18 Imitation Learning

Previous chapters have assumed that a reward function is either known or that rewards are received while interacting with the environment. For some applications, it may be easier for an expert to demonstrate the desired behavior rather than specifying a reward function. This chapter discusses algorithms for *imitation learning*, where the desired behavior is learned from expert demonstration. We will cover a variety of methods ranging from very simple likelihood-maximization methods to more complicated iterative methods that involve reinforcement learning.[1]

[1] Additional methods and applications are surveyed by A. Hussein, M. M. Gaber, E. Elyan, and C. Jayne, "Imitation Learning," *ACM Computing Surveys*, vol. 50, no. 2, pp. 1–35, 2017.

18.1 Behavioral Cloning

A simple form of imitation learning is to treat it as a supervised learning problem. This method, called *behavioral cloning*,[2] trains a stochastic policy π_θ parameterized by θ to maximize the likelihood of actions from a dataset $\mathcal{D}$ of expert state-action pairs:

$$\underset{\theta}{\text{maximize}} \quad \prod_{(s,a)\in\mathcal{D}} \pi_\theta(a \mid s) \tag{18.1}$$

[2] D. A. Pomerleau, "Efficient Training of Artificial Neural Networks for Autonomous Navigation," *Neural Computation*, vol. 3, no. 1, pp. 88–97, 1991.

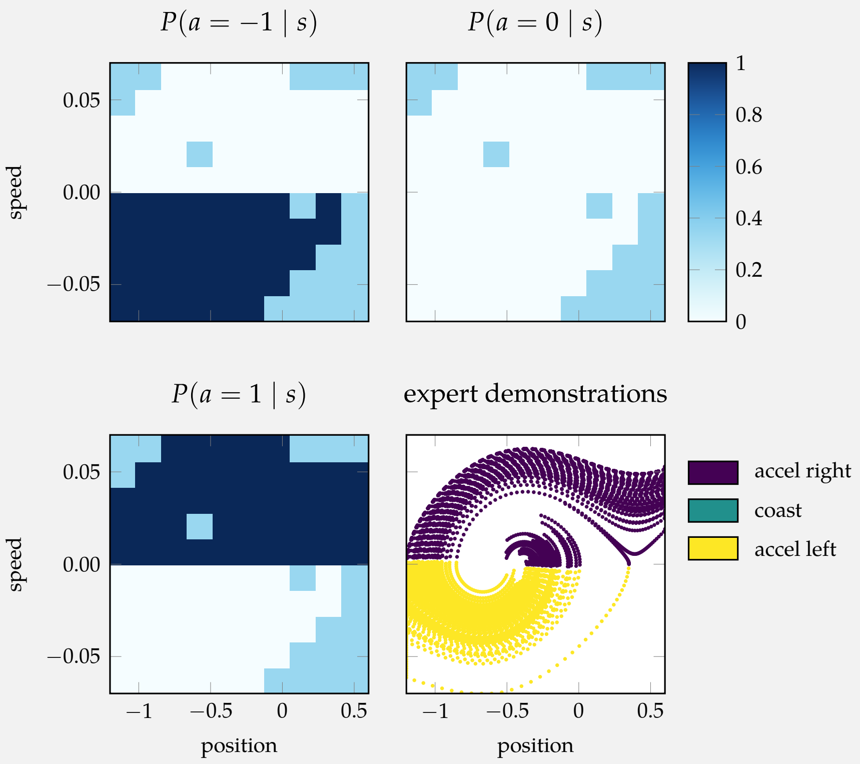
As done in earlier chapters, we can transform the maximization over the product over $\pi_\theta(a \mid s)$ to a sum over $\log \pi_\theta(a \mid s)$.

Depending on how we want to represent the conditional distribution $\pi_\theta(a \mid s)$, we may compute the maximum likelihood estimate of θ analytically. For example, if we use a discrete conditional model (section 2.4), θ would consist of all of the counts $N(s, a)$ from $\mathcal{D}$ and $\pi_\theta(a \mid s) = N(s, a) / \sum_a N(s, a)$. Example 18.1 applies a discrete conditional model to data from the mountain car problem.

If we have a factored representation of our policy, we can use a Bayesian network to represent the joint distribution over our state and action variables. Figure 18.1 shows an example. We can learn both the structure (chapter 5) and

Example 18.1. A demonstration of behavioral cloning applied to the mountain car problem. The light blue regions are areas without training data, resulting in poor policy performance when the agent encounters those states.

Consider using behavioral cloning on expert demonstrations for the mountain car problem (appendix F.4). We are given 10 rollouts from an expert policy. We fit a conditional distribution and plot the results. The continuous trajectories were discretized with 10 bins each for position and for speed.



The state space is not fully covered by the exert demonstrations, which is typical for imitation learning problems. The resulting policy may perform well when used in regions with coverage, but it assigns a uniform distribution to actions in regions without coverage. Even if we start in a region with coverage, we may transition to regions without coverage due to stochasticity in the environment.

parameters (chapter 4) from the data $\mathcal{D}$. Given the current state, we can then infer the distribution over actions using one of the inference algorithms discussed earlier (chapter 3).

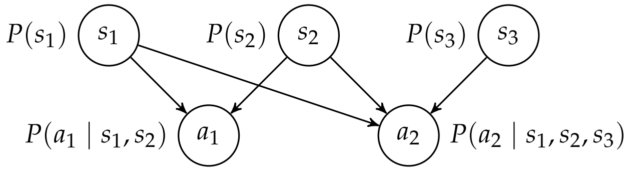


Figure 18.1. Bayesian networks can be used to represent a joint distribution over the state and action variables. We can apply an inference algorithm to generate a distribution over actions given the current values of the state variables.

We can use many other representations for π_θ. For example, we might want to use a neural network, where the input corresponds to the values of the state variables and the output corresponds to parameters of a distribution over the action space. If our representation is differentiable, which is the case with neural networks, we can attempt to optimize equation (18.1) using gradient ascent. This approach is implemented in algorithm 18.1.

```
struct BehavioralCloning
    α     # step size
    k_max # number of iterations
    ∇logπ # log likelihood gradient
end

function optimize(M::BehavioralCloning, D, θ)
    α, k_max, ∇logπ = M.α, M.k_max, M.∇logπ
    for k in 1:k_max
        ∇ = mean(∇logπ(θ, a, s) for (s,a) in D)
        θ += α*∇
    end
    return θ
end
```

Algorithm 18.1. A method for learning a parameterized stochastic policy from expert demonstrations in the form of a set of state-action tuples `D`. The policy parameterization vector `θ` is iteratively improved by maximizing the log likelihood of the actions given the states. Behavioral cloning requires a step size `α`, an iteration count `k_max`, and a log likelihood gradient `∇logπ`.

The closer the expert demonstrations are to optimal, the better the resulting behavioral cloning policy will perform.[3] However, behavioral cloning suffers from *cascading errors*. As discussed in example 18.2, small inaccuracies compound during a rollout and eventually lead to states that are poorly represented in the training data, thereby leading to worse decisions, and ultimately to invalid or unseen situations. Though behavioral cloning is attractive due to its simplicity, cascading errors cause the method to perform poorly on many problems, especially when policies must be used for long time horizons.

[3] U. Syed and R. E. Schapire, "A Reduction from Apprenticeship Learning to Classification," in *Advances in Neural Information Processing Systems (NIPS)*, 2010.

Consider applying behavioral cloning to train a policy for driving an autonomous race car. A human race car driver provides expert demonstrations. Being an expert, the driver never drifts onto the grass or too close to a railing. A model trained with behavioral cloning would have no information to go off of when near a railing or when drifting onto the grass, and would not know how to recover.

Example 18.2. A brief example of the generalization issue inherent to behavioral cloning approaches.

18.2 Dataset Aggregation

One way to address the problem of cascading errors is to correct a trained policy using additional expert input. *Sequential interactive demonstration* methods alternate between collecting data from an expert in situations generated by a trained policy and using this data to improve this policy.

One type of sequential interactive demonstration method is called *dataset aggregation* (*DAgger*) (algorithm 18.2).[4] It starts by training a stochastic policy using behavioral cloning. The policy is then used to run several rollouts from an initial state distribution b, which are then given to an expert to provide the correct actions for each state. The new data is aggregated with the previous dataset and a new policy is trained. Example 18.3 illustrates this process.

These interactive demonstrations iteratively builds a dataset covering the regions of the state space that the agent is likely to encounter, based on previous learning iterations. With each iteration, newly added examples compose a smaller fraction of the dataset, thereby leading to smaller policy changes. While sequential interactive demonstration can work well in practice, it is not guaranteed to converge.

[4] S. Ross, G. J. Gordon, and J. A. Bagnell, "A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning," in *International Conference on Artificial Intelligence and Statistics* (*AISTATS*), vol. 15, 2011.

18.3 Stochastic Mixing Iterative Learning

Sequential interactive methods can also iteratively build up a policy by stochastically mixing in newly trained policies. One such method is *stochastic mixing iterative learning* (*SMILe*) (algorithm 18.3).[5] It uses behavioral cloning in every iteration but mixes the newly trained policy in with the previous ones.

We start with the expert policy, $\pi^{(1)} = \pi_E$.[6] In each iteration, we execute the latest policy $\pi^{(k)}$ to generate a new dataset, querying the expert to provide

[5] S. Ross and J. A. Bagnell, "Efficient Reductions for Imitation Learning," in *International Conference on Artificial Intelligence and Statistics* (*AISTATS*), 2010.

[6] We do not have an explicit representation of π_E. Evaluating π_E requires interactively querying the expert as done in the previous section.

```
struct DatasetAggregation
    𝒫     # problem with unknown reward function
    bc    # behavioral cloning struct
    k_max # number of iterations
    m     # number of rollouts per iteration
    d     # rollout depth
    b     # initial state distribution
    πE    # expert
    πθ    # parameterized policy
end

function optimize(M::DatasetAggregation, D, θ)
    𝒫, bc, k_max, m = M.𝒫, M.bc, M.k_max, M.m
    d, b, πE, πθ = M.d, M.b, M.πE, M.πθ
    θ = optimize(bc, D, θ)
    for k in 2:k_max
        for i in 1:m
            s = rand(b)
            for j in 1:d
                push!(D, (s, πE(s)))
                a = rand(πθ(θ, s))
                s = rand(𝒫.T(s, a))
            end
        end
        θ = optimize(bc, D, θ)
    end
    return θ
end
```

Algorithm 18.2. The DAgger method of dataset aggregation for learning a stochastic parameterized policy from expert demonstrations. This method takes an initial dataset of state-action tuples `D`, a stochastic parameterized policy `πθ(θ, s)`, an MDP `𝒫` that defines a transition function, and an initial state distribution `b`. Behavioral cloning, algorithm 18.1, is used in each iteration to improve the policy.

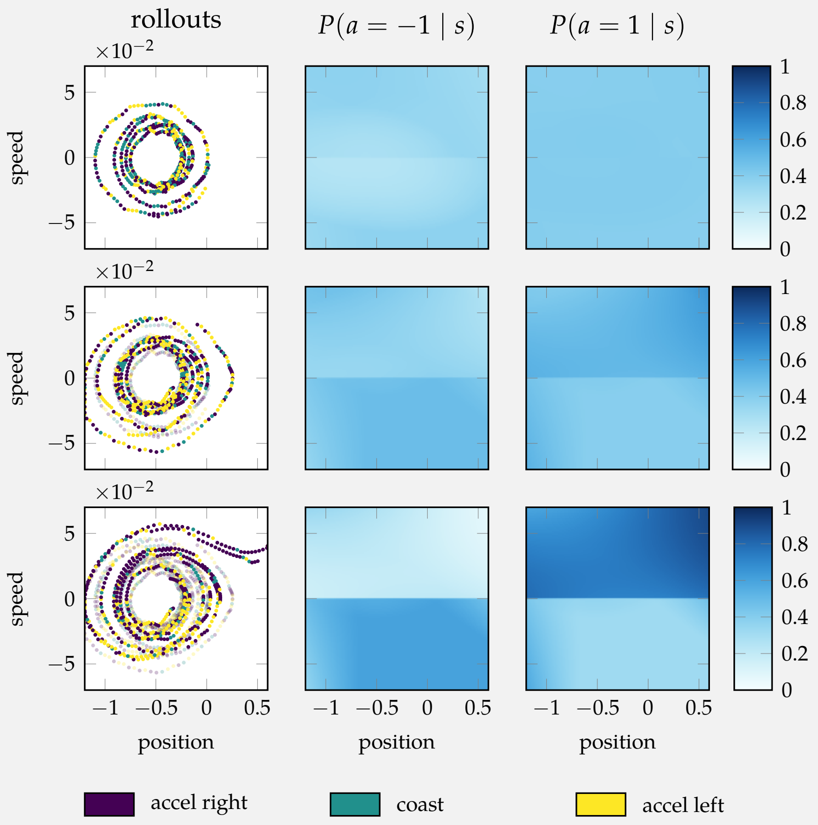
An expert policy `πE` labels trajectories sampled from the latest learned policy to augment the dataset. The original paper generated trajectories by stochastically mixing in the expert policy. This implementation is thus the original DAgger with an extreme mixing value of zero.

In practice an expert policy would not exist, and calls to this policy would be replaced with queries to a human expert.

Consider using dataset aggregation to train a policy on the mountain car problem where the reward is not observed. We use an expert policy that accelerates in the direction of travel. In this example we train a policy using the features:

$$\mathbf{f}(s) = [1[v > 0], 1[v < 0], x, x^2, v, v^2, xv]$$

where x and v are the position and speed of the car.



Trajectories begin in red and proceed to blue. In the first iteration, the agent behaves randomly, unable to make progress toward the goal ($x \geq 0.6$). With additional iterations, the agent learns to mimic the expert policy of accelerating in the direction of travel. This behavior is apparent in the new trajectories, which spiral outward, and the policy, which assigns high likelihood to $a = 1$ when $v > 0$ and $a = -1$ when $v < 0$.

Example 18.3. Dataset aggregation applied to the mountain car problem, with iterations running from top to bottom. Trajectories accumulate in the dataset over time. The behavior of the agent improves with each iteration.

the correct actions. Behavioral cloning is applied only to this new dataset to train a new *component policy* $\hat{\pi}^{(k)}$. This component policy is mixed together with component policies from the previous iterations to produce a new policy $\pi^{(k+1)}$.

The mixing of component policies to generate $\pi^{(k+1)}$ is governed by a mixing scalar $\beta \in (0, 1)$. The probability of acting according to the expert policy is $(1-\beta)^k$, and the probability of acting according to $\hat{\pi}^{(i)}$ is $\beta(1-\beta)^{i-1}$. This scheme assigns more weight to older policies under the hypothesis that older policy components were trained on the states more likely to be encountered. With each iteration, the probability of acting according to the original expert policy decays to zero. The mixing scalar is typically small such that the agent does not abandon the expert's policy too quickly. Example 18.4 demonstrates this approach on mountain car.

18.4 Maximum Margin Inverse Reinforcement Learning

In many application settings, we do not have an expert that can be interactively queried, but instead have a batch of expert demonstration trajectories. We will assume that the expert demonstration data $\mathcal{D}$ consists of m trajectories. Each trajectory τ in $\mathcal{D}$ involves a rollout to depth d. In *inverse reinforcement learning*, we assume that the expert is optimizing some unknown reward function. From $\mathcal{D}$, we attempt to derive that reward function. With that reward function, we can use the methods discussed in prior chapters to derive an optimal policy.

There are different approaches to inverse reinforcement learning. We generally need to define a parameterization of the reward function. A common assumption is that this parameterization is linear with $R_{\boldsymbol{\phi}}(s, a) = \boldsymbol{\phi}^\top \boldsymbol{\beta}(s, a)$, where $\boldsymbol{\beta}(s, a)$ is a feature vector and $\boldsymbol{\phi}$ is a vector of weightings. In this section, we will focus on an approach known as *maximum margin inverse reinforcement learning*,[7] where the features are assumed to be binary. Since optimal policies remain optimal with positive scaling of the reward function, this method additionally constrains the weight vector such that $\|\boldsymbol{\phi}\|_2 \leq 1$. The expert data activates each binary feature with different frequencies, perhaps pursuing some and avoiding others. This approach attempts to learn this pattern of activation and trains an agent to mimic these activation frequencies.

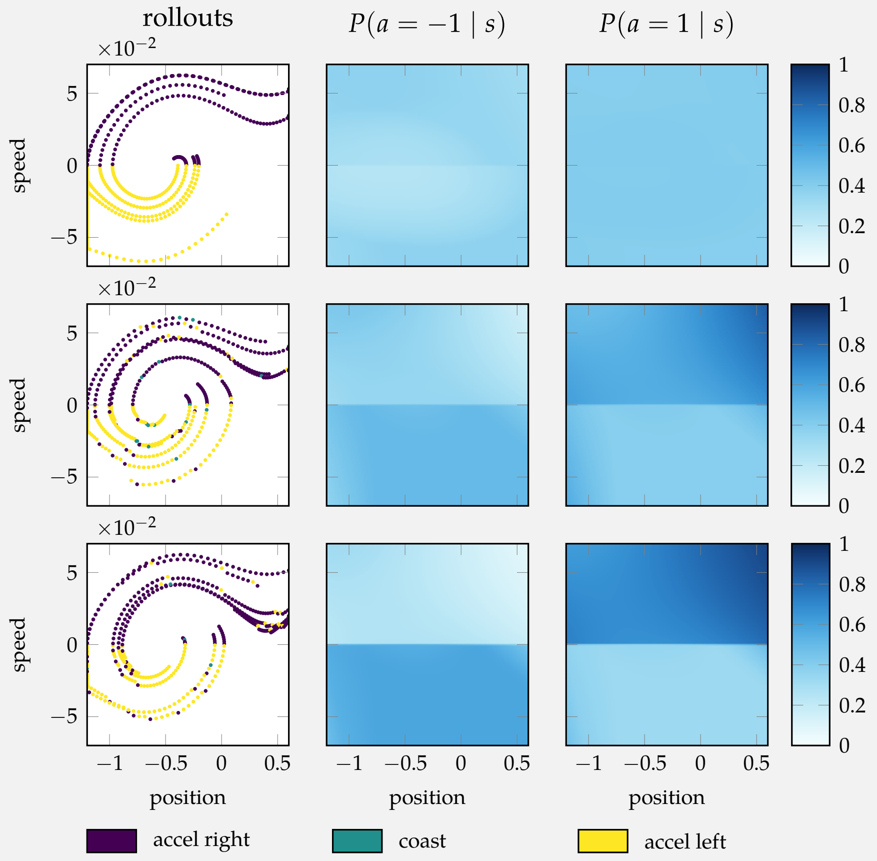
[7] P. Abbeel and A. Y. Ng, "Apprenticeship Learning via Inverse Reinforcement Learning," in *International Conference on Machine Learning (ICML)*, 2004.

```
struct SMILe
    𝒫     # problem with unknown reward
    bc    # Behavioral cloning struct
    k_max # number of iterations
    m     # number of rollouts per iteration
    d     # rollout depth
    b     # initial state distribution
    β     # mixing scalar (e.g., d^-3)
    πE    # expert policy
    πθ    # parameterized policy
end

function optimize(M::SMILe, θ)
    𝒫, bc, k_max, m = M.𝒫, M.bc, M.k_max, M.m
    d, b, β, πE, πθ = M.d, M.b, M.β, M.πE, M.πθ
    𝒜, T = 𝒫.𝒜, 𝒫.T
    θs = []
    π = s -> πE(s)
    for k in 1:k_max
        # execute latest π to get new dataset D
        D = []
        for i in 1:m
            s = rand(b)
            for j in 1:d
                push!(D, (s, πE(s)))
                a = π(s)
                s = rand(T(s, a))
            end
        end
        # train new policy classifier
        θ = optimize(bc, D, θ)
        push!(θs, θ)
        # compute a new policy mixture
        Pπ = Categorical(normalize([(1-β)^(i-1) for i in 1:k],1))
        π = s -> begin
            if rand() < (1-β)^(k-1)
                return πE(s)
            else
                return rand(Categorical(πθ(θs[rand(Pπ)], s)))
            end
        end
    end
    Ps = normalize([(1-β)^(i-1) for i in 1:k_max],1)
    return Ps, θs
end
```

Algorithm 18.3. The SMILe algorithm for training a stochastic parameterized policy from expert demonstrations for an MDP 𝒫. SMILe successively mixes in new component policies with smaller and smaller weight, while simultaneously reducing the probability of acting according to the expert policy. The method returns the probabilities `Ps` and parameterizations `θs` for the component policies.

Consider using SMILe to train a policy on the mountain car problem where the reward is not observed. We use the same features that were used for dataset aggregation in example 18.3. Both dataset aggregation and SMILe receive a new expert-labeled dataset with each iteration. Instead of accumulating a larger dataset of expert-labeled data, SMILe trains a new policy component using only the most recent data, mixing the new policy component with the previous policy components.



Example 18.4. An example of using SMILe to learn a policy for the mountain car problem. In contrast with dataset aggregation in example 18.3, SMILe mixes the expert into the policy during rollouts. This expert component, whose influence wanes with each iteration, causes the initial rollouts to better progress toward the goal.

An important part of this algorithm involves reasoning about the expected return under a policy π for a weighting $\boldsymbol{\phi}$ and initial state distribution b:

$$\begin{aligned}
\mathop{\mathbb{E}}_{s\sim b}[U(s)] &= \mathbb{E}_\tau\left[\sum_{k=1}^{d} \gamma^{k-1} R_{\boldsymbol{\phi}}(s^{(k)}, a^{(k)})\right] && (18.2)\\
&= \mathbb{E}_\tau\left[\sum_{k=1}^{d} \gamma^{k-1} \boldsymbol{\phi}^\top \boldsymbol{\beta}(s^{(k)}, a^{(k)})\right] && (18.3)\\
&= \boldsymbol{\phi}^\top\left(\mathbb{E}_\tau\left[\sum_{k=1}^{d} \gamma^{k-1} \boldsymbol{\beta}(s^{(k)}, a^{(k)})\right]\right) && (18.4)\\
&= \boldsymbol{\phi}^\top \boldsymbol{\mu}_\pi && (18.5)
\end{aligned}$$

where τ corresponds to trajectories generated by π to depth d. Here, we introduce the *feature expectations* $\boldsymbol{\mu}_\pi$, which is the expected discounted accumulated feature values. These feature expectations can be estimated from m rollouts, as implemented in algorithm 18.4.

```
struct InverseReinforcementLearning
    𝒫  # problem
    b  # initial state distribution
    d  # depth
    m  # number of samples
    π  # parameterized policy
    β  # binary feature mapping
    μE # expert feature expectations
    RL # reinforcement learning method
    ϵ  # tolerance
end

function feature_expectations(M::InverseReinforcementLearning, π)
    𝒫, b, m, d, β, γ = M.𝒫, M.b, M.m, M.d, M.β, M.𝒫.γ
    μ(τ) = sum(γ^(k-1)*β(s, a) for (k,(s,a)) in enumerate(τ))
    τs = [simulate(𝒫, rand(b), π, d) for i in 1:m]
    return mean(μ(τ) for τ in τs)
end
```

Algorithm 18.4. A structure for inverse reinforcement learning and a method for estimating a feature expectations vector from rollouts.

We can use the expert demonstrations to estimate the expert feature expectations $\boldsymbol{\mu}_E$, and we want to find a policy that matches these feature expectations as closely as possible. At the first iteration, we begin with a randomized policy $\pi^{(1)}$ and estimate its feature expectations that we denote $\boldsymbol{\mu}^{(1)}$. At iteration k, we find a new $\boldsymbol{\phi}^{(k)}$, corresponding to a reward function $R_{\boldsymbol{\phi}^{(k)}}(s, a) = \boldsymbol{\phi}^{(k)\top}\boldsymbol{\beta}(s, a)$, such that the expert outperforms all previously found policies by the greatest *margin t*:

$$\begin{aligned} \underset{t,\boldsymbol{\phi}}{\text{maximize}} \quad & t \\ \text{subject to} \quad & \boldsymbol{\phi}^\top \boldsymbol{\mu}_E \geq \boldsymbol{\phi}^\top \boldsymbol{\mu}^{(i)} + t \ \text{ for } \ i = 1, \ldots, k-1 \\ & \|\boldsymbol{\phi}\|_2 \leq 1 \end{aligned} \tag{18.6}$$

Equation (18.6) is a quadratic program that can be easily solved. We then solve for a new policy $\pi^{(k)}$ using the reward function $R(s, a) = \boldsymbol{\phi}^{(k)\top} \boldsymbol{\beta}(s, a)$ and produce a new vector of feature expectations. Figure 18.2 illustrates this margin maximization process.

We iterate until the margin is sufficiently small with $t \leq \epsilon$. At convergence, we can solve for a mixed policy that attempts to have feature expectations as close as possible to that of the expert policy:

$$\begin{aligned} \underset{\boldsymbol{\lambda}}{\text{minimize}} \quad & \|\boldsymbol{\mu}_E - \boldsymbol{\mu}_{\boldsymbol{\lambda}}\|_2 \\ \text{subject to} \quad & \boldsymbol{\lambda} \geq 0 \\ & \|\boldsymbol{\lambda}\|_1 = 1 \end{aligned} \tag{18.7}$$

where $\boldsymbol{\mu}_{\boldsymbol{\lambda}} = \sum_i \lambda_i \boldsymbol{\mu}^{(i)}$. The mixture weights $\boldsymbol{\lambda}$ combine the policies found at each iteration. With probability λ_i, we follow policy $\pi^{(i)}$. Inverse reinforcement learning is implemented in algorithm 18.5.

18.5 Maximum Entropy Inverse Reinforcement Learning

The inverse reinforcement learning approach from the previous section is underspecified, meaning there are often multiple policies that can produce the same feature expectations as the expert demonstrations. This section introduces *maximum entropy inverse reinforcement learning*, which avoids this ambiguity by preferring the policy that results in the distribution over trajectories that has maximum *entropy* (appendix A.8).[8] The problem can be transformed into one of finding the best reward function parameters $\boldsymbol{\phi}$ in a maximum likelihood estimation problem given the expert data $\mathcal{D}$.

[8] B. D. Ziebart, A. Maas, J. A. Bagnell, and A. K. Dey, "Maximum Entropy Inverse Reinforcement Learning," in *AAAI Conference on Artificial Intelligence* (*AAAI*), 2008.

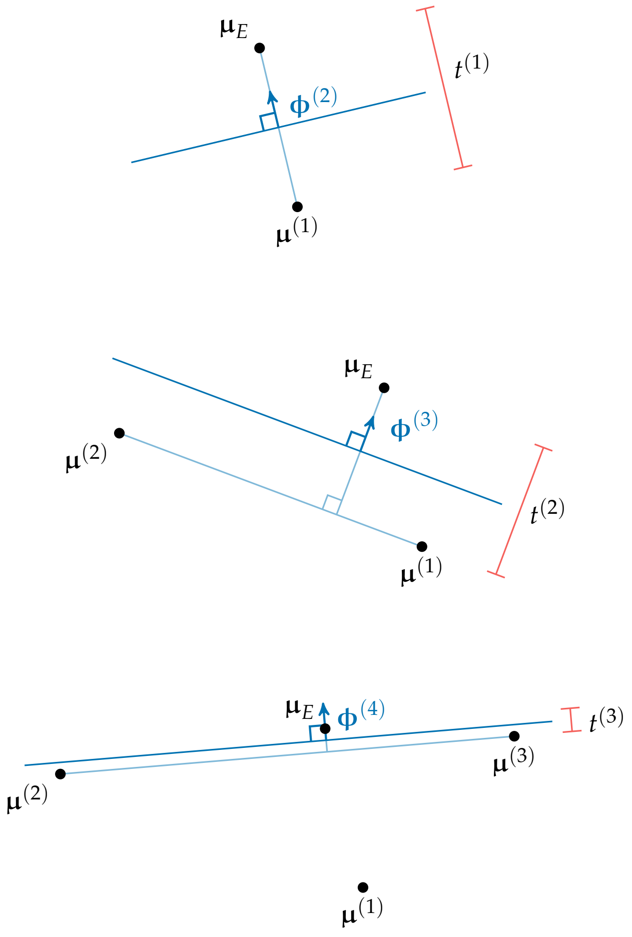


Figure 18.2. A geometric visualization of three example iterations of the maximum-margin inverse reinforcement learning algorithm, going top to bottom. In each iteration, the new weight vector points in the direction perpendicular to the hyperplane that separates the expert feature expectation vector from the those of the previous policy with the largest possible margin. The margin decreases with each iteration.

```julia
function calc_weighting(M::InverseReinforcementLearning, μs)
    μE = M.μE
    k = length(μE)
    model = Model(Ipopt.Optimizer)
    @variable(model, t)
    @variable(model, ϕ[1:k] ≥ 0)
    @objective(model, Max, t)
    for μ in μs
        @constraint(model, ϕ⋅μE ≥ ϕ⋅μ + t)
    end
    @constraint(model, ϕ⋅ϕ ≤ 1)
    optimize!(model)
    return (value(t), value.(ϕ))
end

function calc_policy_mixture(M::InverseReinforcementLearning, μs)
    μE = M.μE
    k = length(μs)
    model = Model(Ipopt.Optimizer)
    @variable(model, λ[1:k] ≥ 0)
    @objective(model, Min, (μE - sum(λ[i]*μs[i] for i in 1:k))⋅
                            (μE - sum(λ[i]*μs[i] for i in 1:k)))
    @constraint(model, sum(λ) == 1)
    optimize!(model)
    return value.(λ)
end

function optimize(M::InverseReinforcementLearning, θ)
    π, ϵ, RL = M.π, M.ϵ, M.RL
    θs = [θ]
    μs = [feature_expectations(M, s->π(θ,s))]
    while true
        t, ϕ = calc_weighting(M, μs)
        if t ≤ ϵ
            break
        end
        copyto!(RL.ϕ, ϕ) # R(s,a) = ϕ⋅β(s,a)
        θ = optimize(RL, π, θ)
        push!(θs, θ)
        push!(μs, feature_expectations(M, s->π(θ,s)))
    end
    λ = calc_policy_mixture(M, μs)
    return λ, θs
end
```

Algorithm 18.5. Maximum margin inverse reinforcement learning, which computes a mixed policy whose feature expectations match those of given expert demonstrations. We use `JuMP.jl` to solve our constrained optimization problems. This implementation requires that the provided reinforcement learning struct has a weight vector `ϕ` that can be updated with new values. The method returns the stochastic weightings `λ` and parameterizations `θs` for the component policies.

Any policy π induces a distribution over trajectories[9] $P_\pi(\tau)$. Different policies produce different trajectory distributions. We are free to choose any of these distributions over trajectories that match the expert feature expectations. The *principle of maximum entropy* chooses the least informative distribution, which corresponds to the one with maximum entropy.[10] It can be shown that the least informative trajectory distribution has the form:

$$P_{\boldsymbol{\phi}}(\tau) = \frac{1}{Z(\boldsymbol{\phi})} \exp(R_{\boldsymbol{\phi}}(\tau)) \tag{18.8}$$

where $P_{\boldsymbol{\phi}}(\tau)$ is the likelihood of a trajectory τ given reward parameter $\boldsymbol{\phi}$, and

$$R_{\boldsymbol{\phi}}(\tau) = \sum_{k=1}^{d} \gamma^{k-1} R_{\boldsymbol{\phi}}(s^{(k)}, a^{(k)}) \tag{18.9}$$

is the discounted trajectory reward. We make no assumption on the parameterization of $R_{\boldsymbol{\phi}}(s^{(k)}, a^{(k)})$ other than that it is differentiable, allowing for representations such as neural networks. The normalization scalar $Z(\boldsymbol{\phi})$ ensures that the probabilities sum to 1:

$$Z(\boldsymbol{\phi}) = \sum_{\tau} \exp(R_{\boldsymbol{\phi}}(\tau)) \tag{18.10}$$

The summation is over all possible trajectories.

We seek to maximize the likelihood of the expert demonstrations under this distribution:

$$\max_{\boldsymbol{\phi}} f(\boldsymbol{\phi}) = \max_{\boldsymbol{\phi}} \sum_{\tau \in \mathcal{D}} \log P_{\boldsymbol{\phi}}(\tau) \tag{18.11}$$

We can rewrite the objective function $f(\boldsymbol{\phi})$ from equation (18.11):

$$\begin{aligned} f(\boldsymbol{\phi}) &= \sum_{\tau \in \mathcal{D}} \log \frac{1}{Z(\boldsymbol{\phi})} \exp(R_{\boldsymbol{\phi}}(\tau)) && (18.12) \\ &= \left(\sum_{\tau \in \mathcal{D}} R_{\boldsymbol{\phi}}(\tau) \right) - |\mathcal{D}| \log Z(\boldsymbol{\phi}) && (18.13) \\ &= \left(\sum_{\tau \in \mathcal{D}} R_{\boldsymbol{\phi}}(\tau) \right) - |\mathcal{D}| \log \sum_{\tau} \exp(R_{\boldsymbol{\phi}}(\tau)) && (18.14) \end{aligned}$$

[9] For simplicity, this section assumes a finite horizon and that the state and action spaces are discrete, making $P_{\boldsymbol{\phi}}(\tau)$ a probability mass. To extend maximum entropy inverse reinforcement learning both to problems with continuous state and action spaces where the dynamics may be unknown, consider guided cost learning: C. Finn, S. Levine, and P. Abbeel, "Guided Cost Learning: Deep Inverse Optimal Control via Policy Optimization," in *International Conference on Machine Learning* (*ICML*), 2016.

[10] For an introduction to this principle, see E. T. Jaynes, "Information Theory and Statistical Mechanics," *Physical Review*, vol. 106, no. 4, pp. 620–630, 1957.

We can attempt to optimize this objective function through gradient ascent. The gradient of f is:

$$\nabla_{\boldsymbol{\phi}} f = \left(\sum_{\tau \in \mathcal{D}} \nabla_{\boldsymbol{\phi}} R_{\boldsymbol{\phi}}(\tau)\right) - \frac{|\mathcal{D}|}{\sum_\tau \exp(R_{\boldsymbol{\phi}}(\tau))} \sum_\tau \exp(R_{\boldsymbol{\phi}}(\tau)) \nabla_{\boldsymbol{\phi}} R_{\boldsymbol{\phi}}(\tau) \tag{18.15}$$

$$= \left(\sum_{\tau \in \mathcal{D}} \nabla_{\boldsymbol{\phi}} R_{\boldsymbol{\phi}}(\tau)\right) - |\mathcal{D}| \sum_\tau P_{\boldsymbol{\phi}}(\tau) \nabla_{\boldsymbol{\phi}} R_{\boldsymbol{\phi}}(\tau) \tag{18.16}$$

$$= \left(\sum_{\tau \in \mathcal{D}} \nabla_{\boldsymbol{\phi}} R_{\boldsymbol{\phi}}(\tau)\right) - |\mathcal{D}| \sum_s b_{\gamma,\boldsymbol{\phi}}(s) \sum_a \pi_{\boldsymbol{\phi}}(a \mid s) \nabla_{\boldsymbol{\phi}} R_{\boldsymbol{\phi}}(s,a) \tag{18.17}$$

If the reward function is linear with $R_{\boldsymbol{\phi}}(s,a) = \boldsymbol{\phi}^\top \boldsymbol{\beta}(s,a)$ as in the previous section, then $\nabla_{\boldsymbol{\phi}} R_{\boldsymbol{\phi}}(s,a)$ is simply $\boldsymbol{\beta}(s,a)$.

Updating the parameter vector $\boldsymbol{\phi}$ thus requires both the discounted state visitation frequency $b_{\gamma,\boldsymbol{\phi}}$ and the optimal policy under the current parameter vector, $\pi_{\boldsymbol{\phi}}(a \mid s)$. We can obtain the optimal policy by running reinforcement learning. To compute the discounted state visitation frequencies we can use rollouts or take a dynamic programming approach.

If we take a dynamic programming approach to compute the discounted state visitation frequencies, we can start with the initial state distribution $b_{\gamma\boldsymbol{\phi}}^{(1)} = b(s)$ and iteratively work forward in time:

$$b_{\gamma,\boldsymbol{\phi}}^{(k+1)}(s) = \gamma \sum_a \sum_{s'} b_{\gamma,\boldsymbol{\phi}}^{(k)}(s) \pi(a \mid s) T(s' \mid s,a) \tag{18.18}$$

This version of maximum entropy inverse reinforcement learning is implemented in algorithm 18.6.

18.6 Generative Adversarial Imitation Learning

In *generative adversarial imitation learning* (*GAIL*),[11] we optimize a differentiable parameterized policy $\pi_{\boldsymbol{\theta}}$, often represented by a neural network. Rather than provide a reward function, we use *adversarial learning* (appendix D.7). We also train a *discriminator* $C_{\boldsymbol{\phi}}(s,a)$, typically also a neural network, to return the probability it assigns to the state-action pair coming from the learned policy. The process involves alternating between training this discriminator to become better

[11] J. Ho and S. Ermon, "Generative Adversarial Imitation Learning," in *Advances in Neural Information Processing Systems* (*NIPS*), 2016.

```
struct MaximumEntropyIRL
    𝒫     # problem
    b     # initial state distribution
    d     # depth
    π     # parameterized policy π(θ,s)
    Pπ    # parameterized policy likelihood π(θ, a, s)
    ∇R    # reward function gradient
    RL    # reinforcement learning method
    α     # step size
    k_max # number of iterations
end

function discounted_state_visitations(M::MaximumEntropyIRL, θ)
    𝒫, b, d, Pπ = M.𝒫, M.b, M.d, M.Pπ
    𝒮, 𝒜, T, γ = 𝒫.𝒮, 𝒫.𝒜, 𝒫.T, 𝒫.γ
    b_sk = zeros(length(𝒫.𝒮), d)
    b_sk[:,1] = [pdf(b, s) for s in 𝒮]
    for k in 2:d
        for (si′, s′) in enumerate(𝒮)
            b_sk[si′,k] = γ*sum(sum(b_sk[si,k-1]*Pπ(θ, a, s)*T(s, a, s′)
                    for (si,s) in enumerate(𝒮))
                for a in 𝒜)
        end
    end
    return normalize!(vec(mean(b_sk, dims=2)),1)
end

function optimize(M::MaximumEntropyIRL, D, ϕ, θ)
    𝒫, π, Pπ, ∇R, RL, α, k_max = M.𝒫, M.π, M.Pπ, M.∇R, M.RL, M.α, M.k_max
    𝒮, 𝒜, γ, nD = 𝒫.𝒮, 𝒫.𝒜, 𝒫.γ, length(D)
    for k in 1:k_max
        copyto!(RL.ϕ, ϕ) # update parameters
        θ = optimize(RL, π, θ)
        b = discounted_state_visitations(M, θ)
        ∇Rτ = τ -> sum(γ^(i-1)*∇R(ϕ,s,a) for (i,(s,a)) in enumerate(τ))
        ∇f = sum(∇Rτ(τ) for τ in D) - nD*sum(b[si]*sum(Pπ(θ,a,s)*∇R(ϕ,s,a)
                    for (ai,a) in enumerate(𝒜))
                for (si, s) in enumerate(𝒮))
        ϕ += α*∇f
    end
    return ϕ, θ
end
```

Algorithm 18.6. Maximum entropy inverse reinforcement learning, which finds a stochastic policy that maximizes the likelihood of the expert demonstrations under a maximum-entropy trajectory distribution. This implementation computes the expected visitations using dynamic programming over all states, which requires that the problem be discrete.

at distinguishing, and training the policy to look indistinguishable from the expert demonstrations. The process is sketched in figure 18.3.

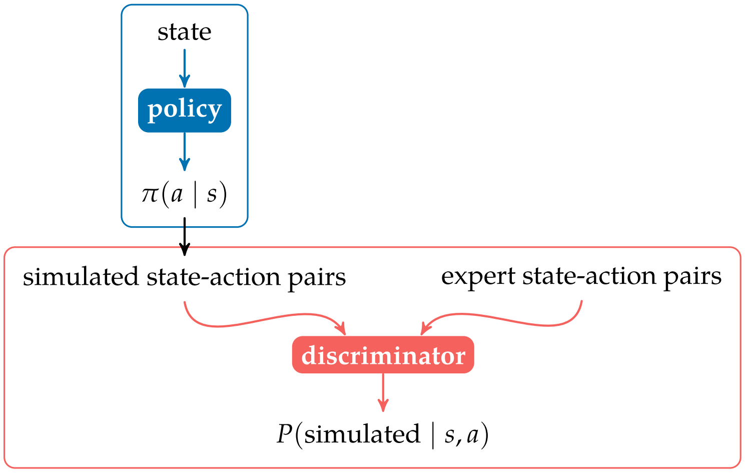

state
policy
$\pi(a \mid s)$
simulated state-action pairs
expert state-action pairs
discriminator
$P(\text{simulated} \mid s, a)$

Figure 18.3. Instead of inferring a reward function, generative adversarial imitation learning optimizes a discriminator to distinguish between simulated and expert state-action pairs, and it optimizes a policy to appear indistinguishable to the discriminator. The aim is to eventually produce a policy that resembles the expert.

The discriminator and policy have opposite objectives. Generative adversarial imitation learning seeks to find a saddle-point $(\boldsymbol{\theta}, \boldsymbol{\phi})$ of the negative log loss of the discriminator's binary classification problem:[12]

$$\max_{\boldsymbol{\phi}} \min_{\boldsymbol{\theta}} \mathbb{E}_{(s,a)\sim\pi_{\boldsymbol{\theta}}}\left[\log(C_{\boldsymbol{\phi}}(s,a))\right] + \mathbb{E}_{(s,a)\sim\mathcal{D}}\left[\log(1 - C_{\boldsymbol{\phi}}(s,a))\right] \tag{18.19}$$

[12] The original paper also includes an entropy term:

$$-\lambda\, \mathbb{E}_{(s,a)\sim\mathcal{D}}[-\log \pi_{\boldsymbol{\theta}}(a \mid s)]$$

where we use $(s, a) \sim \mathcal{D}$ to represent samples from the distribution represented by the expert dataset $\mathcal{D}$. We can alternate between gradient ascent on $\boldsymbol{\phi}$ to increase the objective and trust region policy optimization (section 12.4) on $\boldsymbol{\theta}$ to reduce the objective, generating the necessary trajectory samples from the policy to conduct each of these steps. The discriminator provides a learning signal to the policy similar to how a reward signal would if it were known.

18.7 *Summary*

- Imitation learning involves learning the desired behavior from expert demonstration without the use of a reward function.
- One type of imitation learning is behavioral cloning, which produces a stochastic policy that maximizes the conditional likelihood of the actions in the dataset.

- When an expert can be queried multiple times, we can use iterative approaches like dataset aggregation or stochastic mixing iterative learning.
- Inverse reinforcement learning involves inferring a reward function from expert data and then using traditional methods for finding an optimal policy.
- Maximum margin inverse reinforcement learning attempts to find a policy that matches the frequency of binary features found in the expert dataset.
- Maximum entropy inverse reinforcement learning frames the problem of finding the best reward parameter as a maximum likelihood estimation problem, which it tries to solve using gradient ascent.
- Generative adversarial imitation learning iteratively optimizes a discriminator and a policy; the discriminator tries to discriminate between decisions made by the policy and decisions made by the expert, and the policy attempts to deceive the discriminator.

18.8 Exercises

Exercise 18.1. Consider applying behavioral cloning to a discrete problem where we have been given expert demonstrations. An alternative could be to define a feature function $\boldsymbol{\beta}(s)$ and represent the policy with a softmax distribution

$$\pi(a \mid s) \propto \exp(\boldsymbol{\theta}_a^\top \boldsymbol{\beta}(s))$$

We would then learn the parameters $\boldsymbol{\theta}_a$ for each action from the expert data. Why might we want to use this approach over one where we directly estimate a discrete distribution for each state, with one parameter per state-action pair?

Solution: In imitation learning, we are generally limited to a relatively small set of expert demonstrations. The distribution $P(a \mid s)$ has $(|\mathcal{A}| - 1)|\mathcal{S}|$ independent parameters that must be learned, which is often prohibitively large. Expert demonstrations typically only cover a small portion of the state-space. Even if $P(a \mid s)$ can be reliably trained for the states covered in the provided dataset, the resulting policy would be untrained in other states. Using a feature function allows for generalization to unseen states.

Exercise 18.2. The section on behavioral cloning suggested using a maximum likelihood approach for training a policy from expert data. This approach attempts to find the parameters of the policy that maximizes the likelihood assigned to the training examples. In some problems, however, we know that assigning high probability to one incorrect action is not as bad as assigning high probability to another incorrect action. For example, predicting an acceleration of -1 in the mountain car problem when the expert dictates an acceleration of 1 is worse than predicting an acceleration of 0. How might behavioral cloning be modified to allow different penalties to be given to different misclassifications?

Solution: We can instead supply a cost function $C(s, a_{\text{true}}, a_{\text{pred}})$ that defines the cost of predicting action a_{pred} for state s when the expert's action is a_{true}. For example, with the mountain-car problem we might use:

$$C(s, a_{\text{true}}, a_{\text{pred}}) = -|a_{\text{true}} - a_{\text{pred}}|$$

which penalizes greater deviations more than smaller deviations. The cost associated with the expert's action is typically zero.

If we have a stochastic policy $\pi(a \mid s)$, then we seek to minimize the cost over our dataset:

$$\underset{\theta}{\text{minimize}} \quad \sum_{(s, a_{\text{true}}) \in \mathcal{D}} \sum_{a_{\text{pred}}} C\Big(s, a_{\text{true}}, a_{\text{pred}}\Big) \pi\Big(a_{\text{pred}} \mid s\Big)$$

This technique is called *cost-sensitive classification*.[13] One benefit of cost-sensitive classification is that we can use a wide variety of off-the-shelf classification models like k-nearest neighbors, support vector machines, or decision trees to train a policy.

[13] C. Elkan, "The Foundations of Cost-Sensitive Learning," in *International Joint Conference on Artificial Intelligence (IJCAI)*, 2001.

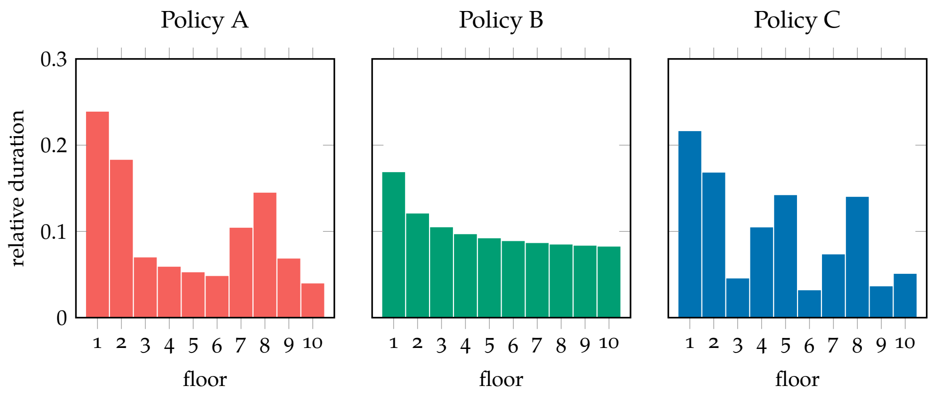
Exercise 18.3. Provide an example of where maximum margin inverse reinforcement learning does not uniquely define an optimal policy.

Solution: Maximum margin inverse reinforcement learning extracts binary features from the expert data and seeks a reward function whose optimal policy produces trajectories with the same frequencies of these binary features. There is no guarantee that multiple policies do not produce the same feature expectations. For example, an autonomous car that only makes left lane changes could have the same lane change frequencies as an autonomous car that only makes right lane changes.

Exercise 18.4. Maximum margin inverse reinforcement learning measures how similar a policy is to expert demonstrations using feature expectations. How is this similarity measure affected if non-binary features are used?

Solution: If we use non-binary features, then it is possible that some features can get larger than others, incentivizing the agent to match those features over those that tend to be smaller. Scale is not the only issue. Even if all features are constrained to lie within $[0, 1]$, then a policy that consistently produces $\phi(s, a)_1 = 0.5$ will have the same feature expectations as one that produces $\phi(s, a)_1 = 0$ half the time and $\phi(s, a)_1 = 1$ half the time. Depending on what the feature encodes, this can result in very different policies. Any set of continuous features can be discretized, and thus approximated by a set of binary features.

Exercise 18.5. Suppose we are building a system for an elevator in a high-rise that has to choose which floor to send an elevator. We have trained several policies to match the feature expectations of expert demonstrations, such as how long a customer has to wait for an elevator or how long they have to wait to get to their destination. We run multiple rollouts for each policy and plot the relative duration spent on each floor. Which policy should we prefer according to the principle of maximum entropy, assuming each policy matches the feature expectations equally?



Solution: These distributions over relative duration are analogous to distributions over trajectories for this elevator problem. In applying the principle of maximum entropy, we prefer the distribution with most entropy. Hence, we would choose policy B, which in being most uniform, has the greatest entropy.

Exercise 18.6. Consider the policy optimization step in generative adversarial imitation learning. Rewrite the objective in the form of a reward function so that traditional reinforcement learning techniques can be applied.

Solution: We rewrite equation (18.19), dropping the terms dependent on the expert dataset, and flip the sign to change from minimization over θ to a maximization over θ of the reward, producing the surrogate reward function:

$$\tilde{R}_{\boldsymbol{\phi}}(s, a) = -\log C_{\boldsymbol{\phi}}(s, a)$$

Although $\tilde{R}_{\boldsymbol{\phi}}(s, a)$ may be quite different from the unknown true reward function, it can be used to drive the learned policy into regions of the state-action space similar to those covered by the expert.

Exercise 18.7. Explain how generative adversarial imitation learning could be changed such that the discriminator takes in trajectories rather than state-action pairs. Why might this be useful?

Solution: Changing generative adversarial imitation learning such that the discriminator takes trajectories is straightforward, especially if the trajectories are of fixed length. The expert dataset is split into trajectories, and the learned policy is used to produce trajectories, just as it was before. Rather than operating on state-action pairs, the discriminator takes in trajectories using a representation such as a recurrent neural network (appendix D.5) and produces a classification probability. The objective function remains largely unchanged:

$$\max_{\boldsymbol{\phi}} \min_{\boldsymbol{\theta}} \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}}\left[\log(C_{\boldsymbol{\phi}}(\tau))\right] + \mathbb{E}_{\tau \sim \mathcal{D}}\left[\log(1 - C_{\boldsymbol{\phi}}(\tau))\right]$$

The advantage of running the discriminator over entire trajectories is that it can help the discriminator capture features that are not apparent from individual state-action pairs, which can result in better policies. For example, when looking at individual accelerations and turn rates for an autonomous driving policy, there is very little for a discriminator to learn. A discriminator trained to look at longer trajectories can see more of the vehicle's behavior, such as lane change aggressiveness and smoothness, to better match expert driving demonstrations.[14]

[14] This approach was used in A. Kuefler, J. Morton, T. A. Wheeler, and M. J. Kochenderfer, "Imitating Driver Behavior with Generative Adversarial Networks," in *IEEE Intelligent Vehicles Symposium (IV)*, 2017.