action visit counts, `α` is a constant learning rate, `λ` is an exponential decay parameter, and `ℓ` is the last experience tuple.

Special care must be taken when applying eligibility traces to an off-policy algorithm like Q-learning that attempts to learn the value of the optimal policy.[7] Eligibility traces propagate back values obtained from an exploration policy. This mismatch can result in learning instabilities.

17.5 *Reward Shaping*

Especially in problems where the reward function is sparse, we may want to augment the reward function to help guide our learning. For example, if we are trying to reach a single goal state, we could supplement the reward function by an amount inversely proportional to the distance to the goal. Or, alternatively, we could add an additional penalty based on how far away we are from the goal. If we are playing chess, we might add a penalty to our reward function when we lose a piece, even though we only care about winning or losing the game at the end and not about individual pieces.

Modifications to the reward function during training by incorporating domain knowledge to speed training is known as *reward shaping*. Suppose rewards in our problem are generated according to $R(s, a, s')$, allowing rewards to depend on the resulting state. We will use $F(s, a, s')$ to represent our *shaping function*. During training, instead of using $R(s, a, s')$ as our reward, we use $R(s, a, s') + F(s, a, s')$.

Adding $F(s, a, s')$ to our reward can, of course, change the optimal policy. We are often interested in shaping reward without changing what is optimal. It turns out that a policy that is optimal under the original reward remains optimal under the shaped reward if and only if

$$F(s, a, s') = \gamma\beta(s') - \beta(s) \tag{17.15}$$

for some potential function $\beta(s)$.[8]

17.6 *Action Value Function Approximation*

The algorithms we have discussed so far in this chapter have assumed discrete state and action spaces where the action value function can be stored in a lookup table. We can adapt our algorithms to use global value function approximation, allowing us to apply them to problems with large or continuous spaces and generalize from limited experience. Similar to the approach taken in chapter 8

[7] For an overview of this problem and a potential solution, see A. Harutyunyan, M. G. Bellemare, T. Stepleton, and R. Munos, "Q(λ) with Off-Policy Corrections," in *International Conference on Algorithmic Learning Theory*, 2016.

[8] A. Y. Ng, D. Harada, and S. Russell, "Policy Invariance Under Reward Transformations: Theory and Application to Reward Shaping," in *International Conference on Machine Learning (ICML)*, 1999.

in the context of a known model, we will use $Q_{\boldsymbol{\theta}}(s, a)$ to represent a parametric approximation of our action value function when the model is unknown.[9]

To illustrate the concept, we will derive a version of Q-learning that uses our parametric approximation. We want to minimize the loss between our approximation and the optimal action value function $Q^*(s, a)$, which we define to be[10]

$$\ell(\boldsymbol{\theta}) = \frac{1}{2} \mathop{\mathbb{E}}_{(s,a)\sim\pi^*}\left[\left(Q^*(s, a) - Q_{\boldsymbol{\theta}}(s, a)\right)^2\right] \tag{17.16}$$

The expectation is over the state-action pairs experienced when following the optimal policy π^*.

A common approach to minimizing this loss is to use some form of gradient descent. The gradient of the loss is

$$\nabla\ell(\boldsymbol{\theta}) = -\mathop{\mathbb{E}}_{(s,a)\sim\pi^*}\left[(Q^*(s, a) - Q_{\boldsymbol{\theta}}(s, a))\nabla_{\boldsymbol{\theta}} Q_{\boldsymbol{\theta}}(s, a)\right] \tag{17.17}$$

We typically choose parametric representations of the action value function that are differentiable and where $\nabla_{\boldsymbol{\theta}} Q_{\boldsymbol{\theta}}(s, a)$ is easy to compute, such as linear or neural network representations. If we apply gradient descent,[11] our update rule is

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \mathop{\mathbb{E}}_{(s,a)\sim\pi^*}\left[(Q^*(s, a) - Q_{\boldsymbol{\theta}}(s, a))\nabla_{\boldsymbol{\theta}} Q_{\boldsymbol{\theta}}(s, a)\right] \tag{17.18}$$

where α is our step factor or learning rate. We can approximate the update rule above using samples of our state-action (s, a) pairs as we experience them:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha(Q^*(s, a) - Q_{\boldsymbol{\theta}}(s, a))\nabla_{\boldsymbol{\theta}} Q_{\boldsymbol{\theta}}(s, a) \tag{17.19}$$

Of course, we cannot compute equation (17.19) directly because that would require knowing the optimal policy, which is precisely what we are attempting to find. Instead, we attempt to estimate it from our observed transition and our action value approximation:

$$Q^*(s, a) \approx r + \gamma \max_{a'} Q_{\boldsymbol{\theta}}(s', a') \tag{17.20}$$

which results in the following update rule:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha(r + \gamma \max_{a'} Q_{\boldsymbol{\theta}}(s', a') - Q_{\boldsymbol{\theta}}(s, a))\nabla_{\boldsymbol{\theta}} Q_{\boldsymbol{\theta}}(s, a) \tag{17.21}$$

[9] In recent years, a major focus has been on *deep reinforcement learning*, where deep neural networks are used for this parametric approximation. A discussion of practical implementations is provided by L. Graesser and W. L. Keng, *Foundations of Deep Reinforcement Learning*. Addison Wesley, 2020.

[10] The $1/2$ in the front is for convenience because we will later be computing the derivative of this quadratic.

[11] We want to descend rather than ascend because we are trying to minimize our loss.

This update is implemented in algorithm 17.5 with the addition of a scaled gradient step (algorithm 12.2), which is often needed to ensure that the gradient steps do not become too large. Example 17.3 shows how to use this update with a linear action value approximation. Figure 17.3 demonstrates this algorithm on the mountain car problem.

```
struct GradientQLearning
    𝒜  # action space (assumes 1:nactions)
    γ  # discount
    Q  # parameterized action value function Q(θ,s,a)
    ∇Q # gradient of action value function
    θ  # action value function parameter
    α  # learning rate
end

function lookahead(model::GradientQLearning, s, a)
    return model.Q(model.θ, s,a)
end

function update!(model::GradientQLearning, s, a, r, s′)
    𝒜, γ, Q, θ, α = model.𝒜, model.γ, model.Q, model.θ, model.α
    u = maximum(Q(θ,s′,a′) for a′ in 𝒜)
    Δ = (r + γ*u - Q(θ,s,a))*model.∇Q(θ,s,a)
    θ[:] += α*scale_gradient(Δ, 1)
    return model
end
```

Algorithm 17.5. The Q-learning update with action value function approximation. With each new experience tuple `s`, `a`, `r`, `s′`, we update our vector `θ` with constant learning rate `α`. Our parameterized action value function is given by `Q(θ,s,a)` and its gradient is `∇Q(θ,s,a)`.

17.7 *Experience Replay*

A major challenge of using global function approximation with reinforcement learning is *catastrophic forgetting*. For example, we might initially discover that our particular policy brings us to a low-reward region of the state space. We then refine our policy to avoid that area. However, after some amount of time, we may forget why it was important to avoid that region of the state space, and we may risk reverting back to a poorly performing policy.

Catastrophic forgetting can be mitigated with *experience replay*,[12] where a fixed number of the most recent experience tuples are stored across training iterations. A *batch* of tuples are sampled uniformly from this *replay memory* to remind us to avoid strategies that we have already discovered are poor.[13] The update equation

[12] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, et al., "Playing Atari with Deep Reinforcement Learning," 2013. arXiv: `1312.5602v1`.

[13] Variations of this approach include prioritizing experiences. T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized Experience Replay," in *International Conference on Learning Representations (ICLR)*, 2016.

We are interesting in applying Q-learning with a linear action value approximation to the simple regulator problem with $\gamma = 1$. Our action value approximation is $Q_{\boldsymbol{\theta}}(s,a) = \boldsymbol{\theta}^\top \boldsymbol{\beta}(s,a)$, where our basis function is

$$\boldsymbol{\beta}(s,a) = [s, s^2, a, a^2, 1]$$

With this linear model,

$$\nabla_{\boldsymbol{\theta}} Q_{\boldsymbol{\theta}}(s,a) = \boldsymbol{\beta}(s,a)$$

We can implement this as follows for problem 𝒫:

```
β(s,a) = [s,s^2,a,a^2,1]
Q(θ,s,a) = dot(θ,β(s,a))
∇Q(θ,s,a) = β(s,a)
θ = [0.1,0.2,0.3,0.4,0.5] # initial parameter vector
α = 0.5 # learning rate
model = GradientQLearning(𝒫.𝒜, 𝒫.γ, Q, ∇Q, θ, α)
ϵ = 0.1 # probability of random action
α = 1.0 # exploration decay factor
π = EpsilonGreedyExploration(ϵ, α)
k = 20  # number of steps to simulate
s = 0.0 # initial state
simulate(𝒫, model, π, k, s)
```

Example 17.3. An example of how to use an exploration strategy with Q-learning with action value function approximation in simulation. The parameter settings are notional.

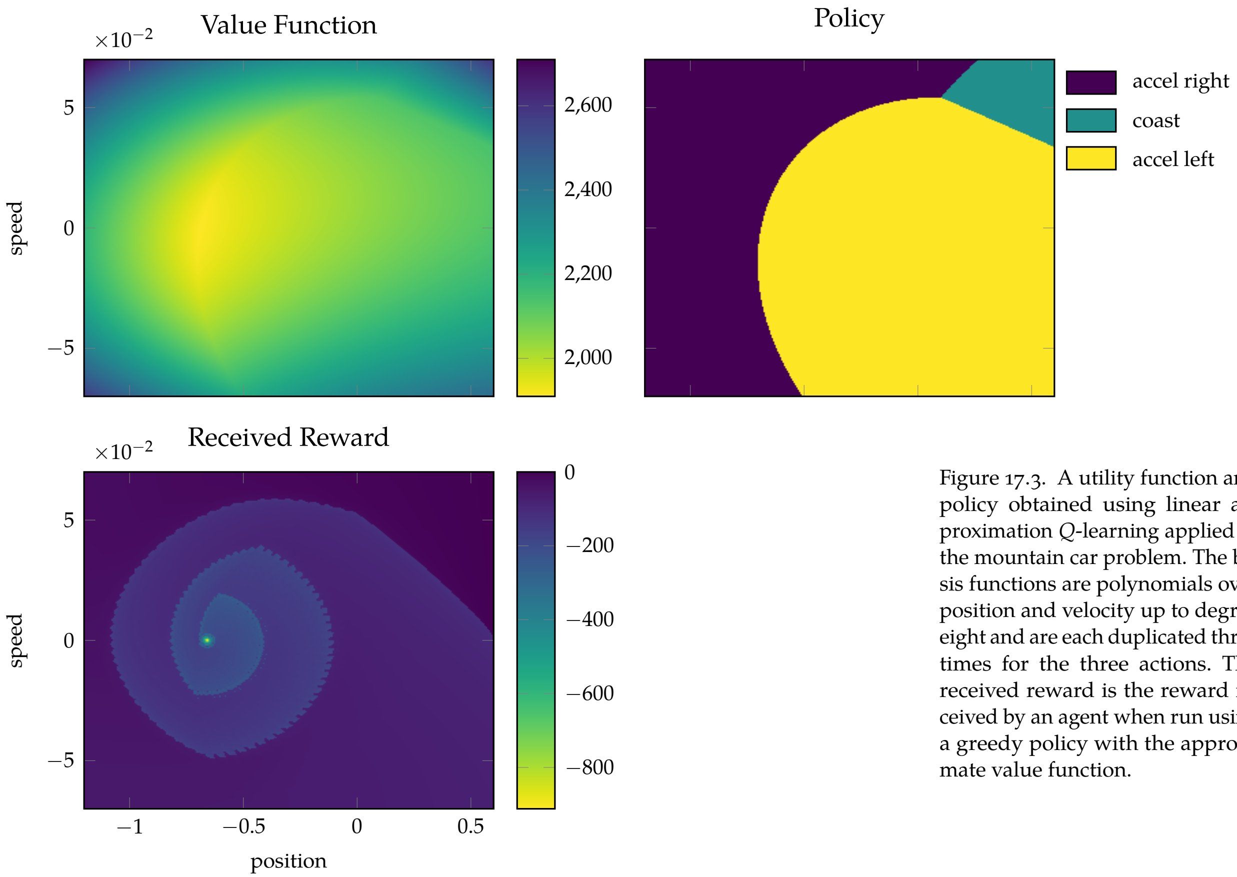


Figure 17.3. A utility function and policy obtained using linear approximation *Q*-learning applied to the mountain car problem. The basis functions are polynomials over position and velocity up to degree eight and are each duplicated three times for the three actions. The received reward is the reward received by an agent when run using a greedy policy with the approximate value function.

from equation (17.21) is modified to become

$$\theta \leftarrow \theta + \alpha \frac{1}{m_{\text{grad}}} \sum_i (r^{(i)} + \gamma \max_{a'} Q_\theta(s'^{(i)}, a') - Q_\theta(s^{(i)}, a^{(i)})) \nabla_\theta Q_\theta(s^{(i)}, a^{(i)}) \tag{17.22}$$

where $s^{(i)}, a^{(i)}, r^{(i)}, s'^{(i)}$ is the ith experience tuple in a random batch of size m_{grad}.

Experience replay allows experience tuples to contribute to learning multiple times, thereby increasing data efficiency. Furthermore, sampling uniformly at random from the replay memory breaks apart otherwise correlated sequences that are obtained from rollouts, thereby reducing the variance of the gradient estimate. Experience replay stabilizes the learning process by retaining information from previous policy parameterizations.

Algorithm 17.6 shows how to incorporate experience replay into Q-learning with action value function approximation. Example 17.4 shows how to apply this approach to a simple regulator problem.

17.8 Summary

- Model-free methods seek to directly learn an action value function rather than transition and reward models.

- Simple techniques can be used to incrementally learn a mean from sequential updates.

- The Q-learning algorithm incrementally learns an action value function using an approximation of the Bellman equation.

- In contrast with Q-learning, Sarsa uses the action taken by the exploration policy rather than maximizing over all subsequent actions in its update.

- Eligibility traces can speed learning by propagating sparse rewards through the state-action space.

- Q-learning can be applied to approximate value functions using stochastic gradient descent.

- The catastrophic forgetting experienced by Q-learning and Sarsa can be mitigated using experience replay, which reuses past experience tuples.

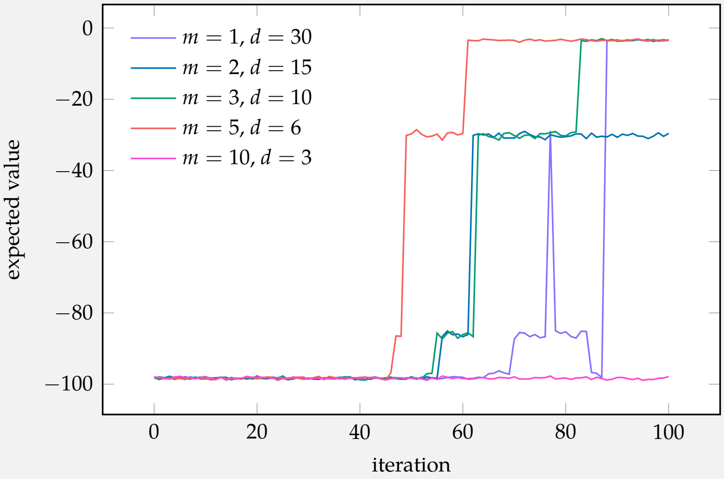
```
struct ReplayGradientQLearning
    𝒜      # action space (assumes 1:nactions)
    γ      # discount
    Q      # parameterized action value funciton Q(θ,s,a)
    ∇Q     # gradient of action value function
    θ      # action value function parameter
    α      # learning rate
    buffer # circular memory buffer
    m      # number of steps between gradient updates
    m_grad # batch size
end

function lookahead(model::ReplayGradientQLearning, s, a)
    return model.Q(model.θ, s,a)
end

function update!(model::ReplayGradientQLearning, s, a, r, s′)
    𝒜, γ, Q, θ, α = model.𝒜, model.γ, model.Q, model.θ, model.α
    buffer, m, m_grad = model.buffer, model.m, model.m_grad
    if isfull(buffer)
        U(s) = maximum(Q(θ,s,a) for a in 𝒜)
        ∇Q(s,a,r,s′) = (r + γ*U(s′) - Q(θ,s,a))*model.∇Q(θ,s,a)
        Δ = mean(∇Q(s,a,r,s′) for (s,a,r,s′) in rand(buffer, m_grad))
        θ[:] += α*scale_gradient(Δ, 1)
        for i in 1:m # discard oldest experiences
            popfirst!(buffer)
        end
    else
        push!(buffer, (s,a,r,s′))
    end
    return model
end
```

Algorithm 17.6. Q-learning with function approximation and experience replay. The update depends upon a parameterized policy `Q(θ,s,a)` and gradient `∇Q(θ,s,a)`. It updates the parameter vector `θ` and the circular memory buffer provided by `DataStructures.jl`. It updates `θ` every `m` steps using a gradient estimated from `m_grad` samples from the buffer.

Suppose we want to add experience replay to example 17.3. When constructing the model, we need to provide a replay buffer with the desired capacity:

```
capacity = 100 # maximum size of the replay buffer
ExperienceTuple = Tuple{Float64,Float64,Float64,Float64}
M = CircularBuffer{ExperienceTuple}(capacity) # replay buffer
m_grad = 20 # batch size
model = ReplayGradientQLearning(𝒫.𝒜, 𝒫.γ, Q, ∇Q, θ, α, M, m, m_grad)
```

We can vary the number of steps between gradient updates m and the depth of each simulation d. In the plot below, we limit all training runs to $md = 30$ experience tuples with each iteration. The plot indicates that rollouts to a sufficient depth are necessary for training to succeed. In addition, very few rollouts to an excessive depth do not perform as well as a moderate number of rollouts to a moderate depth.



Example 17.4. Application of experience replay to the simple regulator problem with Q-learning and action value approximation.

17.9 Exercises

Exercise 17.1. Given the following set of samples, perform incremental estimation of the mean twice; once using a learning rate of $\alpha = 0.1$ and once using a learning rate of $\alpha = 0.5$. In both, use an initial mean equal to the first sample.

$$x^{(1:5)} = \{1.0, 1.8, 2.0, 1.6, 2.2\}$$

Solution: We set the mean at the first iteration equal to the first sample and proceed to incrementally estimate the mean using equation (17.6):

$$\begin{aligned}
\hat{x}_1 &= 1.0 & \hat{x}_1 &= 1.0\\
\hat{x}_2 &= 1.0 + 0.1(1.8 - 1.0) = 1.08 & \hat{x}_2 &= 1.0 + 0.5(1.8 - 1.0) = 1.4\\
\hat{x}_3 &= 1.08 + 0.1(2.0 - 1.08) = 1.172 & \hat{x}_3 &= 1.4 + 0.5(2.0 - 1.4) = 1.7\\
\hat{x}_4 &= 1.172 + 0.1(1.6 - 1.172) \approx 1.215 & \hat{x}_4 &= 1.7 + 0.5(1.6 - 1.7) = 1.65\\
\hat{x}_5 &= 1.215 + 0.1(2.2 - 1.215) \approx 1.313 & \hat{x}_5 &= 1.65 + 0.5(2.2 - 1.65) = 1.925
\end{aligned}$$

Exercise 17.2. Following the previous exercise, suppose once we have estimated the mean with five samples for both methods, we are provided with a single additional sample $x^{(6)}$ that we will use as the final sample in estimating our mean. Which of the two incremental estimation methods (i.e., $\alpha = 0.1$ or $\alpha = 0.5$) would be preferable?

Solution: While we do not know what the sample would be or what the underlying mean of the process is, we would likely prefer the second incrementally estimated mean that uses $\alpha = 0.5$. Since we only have one sample left, the first learning rate is too small to considerably change the mean, while the second learning rate is large enough to be responsive, without neglecting the past samples. Consider two cases:

1. If we assume the next sample is approximately equal to the *incremental* mean of all previous samples, then we have $x^{(6)} \approx \hat{x}_5$. Thus, performing an incremental update of the mean yields no change to our estimate. We have $\hat{x}_6 \approx 1.313$ for a learning rate of 0.1, and we have $\hat{x}_6 = 1.925$ for a learning rate of 0.5.
2. If we assume the next sample is approximately equal to the *exact* mean of all previous samples, then we have $x^{(6)} \approx 1.72$. The update using a learning rate of 0.1 yields $\hat{x}_6 \approx 1.354$, while the update using a learning rate of 0.5 yields $\hat{x}_6 \approx 1.823$.

In both of these cases, supposing the next sample is equal to the mean all previous samples, then the estimate using a learning rate of 0.5 is more accurate.

Exercise 17.3. Using incremental estimation of the mean, derive an update rule for a value function $U(s)$ for experience tuples of the form (s, r, s'). Is the resulting algorithm model-based or model-free?

Solution: The incremental estimation of the mean equation is

$$\hat{x} \leftarrow \hat{x} + \alpha(x - \hat{x})$$

Our value function can be written in the familiar form

$$U(s) = \max_a \left(R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U(s') \right)$$

Applying incremental estimation to the value function where we receive experience tuples of the form (s_t, r_t, s_{t+1}), we can derive an update rule for the value function.

$$\begin{aligned}
U(s) &\leftarrow U(s) + \alpha \left[\max_a \left(R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U(s') \right) - U(s) \right] \\
U(s) &\leftarrow U(s) + \alpha \left[\max_a \left(r + \gamma U(s') \right) - U(s) \right] \\
U(s) &\leftarrow U(s) + \alpha \left(r + \gamma U(s') - U(s) \right)
\end{aligned}$$

This algorithm is known as *temporal difference* (*TD*) learning. While the update rule is model-free, extracting an action or policy requires a transition and reward model:

$$a^* = \pi^*(s) = \arg\max_a \left(R(s,a) + \gamma \sum_{s'} T(s' \mid s,a) U(s) \right)$$

Exercise 17.4. What is the complexity of Q-learning if we interact with the environment for d time steps? What is the complexity of Sarsa if we interact with the environment for d time steps?

Solution: For Q-learning, our update rule is:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right)$$

At each time step, we must perform a maximization over actions, so for d time steps, the complexity of Q-learning is $O(d|\mathcal{A}|)$. For Sarsa, our update rule is:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma Q(s',a') - Q(s,a) \right) \tag{17.23}$$

At each time step, unlike Q-learning, we do not have to perform a maximization over actions, so for d time steps, the complexity of Sarsa is simply $O(d)$.

Exercise 17.5. Is the computational complexity of Sarsa per experience tuple (s_t, a_t, r_t, s_{t+1}) more or less than that of Sarsa(λ)?

Solution: For Sarsa, our update rule is:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma Q(s',a') - Q(s,a) \right) \tag{17.24}$$

So, for each experience tuple, we have $O(1)$ complexity. For Sarsa(λ), our update rules are:

$$\begin{aligned} \delta &\leftarrow r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \\ N(s_t, a_t) &\leftarrow N(s_t, a_t) + 1 \\ Q(s, a) &\leftarrow Q(s, a) + \alpha \delta N(s, a) \qquad \text{for all } s, a \\ N(s, a) &\leftarrow \gamma \lambda N(s, a) \qquad \text{for all } s, a \end{aligned}$$

For each experience tuple, we need to compute δ and increment the visit count at (s_t, a_t), which are both $O(1)$. However, we need to update both the action value function and the visit counts for all states and actions, which are both $O(|\mathcal{S}||\mathcal{A}|)$. Thus, the computational complexity per experience tuple is greater for Sarsa(λ). However, Sarsa(λ) often converges using fewer experience tuples.

Exercise 17.6. What is the behavior of $Q(\lambda)$ in the limit as $\lambda \to 0$? What is the behavior of $Q(\lambda)$ in the limit as $\lambda \to 1$?

Solution: For $Q(\lambda)$, we perform the following update rules:

$$\begin{aligned} \delta &\leftarrow r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \\ N(s_t, a_t) &\leftarrow N(s_t, a_t) + 1 \\ Q(s, a) &\leftarrow Q(s, a) + \alpha \delta N(s, a) \qquad \text{for all } s, a \\ N(s, a) &\leftarrow \gamma \lambda N(s, a) \qquad \text{for all } s, a \end{aligned}$$

In the limit as $\lambda \to 0$, for our first iteration, we compute the temporal difference error δ and we increment the visit count $N(s_t, a_t)$. In the action value function update, the only non-zero $N(s, a)$ is at $N(s_t, a_t)$, so we perform $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \delta N(s_t, a_t)$. Finally, we reset all the visit counts to zero. From this, we can see that in the limit as $\lambda \to 0$, we have no eligibility traces and we are performing a straightforward Q-learning update.

In the limit as $\lambda \to 1$, our visit counts will accumulate and we have full eligibility traces, which will spread the reward the most possible over previously visited state-action pairs.

Exercise 17.7. Compute $Q(s, a)$ using Sarsa(λ) after following the trajectory

$$(s_1, a_R, 0, s_2, a_R, 0, s_3, a_L, 10, s_2, a_R, 4, s_1, a_R)$$

Use $\alpha = 0.5$, $\lambda = 1$, $\gamma = 0.9$, and initial action value function and visit counts equal to zero everywhere. Assume $\mathcal{S} = \{s_1, s_2, s_3, s_4\}$ and $\mathcal{A} = \{a_L, a_R\}$.

Solution: The Sarsa(λ) update rules are

$$\begin{aligned} \delta &\leftarrow r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \\ N(s_t, a_t) &\leftarrow N(s_t, a_t) + 1 \\ Q(s, a) &\leftarrow Q(s, a) + \alpha \delta N(s, a) \qquad \text{for all } s, a \\ N(s, a) &\leftarrow \gamma \lambda N(s, a) \qquad \text{for all } s, a \end{aligned}$$

For the first experience tuple, we have $\delta = 0 + 0.9 \times 0 - 0 = 0$, we increment the visit count at $N(s_1, a_R)$, the action value function does not change since $\delta = 0$, and we update our counts. After completing this, we have:

$Q(s,a)$	s_1	s_2	s_3	s_4
a_L	0	0	0	0
a_R	0	0	0	0

$N(s,a)$	s_1	s_2	s_3	s_4
a_L	0	0	0	0
a_R	0.9	0	0	0

For the second experience tuple, we have $\delta = 0$, we increment the visit count at $N(s_2, a_R)$, the action value function does not change since $\delta = 0$, and we update our counts. After completing this, we have:

$Q(s,a)$	s_1	s_2	s_3	s_4
a_L	0	0	0	0
a_R	0	0	0	0

$N(s,a)$	s_1	s_2	s_3	s_4
a_L	0	0	0	0
a_R	0.81	0.9	0	0

For the third experience tuple, we have $\delta = 10$, we increment the visit count at $N(s_3, a_L)$, we update the action value function, and we update our counts. After completing this, we have:

$Q(s,a)$	s_1	s_2	s_3	s_4
a_L	0	0	5	0
a_R	4.05	4.5	0	0

$N(s,a)$	s_1	s_2	s_3	s_4
a_L	0	0	0.9	0
a_R	0.729	0.81	0	0

For the fourth experience tuple, we have $\delta = 4 + 0.9 \times 4.05 - 4.5 = 3.145$, we increment the visit count at $N(s_2, a_R) = 0.81 + 1 = 1.81$, we update the action value function, and we update our counts. After completing this, we have:

$Q(s,a)$	s_1	s_2	s_3	s_4
a_L	0	0	6.415	0
a_R	5.196	7.346	0	0

$N(s,a)$	s_1	s_2	s_3	s_4
a_L	0	0	0.81	0
a_R	0.656	1.629	0	0