10 Policy Search

Policy search involves searching the space of policies without directly computing a value function. The policy space is often lower-dimensional than the state space and can often be searched more efficiently. Policy optimization optimizes the parameters in a *parameterized policy* in order to maximize utility. This parameterized policy can take many forms, such as neural networks, decision trees, and computer programs. This chapter begins by discussing a way to estimate the value of a policy given an initial state distribution. We will then discuss search methods that do not use estimates of the gradient of the policy, saving gradient methods for the next chapter. Although local search can be quite effective in practice, we will also discuss a few alternative optimization approaches that can avoid local optima.[1]

[1] There are many other optimization approaches, as discussed by M. J. Kochenderfer and T. A. Wheeler, *Algorithms for Optimization*. MIT Press, 2019.

10.1 Approximate Policy Evaluation

As introduced in section 7.2, we can compute the expected discounted return when following a policy π from a state s. This expected discounted return $U^\pi(s)$ can be computed iteratively (algorithm 7.3) or through matrix operations (algorithm 7.4) when the state space is discrete and relatively small. We can use these results to compute the expected discounted return of π:

$$U(\pi) = \sum_s b(s) U^\pi(s) \tag{10.1}$$

assuming an *initial state distribution* $b(s)$.

We will use this definition of $U(\pi)$ throughout this chapter. However, we often cannot compute $U(\pi)$ exactly when the state space is large or continuous. Instead, we can approximate $U(\pi)$ by sampling *trajectories*, consisting of states, actions, and rewards when following π. The definition of $U(\pi)$ can be rewritten as

$$U(\pi) = \mathbb{E}_\tau[R(\tau)] = \int p_\pi(\tau) R(\tau)\,\mathrm{d}\tau \tag{10.2}$$

where $p_\pi(\tau)$ is the probability density associated with trajectory τ when following policy π, starting from initial state distribution b. The *trajectory reward* $R(\tau)$ is the discounted return associated with τ. Figure 10.1 illustrates the computation of $U(\pi)$ in terms of trajectories sampled from an initial state distribution.

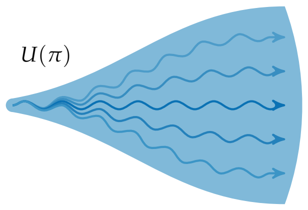


Figure 10.1. The utility associated with a policy from an initial state distribution is computed from the return associated with all possible trajectories under the given policy, weighted according to their likelihood.

Monte Carlo policy evaluation (algorithm 10.1) involves approximating equation (10.2) with m trajectory rollouts of π:

$$U(\pi) \approx \frac{1}{m} \sum_{i=1}^{m} R(\tau^{(i)}) \tag{10.3}$$

where $\tau^{(i)}$ is the ith trajectory sample.

```
struct MonteCarloPolicyEvaluation
    𝒫 # problem
    b # initial state distribution
    d # depth
    m # number of samples
end

function (U::MonteCarloPolicyEvaluation)(π)
    R(π) = rollout(U.𝒫, rand(U.b), π, U.d)
    return mean(R(π) for i = 1:U.m)
end

(U::MonteCarloPolicyEvaluation)(π, θ) = U(s→π(θ, s))
```

Algorithm 10.1. Monte Carlo policy evaluation of a policy `π`. The method runs `m` rollouts to depth `d` according to the dynamics specified by the problem `𝒫`. Each rollout is run from an initial state sampled from state distribution `b`. The final line in this algorithm block evaluates a policy `π` parameterized by `θ`, which will be useful in the algorithms in this chapter that attempt to find a value of `θ` that maximizes `U`.

Monte Carlo policy evaluation is stochastic. Multiple evaluations of equation (10.1) with the same policy can give different estimates. Increasing the number of rollouts decreases the variance of the evaluation, as demonstrated in figure 10.2.

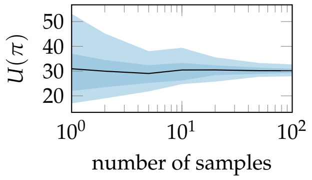


Figure 10.2. The effect of the depth and sample count for Monte Carlo policy evaluation of a uniform random policy on the cart-pole problem (appendix F.3). The variance decreases as the number of samples increases. The blue regions indicate the 5 % to 95 % and 25 % to 75 % empirical quantiles of $U(\pi)$.

We will use π_θ to denote a policy parameterized by θ. For convenience, we will use $U(\theta)$ as shorthand for $U(\pi_\theta)$ in cases where it is not ambiguous. The parameter θ may be a vector or some other more complex representation. For example, we may want to represent our policy using a neural network with a particular structure. We would use θ to represent the weights in the network. Many optimization algorithms assume that θ is a vector with a fixed number of components. Other optimization algorithms allow more flexible representations, including representations like decision trees or computational expressions.[2]

[2] We will not be discussing those representations here, but some are implemented in `ExprOptimization.jl`.

10.2 Local Search

A common approach to optimization is *local search*, where we begin with an initial parameterization and incrementally move from neighbor to neighbor in the search space until convergence occurs. We discussed this type of approach in chapter 5, in the context of optimizing Bayesian network structures with respect to the Bayesian score. Here, we are optimizing policies parameterized by θ. We are trying to find a value of θ that maximizes $U(\theta)$.

There are many local search algorithms, but this section will focus on the *Hooke-Jeeves method* (algorithm 10.2).[3] This algorithm assumes that our policy is parameterized by an n-dimensional vector $\boldsymbol{\theta}$. The algorithm takes a step of size $\pm\alpha$ in each of the coordinate directions from the current $\boldsymbol{\theta}$. These $2n$ points correspond to the neighborhood of $\boldsymbol{\theta}$. If no improvements to the policy are found, then the step size α is decreased by some factor. If an improvement is found, it moves to the best point. The process continues until α drops below some threshold $\epsilon > 0$. An example involving policy optimization is provided in example 10.1, and figure 10.3 illustrates this process.

[3] R. Hooke and T. A. Jeeves, "Direct Search Solution of Numerical and Statistical Problems," *Journal of the ACM (JACM)*, vol. 8, no. 2, pp. 212–229, 1961.

10.3 Genetic Algorithms

A potential issue with local search algorithms like the Hooke-Jeeves method is that the optimization can get stuck in a local optimum. There are a wide variety of approaches that involve maintaining a *population* consisting of samples of points in the parameter space, evaluating them in parallel with respect to our objective, and then recombining them in some way to drive the population toward a global optimum. A *genetic algorithm*[4] is one such approach, which derives inspiration from biological evolution. It is a general optimization method, but it has been successful in the context of optimizing policies. For example, this approach has been used to optimize policies for Atari video games, where the policy parameters correspond to weights in a neural network.[5]

[4] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[5] F. P. Such, V. Madhavan, E. Conti, J. Lehman, K. O. Stanley, and J. Clune, "Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning," 2017. arXiv: 1712.06567v3. The implementation in this section follows their relatively simple formulation. Their formulation does not include crossover, which is typically used to mix parameterizations across a population.

A simple version of this approach (algorithm 10.3) begins with a population of m random parameterizations, $\boldsymbol{\theta}^{(1)}, \ldots, \boldsymbol{\theta}^{(m)}$. We compute $U(\boldsymbol{\theta}^{(i)})$ for each sample

```julia
struct HookeJeevesPolicySearch
    θ # initial parameterization
    α # step size
    c # step size reduction factor
    ϵ # termination step size
end

function optimize(M::HookeJeevesPolicySearch, π, U)
    θ, θ′, α, c, ϵ = copy(M.θ), similar(M.θ), M.α, M.c, M.ϵ
    u, n = U(π, θ), length(θ)
    while α > ϵ
        copyto!(θ′, θ)
        best = (i=0, sgn=0, u=u)
        for i in 1:n
            for sgn in (-1,1)
                θ′[i] = θ[i] + sgn*α
                u′ = U(π, θ′)
                if u′ > best.u
                    best = (i=i, sgn=sgn, u=u′)
                end
            end
            θ′[i] = θ[i]
        end
        if best.i != 0
            θ[best.i] += best.sgn*α
            u = best.u
        else
            α *= c
        end
    end
    return θ
end
```

Algorithm 10.2. Policy search using the Hooke-Jeeves method, which returns a `θ` that has been optimized with respect to `U`. The policy `π` takes as input a parameter `θ` and state `s`. This implementation starts with an initial value of `θ`. The step size `α` is reduced by a factor of `c` if no neighbor improves the objective. Iterations are run until the step size is less than `ϵ`.

Suppose we want to optimize a policy for the simple regulator problem described in appendix F.5. We define a stochastic policy π parameterized by $\boldsymbol{\theta}$ such that the action is generated according to

$$a \sim \mathcal{N}(\theta_1 s, (|\theta_2| + 10^{-5})^2) \tag{10.4}$$

The following code defines the parameterized stochastic policy `π`, evaluation function `U`, and method `M`. It then calls `optimize(M, π, U)`, which returns an optimized value for `θ`. In this case, we use the Hooke-Jeeves method, but the other methods discussed in this chapter can be passed in as `M` instead:

```
function π(θ, s)
    return rand(Normal(θ[1]*s, abs(θ[2]) + 0.00001))
end
b, d, n_rollouts = Normal(0.3,0.1), 10, 3
U = MonteCarloPolicyEvaluation(𝒫, b, d, n_rollouts)
θ, α, c, ϵ = [0.0,1.0], 0.75, 0.75, 0.01
M = HookeJeevesPolicySearch(θ, α, c, ϵ)
θ = optimize(M, π, U)
```

Example 10.1. Using a policy optimization algorithm to optimize the parameters of a stochastic policy.

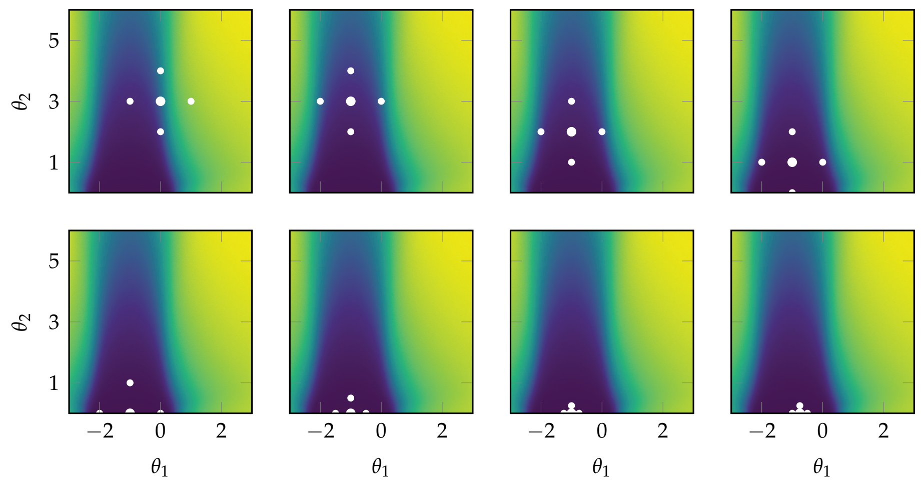


Figure 10.3. The Hooke-Jeeves method applied to optimizing a policy in the simple regulator problem discussed in example 10.1. The evaluations at each iteration are shown as white points. Iterations proceed left to right and top to bottom, and the background is colored according to the expected utility, with yellow indicating lower utility and dark blue indicating higher utility.

i in the population. Since these evaluations potentially involve many rollout simulations and are therefore computationally expensive, they are often run in parallel. These evaluations help us identify the *elite samples*, which are the top m_{elite} samples according to U.

The population at the next iteration is generated by producing $m-1$ new parameterizations by repeatedly selecting a random elite sample $\boldsymbol{\theta}$ and perturbing it with isotropic Gaussian noise, $\boldsymbol{\theta} + \sigma\boldsymbol{\epsilon}$, where $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. The best parameterization, unperturbed, is included as the mth sample. Because the evaluations involve stochastic rollouts, a variation of this algorithm could involve running additional rollouts to help identify which of the elite samples is truly the best. Figure 10.4 shows several iterations, or *generations*, of this approach in a sample problem.

```
struct GeneticPolicySearch
    θs      # initial population
    σ       # initial standard deviation
    m_elite # number of elite samples
    k_max   # number of iterations
end

function optimize(M::GeneticPolicySearch, π, U)
    θs, σ = M.θs, M.σ
    n, m = length(first(θs)), length(θs)
    for k in 1:M.k_max
        us = [U(π, θ) for θ in θs]
        sp = sortperm(us, rev=true)
        θ_best = θs[sp[1]]
        rand_elite() = θs[sp[rand(1:M.m_elite)]]
        θs = [rand_elite() + σ.*randn(n) for i in 1:(m-1)]
        push!(θs, θ_best)
    end
    return last(θs)
end
```

Algorithm 10.3. A genetic policy search method for iteratively updating a population of policy parameterizations `θs`, which takes a policy evaluation function `U`, a policy `π(θ, s)`, a perturbation standard deviation `σ`, an elite sample count `m_elite`, and an iteration count `k_max`. The best `m_elite` samples from each iteration are used to generate the samples for the subsequent iteration.

10.4 Cross Entropy Method

The *cross entropy method* (algorithm 10.4) involves updating a *search distribution* over the parameterized space of policies at each iteration.[6] We parameterize this search distribution $p(\theta \mid \psi)$ with ψ.[7] This distribution can belong to any family, but a Gaussian distribution is a common choice, where ψ represents the mean and

[6] S. Mannor, R. Y. Rubinstein, and Y. Gat, "The Cross Entropy Method for Fast Policy Search," in *International Conference on Machine Learning (ICML)*, 2003.

[7] Often, θ and ψ are vectors, but because this assumption is not required for this method, we will not bold them in this section.

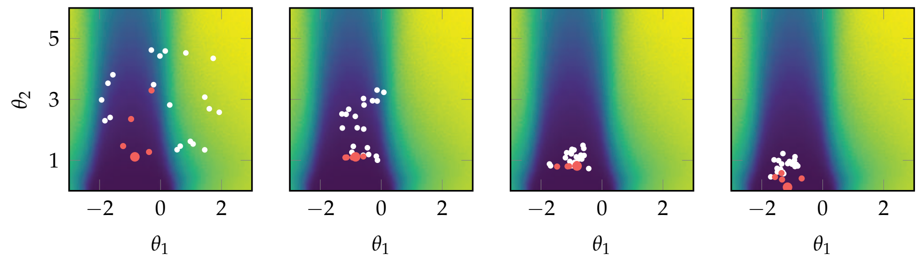


Figure 10.4. Genetic policy search with $\sigma = 0.25$ applied to the simple regulator problem using 25 samples per iteration. The five elite samples in each generation are shown in red, with the best sample indicated by a larger dot.

covariance of the distribution. The objective is to find a value of ψ^* that maximizes the expectation of $U(\theta)$ when θ is drawn from the search distribution:

$$\psi^* = \arg\max_{\psi} \mathop{\mathbb{E}}_{\theta \sim p(\cdot \mid \psi)}[U(\theta)] = \arg\max_{\psi} \int U(\theta) p(\theta \mid \psi)\, d\theta \tag{10.5}$$

Directly maximizing equation (10.5) is typically computationally infeasible. The approach taken in the cross entropy method is to start with an initial value of ψ, typically chosen so that the distribution is spread over the relevant parameter space. At each iteration, we draw m samples from the associated distribution and then update ψ to fit the elite samples. For the fit, we typically use the maximum likelihood estimate (section 4.1).[8] We stop after a fixed number of iterations, or until the search distribution becomes highly focused on an optimum. Figure 10.5 demonstrates the algorithm on a simple problem.

[8] The maximum likelihood estimate corresponds to the choice of ψ that minimizes the *cross entropy* (see appendix A.9) between the search distribution and the elite samples.

10.5 Evolution Strategies

Evolution strategies[9] update a search distribution parameterized by a vector $\boldsymbol{\psi}$ at each iteration. However, instead of fitting the distribution to a set of elite samples, they update the distribution by taking a step in the direction of the gradient.[10] The gradient of the objective in equation (10.5) can be computed as follows:[11]

[9] D. Wierstra, T. Schaul, T. Glasmachers, Y. Sun, J. Peters, and J. Schmidhuber, "Natural Evolution Strategies," *Journal of Machine Learning Research*, vol. 15, pp. 949–980, 2014.

[10] We are effectively doing gradient ascent, which is reviewed in appendix A.11.

[11] The policy parameter θ is not bolded here because it is not required to be a vector. However, $\boldsymbol{\psi}$ is in bold because we require it to be a vector when we work with the gradient of the objective.

```
struct CrossEntropyPolicySearch
    p       # initial distribution
    m       # number of samples
    m_elite # number of elite samples
    k_max   # number of iterations
end

function optimize_dist(M::CrossEntropyPolicySearch, π, U)
    p, m, m_elite, k_max = M.p, M.m, M.m_elite, M.k_max
    for k in 1:k_max
        θs = rand(p, m)
        us = [U(π, θs[:,i]) for i in 1:m]
        θ_elite = θs[:,sortperm(us)[(m-m_elite+1):m]]
        p = Distributions.fit(typeof(p), θ_elite)
    end
    return p
end

function optimize(M, π, U)
    return Distributions.mode(optimize_dist(M, π, U))
end
```

Algorithm 10.4. Cross entropy policy search, which iteratively improves a search distribution initially set to `p`. This algorithm takes as input a parameterized policy `π(θ, s)` and a policy evaluation function `U`. In each iteration, `m` samples are drawn and the top `m_elite` are used to refit the distribution. The algorithm terminates after `k_max` iterations. The distribution `p` can be defined using the `Distributions.jl` package. For example, we might define

```
μ = [0.0,1.0]
Σ = [1.0 0.0; 0.0 1.0]
p = MvNormal(μ,Σ)
```

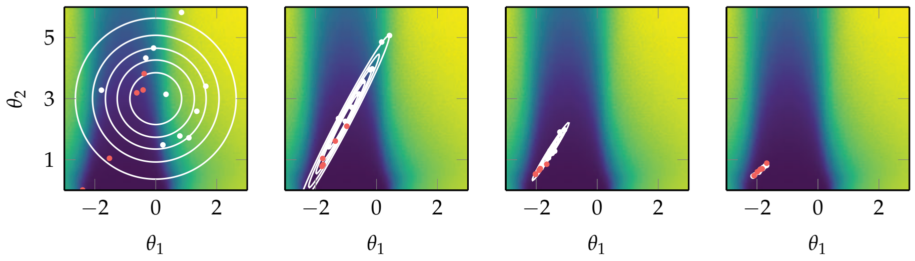


Figure 10.5. The cross entropy method applied to the simple regulator problem using a multivariate Gaussian search distribution. The five elite samples in each iteration are shown in red. The initial distribution is set to $\mathcal{N}([0,3], 2\mathbf{I})$.

$$\begin{aligned}
\nabla_{\boldsymbol{\psi}} \mathop{\mathbb{E}}_{\theta \sim p(\cdot \mid \boldsymbol{\psi})}[U(\theta)] &= \nabla_{\boldsymbol{\psi}} \int U(\theta) p(\theta \mid \boldsymbol{\psi}) \,\mathrm{d}\theta && (10.6)\\
&= \int U(\theta) \nabla_{\boldsymbol{\psi}} p(\theta \mid \boldsymbol{\psi}) \,\mathrm{d}\theta && (10.7)\\
&= \int U(\theta) \nabla_{\boldsymbol{\psi}} p(\theta \mid \boldsymbol{\psi}) \frac{p(\theta \mid \boldsymbol{\psi})}{p(\theta \mid \boldsymbol{\psi})} \,\mathrm{d}\theta && (10.8)\\
&= \int \big(U(\theta) \nabla_{\boldsymbol{\psi}} \log p(\theta \mid \boldsymbol{\psi})\big) p(\theta \mid \boldsymbol{\psi}) \,\mathrm{d}\theta && (10.9)\\
&= \mathop{\mathbb{E}}_{\theta \sim p(\cdot \mid \boldsymbol{\psi})} \big[U(\theta) \nabla_{\boldsymbol{\psi}} \log p(\theta \mid \boldsymbol{\psi})\big] && (10.10)
\end{aligned}$$

The introduction of the logarithm above comes from what is called the *log derivative trick*, which observes that $\nabla_{\boldsymbol{\psi}} \log p(\theta \mid \boldsymbol{\psi}) = \nabla_{\boldsymbol{\psi}} p(\theta \mid \boldsymbol{\psi}) / p(\theta \mid \boldsymbol{\psi})$. This computation requires knowing $\nabla_{\boldsymbol{\psi}} \log p(\theta \mid \boldsymbol{\psi})$, but we can often compute this analytically, as discussed in example 10.2.

The search gradient can be estimated from m samples: $\theta^{(1)}, \dots, \theta^{(m)} \sim p(\cdot \mid \boldsymbol{\psi})$:

$$\nabla_{\boldsymbol{\psi}} \mathop{\mathbb{E}}_{\theta \sim p(\cdot \mid \boldsymbol{\psi})}[U(\theta)] \approx \frac{1}{m} \sum_{i=1}^{m} U(\theta^{(i)}) \nabla_{\boldsymbol{\psi}} \log p(\theta^{(i)} \mid \boldsymbol{\psi}) \qquad (10.11)$$

This estimate depends on the evaluated expected utility, which itself can vary widely. We can make our gradient estimate more resilient with *rank shaping*, which replaces the utility values with weights based on the relative performance of each sample to the other samples in its iteration. The m samples are sorted in descending order of expected utility. Weight $w^{(i)}$ is assigned to sample i according to a weighting scheme with $w^{(1)} \geq \cdots \geq w^{(m)}$. The search gradient becomes

$$\nabla_{\boldsymbol{\psi}} \mathop{\mathbb{E}}_{\theta \sim p(\cdot \mid \boldsymbol{\psi})}[U(\theta)] \approx \sum_{i=1}^{m} w^{(i)} \nabla_{\boldsymbol{\psi}} \log p(\theta^{(i)} \mid \boldsymbol{\psi}) \qquad (10.12)$$

A common weighting scheme is[12]

$$w^{(i)} = \frac{\max\big(0, \log\big(\frac{m}{2} + 1\big) - \log(i)\big)}{\sum_{j=1}^{m} \max\big(0, \log\big(\frac{m}{2} + 1\big) - \log(j)\big)} - \frac{1}{m} \qquad (10.13)$$

[12] N. Hansen and A. Ostermeier, "Adapting Arbitrary Normal Mutation Distributions in Evolution Strategies: The Covariance Matrix Adaptation," in *IEEE International Conference on Evolutionary Computation*, 1996.

These weights, shown in figure 10.6, favor better samples and give most samples a small negative weight. Rank-shaping reduces the influence of outliers.

Algorithm 10.5 provides an implementation of the evolution strategies method. Figure 10.7 shows an example of a search progression.

The multivariate normal distribution $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, with mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$, is a common distribution family. The likelihood in d dimensions takes the form

$$p(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = (2\pi)^{-\frac{d}{2}} |\boldsymbol{\Sigma}|^{-\frac{1}{2}} \exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu})\right)$$

where $|\boldsymbol{\Sigma}|$ is the determinant of $\boldsymbol{\Sigma}$. The log likelihood is

$$\log p(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = -\frac{d}{2} \log(2\pi) - \frac{1}{2} \log |\boldsymbol{\Sigma}| - \frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu})$$

The parameters can be updated using their log likelihood gradients:

$$\begin{aligned}
\nabla_{\boldsymbol{\mu}} \log p(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) &= \boldsymbol{\Sigma}^{-1}(\mathbf{x} - \boldsymbol{\mu}) \\
\nabla_{\boldsymbol{\Sigma}} \log p(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) &= \frac{1}{2}\boldsymbol{\Sigma}^{-1}(\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^\top \boldsymbol{\Sigma}^{-1} - \frac{1}{2}\boldsymbol{\Sigma}^{-1}
\end{aligned}$$

The term $\nabla_{\boldsymbol{\Sigma}}$ contains the partial derivative of each entry of $\boldsymbol{\Sigma}$ with respect to the log likelihood.

Directly updating $\boldsymbol{\Sigma}$ may not result in a positive definite matrix, as is required for covariance matrices. One solution is to represent $\boldsymbol{\Sigma}$ as a product $\mathbf{A}^\top\mathbf{A}$, which guarantees that $\boldsymbol{\Sigma}$ remains positive semidefinite, and then to update $\mathbf{A}$ instead. Replacing $\boldsymbol{\Sigma}$ by $\mathbf{A}^\top\mathbf{A}$ and taking the gradient with respect to $\mathbf{A}$ yields

$$\nabla_{(\mathbf{A})} \log p(\mathbf{x} \mid \boldsymbol{\mu}, \mathbf{A}) = \mathbf{A}\left[\nabla_{\boldsymbol{\Sigma}} \log p(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) + \nabla_{\boldsymbol{\Sigma}} \log p(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma})^\top\right]$$

Example 10.2. A derivation of the log likelihood gradient equations for the multivariate Gaussian distribution. For the original derivation and several more sophisticated solutions for handling the positive definite covariance matrix, see D. Wierstra, T. Schaul, T. Glasmachers, Y. Sun, J. Peters, and J. Schmidhuber, "Natural Evolution Strategies," *Journal of Machine Learning Research*, vol. 15, pp. 949–980, 2014.

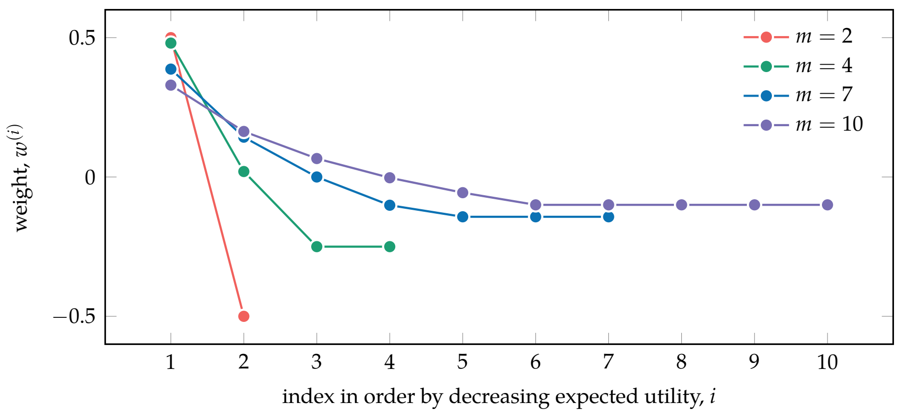


Figure 10.6. Several weightings constructed using equation (10.13).

```
struct EvolutionStrategies
    D        # distribution constructor
    ψ        # initial distribution parameterization
    ∇logp    # log search likelihood gradient
    m        # number of samples
    α        # step factor
    k_max    # number of iterations
end

function evolution_strategy_weights(m)
    ws = [max(0, log(m/2+1) - log(i)) for i in 1:m]
    ws ./= sum(ws)
    ws .-= 1/m
    return ws
end

function optimize_dist(M::EvolutionStrategies, π, U)
    D, ψ, m, ∇logp, α = M.D, M.ψ, M.m, M.∇logp, M.α
    ws = evolution_strategy_weights(m)
    for k in 1:M.k_max
        θs = rand(D(ψ), m)
        us = [U(π, θs[:,i]) for i in 1:m]
        sp = sortperm(us, rev=true)
        ∇ = sum(w.*∇logp(ψ, θs[:,i]) for (w,i) in zip(ws,sp))
        ψ += α.*∇
    end
    return D(ψ)
end
```

Algorithm 10.5. An evolution strategies method for updating a search distribution `D(ψ)` over policy parameterizations for policy `π(θ, s)`. This implementation also takes an initial search distribution parameterization `ψ`, the log search likelihood gradient `∇logp(ψ, θ)`, a policy evaluation function `U`, and an iteration count `k_max`. In each iteration, `m` parameterization samples are drawn and are used to estimate the search gradient. This gradient is then applied with a step factor `α`. We can use `Distributions.jl` to define `D(ψ)`. For example, if we want to define `D` to construct a Gaussian with a given mean `ψ` and fixed covariance `Σ`, we can use `D(ψ) = MvNormal(ψ, Σ)`.

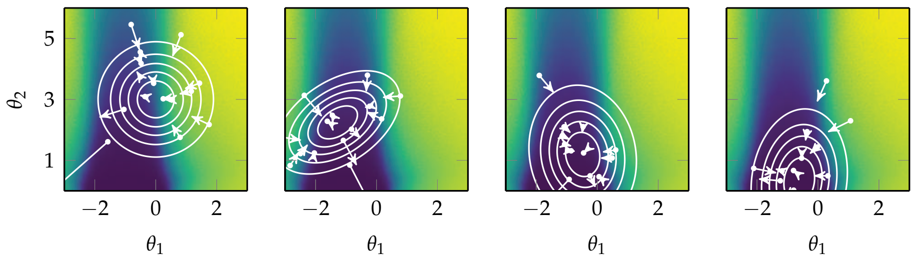


Figure 10.7. Evolution strategies (algorithm 10.5) applied to the simple regulator problem using a multivariate Gaussian search distribution. Samples are shown in white, along with their search gradient contributions, $w\nabla \log p$.

10.6 Isotropic Evolutionary Strategies

The previous section introduced evolutionary strategies that can work with general search distributions. This section will make the assumption that the search distribution is a *spherical* or *isotropic* Gaussian, where the covariance matrix takes the form $\sigma^2\mathbf{I}$.[13] Under this assumption, the expected utility of the distribution introduced in equation (10.5) simplifies to[14]

$$\mathop{\mathbb{E}}_{\boldsymbol{\theta}\sim\mathcal{N}(\boldsymbol{\psi},\sigma^2\mathbf{I})}[U(\boldsymbol{\theta})] = \mathop{\mathbb{E}}_{\boldsymbol{\epsilon}\sim\mathcal{N}(\mathbf{0},\mathbf{I})}[U(\boldsymbol{\psi}+\sigma\boldsymbol{\epsilon})] \tag{10.14}$$

The search gradient reduces to

$$\begin{aligned}
\nabla_{\boldsymbol{\psi}} \mathop{\mathbb{E}}_{\boldsymbol{\theta}\sim\mathcal{N}(\boldsymbol{\psi},\sigma^2\mathbf{I})}[U(\boldsymbol{\theta})] &= \mathop{\mathbb{E}}_{\boldsymbol{\theta}\sim\mathcal{N}(\boldsymbol{\psi},\sigma^2\mathbf{I})}\left[U(\boldsymbol{\theta})\nabla_{\boldsymbol{\psi}} \log p(\boldsymbol{\theta} \mid \boldsymbol{\psi},\sigma^2\mathbf{I})\right] && (10.15)\\
&= \mathop{\mathbb{E}}_{\boldsymbol{\theta}\sim\mathcal{N}(\boldsymbol{\psi},\sigma^2\mathbf{I})}\left[U(\boldsymbol{\theta})\frac{1}{\sigma^2}(\boldsymbol{\theta}-\boldsymbol{\psi})\right] && (10.16)\\
&= \mathop{\mathbb{E}}_{\boldsymbol{\epsilon}\sim\mathcal{N}(\mathbf{0},\mathbf{I})}\left[U(\boldsymbol{\psi}+\sigma\boldsymbol{\epsilon})\frac{1}{\sigma^2}(\sigma\boldsymbol{\epsilon})\right] && (10.17)\\
&= \frac{1}{\sigma}\mathop{\mathbb{E}}_{\boldsymbol{\epsilon}\sim\mathcal{N}(\mathbf{0},\mathbf{I})}[U(\boldsymbol{\psi}+\sigma\boldsymbol{\epsilon})\boldsymbol{\epsilon}] && (10.18)
\end{aligned}$$

Algorithm 10.6 provides an implementation of this strategy. This implementation incorporates *mirrored sampling*.[15] We sample $m/2$ values from the search distribution and then generate the other $m/2$ samples by mirroring them about the mean. Mirrored samples reduce the variance of the gradient estimate.[16] The benefit of using this technique is shown in figure 10.8.

[13] An example of this approach applied to policy search is explored by T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, "Evolution Strategies as a Scalable Alternative to Reinforcement Learning," 2017. arXiv: 1703.03864v2.

[14] In general, if $\mathbf{A}^\top\mathbf{A} = \boldsymbol{\Sigma}$, then $\boldsymbol{\theta} = \boldsymbol{\mu} + \mathbf{A}^\top\boldsymbol{\epsilon}$ transforms $\epsilon \sim \mathcal{N}(\mathbf{0},\mathbf{I})$ into a sample $\boldsymbol{\theta} \sim \mathcal{N}(\boldsymbol{\mu},\boldsymbol{\Sigma})$.

[15] D. Brockhoff, A. Auger, N. Hansen, D. Arnold, and T. Hohm, "Mirrored Sampling and Sequential Selection for Evolution Strategies," in *International Conference on Parallel Problem Solving from Nature*, 2010.

[16] This technique was implemented by T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever, "Evolution Strategies as a Scalable Alternative to Reinforcement Learning," 2017. arXiv: 1703.03864v2. They included other techniques as well, including weight decay.

```
struct IsotropicEvolutionStrategies
    ψ       # initial mean
    σ       # initial standard deviation
    m       # number of samples
    α       # step factor
    k_max   # number of iterations
end

function optimize_dist(M::IsotropicEvolutionStrategies, π, U)
    ψ, σ, m, α, k_max = M.ψ, M.σ, M.m, M.α, M.k_max
    n = length(ψ)
    ws = evolution_strategy_weights(2*div(m,2))
    for k in 1:k_max
        ϵs = [randn(n) for i in 1:div(m,2)]
        append!(ϵs, -ϵs) # weight mirroring
        us = [U(π, ψ + σ.*ϵ) for ϵ in ϵs]
        sp = sortperm(us, rev=true)
        ∇ = sum(w.*ϵs[i] for (w,i) in zip(ws,sp)) / σ
        ψ += α.*∇
    end
    return MvNormal(ψ, σ)
end
```

Algorithm 10.6. An evolution strategies method for updating an isotropic multivariate Gaussian search distribution with mean `ψ` and covariance $\sigma^2\mathbf{I}$ over policy parameterizations for a policy `π(θ, s)`. This implementation also takes a policy evaluation function `U`, a step factor `α`, and an iteration count `k_max`. In each iteration, `m/2` parameterization samples are drawn and mirrored and are then used to estimate the search gradient.

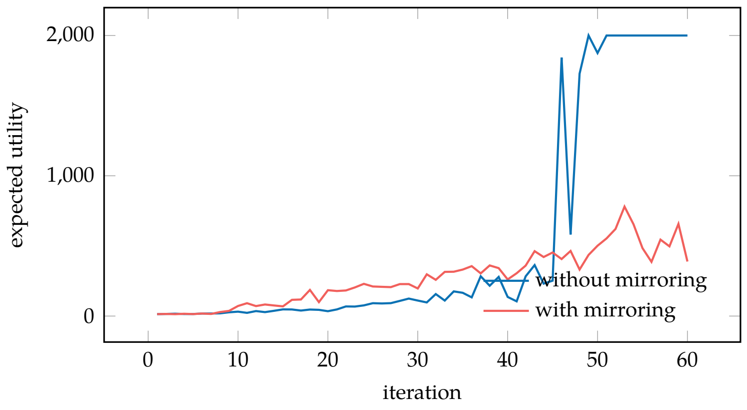


Figure 10.8. A demonstration of the effect that mirrored sampling has on isotropic evolution strategies. Two-layer neural network policies were trained on the cart-pole problem (appendix F.3) using `m = 10`, and `σ = 0.25`, with six rollouts per evaluation. Mirrored sampling significantly speeds and stabilizes learning.

10.7 Summary

- Monte Carlo policy evaluation involves computing the expected utility associated with a policy using a large number of rollouts from states sampled from an initial state distribution.
- Local search methods, such as the Hooke-Jeeves method, improve a policy based on small, local changes.
- Genetic algorithms maintain a population of points in the parameter space, recombining them in different ways in attempt to drive the population toward a global optimum.
- The cross entropy method iteratively improves a search distribution over policy parameters by refitting the distribution to elite samples at each iteration.
- Evolutionary strategies attempt to improve the search distribution using gradient information from samples from that distribution.
- Isotropic evolutionary strategies make the assumption that the search distribution is an isotropic Gaussian.

10.8 Exercises

Exercise 10.1. In Monte Carlo policy evaluation, how is the variance of the utility estimate affected by the number of samples?

Solution: The variance of Monte Carlo policy evaluation is the variance of the mean of m samples. These samples are assumed to be independent, and so the variance of the mean is the variance of a single rollout evaluation divided by the sample size:

$$\text{Var}[\hat{U}(\pi)] = \text{Var}\left[\frac{1}{m}\sum_{i=1}^{m} R(\tau^{(i)})\right] = \frac{1}{m^2}\text{Var}\left[\sum_{i=1}^{m} R(\tau^{(i)})\right] = \frac{1}{m^2}\left(\sum_{i=1}^{m}\text{Var}\left[R(\tau^{(i)})\right]\right) = \frac{1}{m}\text{Var}_\tau[R(\tau)]$$

where $\hat{U}(\pi)$ is the utility from Monte Carlo policy evaluation and $R(\tau)$ is the trajectory reward for a sampled trajectory τ. The sample variance, therefore, decreases with $1/m$.

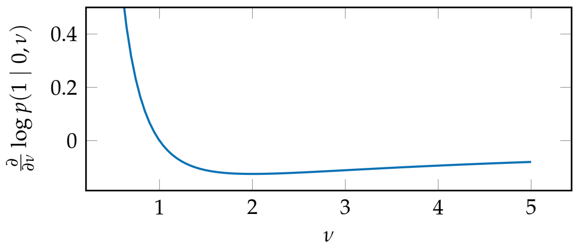
Exercise 10.2. What effect does varying the number of samples m and the number of elite samples m_elite have on cross entropy policy search?

Solution: The computational cost per iteration scales linearly with the number of samples. More samples will better cover the search space, resulting in a better chance of identifying better elite samples to improve the policy. The number of elite samples also has an effect. Making all samples elite provides no feedback to the improvement process. Having too few elite samples can lead to premature convergence to a suboptimal solution.

Exercise 10.3. Consider using evolution strategies with a univariate Gaussian distribution, $\theta \sim \mathcal{N}(\mu, \nu)$. What is the search gradient with respect to the variance ν? What issue arises as the variance becomes small?

Solution: The search gradient is the gradient of the log-likelihood:

$$\begin{aligned}
\frac{\partial}{\partial \nu} \log p(x \mid \mu, \nu) &= \frac{\partial}{\partial \nu} \log \frac{1}{\sqrt{2\pi\nu}} \exp\left(-\frac{(x-\mu)^2}{2\nu}\right) \\
&= \frac{\partial}{\partial \nu}\left(-\frac{1}{2}\log(2\pi) - \frac{1}{2}\log(\nu) - \frac{(x-\mu)^2}{2\nu}\right) \\
&= -\frac{1}{2\nu} + \frac{(x-\mu)^2}{2\nu^2}
\end{aligned}$$



We find that the gradient goes to infinity as the variance approaches zero. This is a problem because the variance should be small when the search distribution converges. Very large gradients can cause simple ascent methods to overshoot optima.

Exercise 10.4. Equation (10.14) defines the objective in terms of a search distribution $\boldsymbol{\theta} \sim \mathcal{N}(\boldsymbol{\psi}, \boldsymbol{\Sigma})$. What advantage does this objective have over directly optimizing $\boldsymbol{\theta}$ using the expected utility objective in equation (10.1)?

Solution: The added Gaussian noise around the policy parameters can smooth discontinuities in the original objective, which can make optimization more reliable.

Exercise 10.5. Which of the methods in this chapter are best suited to the fact that multiple types of policies could perform well in a given problem?

Solution: The Hooke-Jeeves method improves a single policy parameterization, so it cannot retain multiple policies. Both the cross entropy method and evolution strategies use search distributions. In order to successfully represent multiple types of policies, a multimodal distribution would have to be used. One common multimodal distribution is a mixture of Gaussians. A mixture of Gaussians cannot be fit analytically, but they can be reliably fit using expectation maximization (EM), as demonstrated in example 4.4. Genetic algorithms can retain multiple policies if the population size is sufficiently large.

Exercise 10.6. Suppose we have a parameterized policy π_θ that we would like to optimize using the Hooke-Jeeves method. If we initialize our parameter $\theta = 0$ and the utility function is $U(\theta) = -3\theta^2 + 4\theta + 1$, what is the largest step size α that would still guarantee policy improvement in the first iteration of the Hooke-Jeeves method?

Solution: The Hooke-Jeeves method evaluates the objective function at the center point $\pm\alpha$ along each coordinate direction. In order to guarantee improvement in the first iteration of Hooke-Jeeves search, at least one of the objective function values at the new points must improve the objective function value. For our policy optimization problem, this means that we are searching for the largest step size α such that either $U(\theta + \alpha)$ or $U(\theta - \alpha)$ is greater than $U(\theta)$.

Since the underlying utility function is parabolic and concave, the largest step size that would still lead to improvement is slightly less than the width of the parabola at the current point. Thus, we compute the point on the parabola opposite the current point, θ' at which $U(\theta') = U(\theta)$:

$$\begin{aligned}
U(\theta) &= -3\theta^2 + 4\theta + 1 = -3(0)^2 + 4(0) + 1 = 1 \\
U(\theta) &= U(\theta') \\
1 &= -3\theta'^2 + 4\theta' + 1 \\
0 &= -3\theta'^2 + 4\theta' + 0 \\
\theta' &= \frac{-4 \pm \sqrt{4^2 - 4(-3)(0)}}{2(-3)} = \frac{-4 \pm 4}{-6} = \frac{2 \pm 2}{3} = \left\{0, \tfrac{4}{3}\right\}
\end{aligned}$$

The point on the parabola opposite the current point is thus $\theta' = \frac{4}{3}$. The distance between θ and θ' is $\frac{4}{3} - 0 = \frac{4}{3}$. Thus, the maximal step size we can take and still guarantee improvement in the first iteration is just under $\frac{4}{3}$.

Exercise 10.7. Suppose we have a policy parameterized by a single parameter θ. We take an evolution strategies approach with a search distribution that follows a Bernoulli distribution $p(\theta \mid \psi) = \psi^\theta(1 - \psi)^{1-\theta}$. Compute the log-likelihood gradient $\nabla_\psi \log p(\theta \mid \psi)$.

Solution: The log-likelihood gradient can be computed as follows:

$$\begin{aligned}
p(\theta \mid \psi) &= \psi^{\theta}(1-\psi)^{1-\theta} \\
\log p(\theta \mid \psi) &= \log\left(\psi^{\theta}(1-\psi)^{1-\theta}\right) \\
\log p(\theta \mid \psi) &= \theta \log \psi + (1-\theta)\log(1-\psi) \\
\nabla_{\psi} \log p(\theta \mid \psi) &= \frac{d}{d\psi}\left[\theta \log \psi + (1-\theta)\log(1-\psi)\right] \\
\nabla_{\psi} \log p(\theta \mid \psi) &= \frac{\theta}{\psi} - \frac{1-\theta}{1-\psi}
\end{aligned}$$

Exercise 10.8. Compute the sample weights for search gradient estimation with rank shaping given $m = 3$ samples.

Solution: We first compute the numerator of the first term from equation (10.13), for all i:

$$\begin{aligned}
i = 1 &\qquad \max\left(0, \log\left(\tfrac{3}{2}+1\right) - \log 1\right) = \log \tfrac{5}{2} \\
i = 2 &\qquad \max\left(0, \log\left(\tfrac{3}{2}+1\right) - \log 2\right) = \log \tfrac{5}{4} \\
i = 3 &\qquad \max\left(0, \log\left(\tfrac{3}{2}+1\right) - \log 3\right) = 0
\end{aligned}$$

Now, we compute the weights:

$$\begin{aligned}
w^{(1)} &= \frac{\log \frac{5}{2}}{\log \frac{5}{2} + \log \frac{5}{4} + 0} - \frac{1}{3} = 0.47 \\
w^{(2)} &= \frac{\log \frac{5}{4}}{\log \frac{5}{2} + \log \frac{5}{4} + 0} - \frac{1}{3} = -0.14 \\
w^{(3)} &= \frac{0}{\log \frac{5}{2} + \log \frac{5}{4} + 0} - \frac{1}{3} = -0.33
\end{aligned}$$