Reinforcement learning

1 *Introduction*

Our goal, as in behavior cloning, is to learn a mapping π from states s to actions a.

The original model for RL was a single agent interacting with the real environment throughout its lifetime. Its life is like this:

- the agent observes the world state s_0
- it picks an action a_0
- it receives scalar reward value r_0
- the agent observes the world state s_1
- it picks an action a_1
- it receives scalar reward value r_1
- etc

The objective of the agent is to act in such a way as to maximize the sum of rewards it receives in its lifetime. That would be a classic problem of solving an MDP (Markov decision process) if it knew how the world works, but in RL we don't know that.

The major differences from the supervised learning approach we used in behavior cloning are:

- The agent is never told what action it *ought* to have taken, instead it just gets a scalar reward value (it may not even know the possible range of rewards).
- Rewards are generally a function of the current state and action; the distribution of the next state depends on the current state and action. This means that the choice of action on step 1 might have significant effects on the agent's future state trajectory and hence rewards. So, it's not enough to use r_t to evaluate the choice of a_t.
- The agent's action choices don't just earn reward; they govern what future training data the agent is going to get!

These three properties make this problem very hard. It's still worth thinking about because

- Supervised training data (for behavior cloning) can be very hard to obtain.
- Even with a large amount of supervised data, it can happen that a BC-trained policy makes errors in such a way that it ends up in parts of the state space on which it hasn't been trained.
- BC aims to reproduce the demonstrator's behavior. It can sometimes do better by smoothing out mistakes, but its training objective does

Contents

1 Introduction 1
2 Value-based methods 3
3 Policy-based methods 8
4 Combined methods 15
5 Important concerns 23
6 Model-based RL 25
7 Other classes to take! 46
A The PPO version in HW6 47

We use capital letters for random variables and lowercase letters for values: S_t, A_t, R_t are the random state, action, and reward at time t, while s_t, a_t, r_t are their observed values. Likewise, G_t is a random return and g_t a realized return. A random trajectory is $\mathcal{T}$; τ denotes a particular trajectory. State and action sets are $\mathcal{S}$ and $\mathcal{A}$. Functions such as Q, V, R, and fixed quantities such as a horizon H, retain their conventional names.

We write $\mathbb{E}_{x\sim p(X)}[f(x)]$ to average $f(x)$ over values x drawn from the distribution $p(X)$. Thus $p(X)$ names a distribution, while $p(x)$ is its probability or density at a particular value.

See MIT 6.390 course notes for an intro to MDPs.

not explicitly seek higher reward. RL provides a way to search for such improvements.

1.1 Problem settings

There are three main settings for RL:

1. **Online, in the real world.** The OG version of RL, where we try to maximize reward over a single lifetime, combining learning, exploring, and exploiting.
2. **Online, but in a simulator.** The second setting is the most widely used in robotics. It's different from the original model in terms of its overall loss function. The objective of RL in a simulator is to produce a policy $\pi : \mathcal{S} \to \mathcal{A}$ that will perform well (in terms of expected summed reward). However, the actual rewards obtained *during learning* aren't important as long as the end product is good. This will manifest in different kinds of exploration and sampling strategies.
3. **Offline.** The third, offline, setting involves collecting a file of experience tuples: (s_t, a_t, r_t, s_{t+1}). Then an offline RL method has to learn what it can from that data, but is not allowed to do any exploration of its own. This can be a good thing to do when (a) no simulator is available and (b) real-world experience is expensive.

You can think of this as a kind of compilation: you know everything there is to know about the world (instantiated in the simulator) and goal (in the reward function) but you don't immediately know the best action to take. So, (kind of weirdly) we turn this into an RL problem of an agent interacting with the simulator.

1.2 Horizon

We spoke informally above about maximizing the sum of rewards...but what is the agent actually supposed to do? We have to think about how to think about the future.

Finite horizon is the simplest case. We assume the policy, when it's deployed, will be executed for a fixed number, H, of steps, and then be terminated. We write $p_\pi(\mathcal{T} \mid s_0)$ for the distribution over trajectories starting at s_0 and following π. The optimal strategy in that case is:

$$\pi^* = \arg\max_\pi \mathbb{E}_{\tau \sim p_\pi(\mathcal{T} \mid s_0)} \left[\sum_{t=0}^{H-1} r_t \right]$$

where $\pi = (\pi_0, \ldots, \pi_{H-1})$ is a sequence of policies, $A_t = \pi_t(S_t)$, and the expectation is taken with respect to randomness in the world's dynamics and rewards.

The finite horizon case is a little awkward, though, for two reasons:

- It's unusual to have a clear idea of how long you're going to get to live, in advance!
- If you want to have an optimal finite-horizon policy, you actually have to figure out different policies for each specific number of steps you might have left.

Receding horizon control is a simple strategy for avoiding the need for multiple policies. You pick a horizon H (we'll call it a hyperparameter), solve for the single policy that's optimal with H steps to go, and then just execute it over and over. This is often reasonable.

Infinite horizon discounted solves both problems with finite horizon: you don't have to pick a specific H and the optimal policy can be *stationary*: you just need one, fixed mapping from states to actions. In this case:

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\tau \sim p_\pi(\mathcal{T}|s_0)} \left[\sum_{t=0}^{\infty} \gamma^t r_t \right]$$

We introduce a new parameter $0 \leqslant \gamma < 1$ which is a discount factor. One way to interpret it is that, after each reward, the agent survives to the next step with probability γ, independently of everything else. Counting the initial reward, its expected lifetime is $1/(1-\gamma)$ steps. This termination rule has no memory: if you survive today, you have the same expected future lifetime starting tomorrow, so you don't need a policy that depends on your age.

There is also average-case reward, which is interesting and understudied!

1.3 In summary

We'll focus on infinite-horizon problems and on learning in simulation.

2 Value-based methods

Value-based methods have powerful theoretical foundations, but may not work well when their assumptions are violated. They are founded on estimating an action-value function

There is another formulation of this in terms of V instead of Q; they're easily inter-convertible.

$$Q^*(s,a) = \max_{\pi} \mathbb{E}_{\tau \sim p_\pi(\mathcal{T}|s,a)} \left[\sum_{t=0}^{\infty} \gamma^t r_t \right] \quad .$$

That is, the expected discounted *return* (sum of rewards) if we start in state s, execute action a, and act optimally thereafter. Here $p_\pi(\mathcal{T} \mid s, a)$ fixes the first state and action, then follows π. *Here's the cool thing:*

$$Q^*(s,a) = R(s,a) + \gamma \mathbb{E}_{s' \sim P(S'|s,a)} \left[\max_{a'} Q^*(s',a') \right] \quad ! \tag{1}$$

Here $R(s,a) = \mathbb{E}_{r \sim P(R_t|s,a)}[r]$ is the expected immediate reward; $P(R_t \mid s, a)$ is the reward distribution for state s and action a. And, if we know Q^* then

$$\pi^*(s) = \arg\max_{a} Q^*(s,a) \quad .$$

2.1 Q learning

This is a beautiful recursive relationship, which is the basis of our first RL algorithm, Q learning, assuming for now a finite set $\mathcal{S}$ of states and $\mathcal{A}$ of actions. We break ties in arg max arbitrarily. For now, we assume a continuing task; small changes are necessary to handle domains with a terminal state / reset.

`sim` is a simulator: we assume it can reset (return to an initial state drawn from some distribution) and execute a transition given an action, returning the reward received and the next state.

```
1: Q-learning(S, A, sim, α, γ) :
2:     Q := zeros(|S|, |A|)
3:     s := sim.draw_initial_state
4:     loop:    ▷ until tired—lots of interesting theory about this
5:         a := arg max_{a∈A} Q[s, a]    ▷ not right; needs exploration
6:         r, s' := sim.execute(a)
7:         Q[s, a] := (1 − α)Q[s, a] + α(r + γ max_{a'∈A} Q[s', a'])
8:         s = s'
9:     return Q
```

The parameter α is a learning rate. It lets us replace the expectation in math-land (equation 1) with a running average in algorithm-land. Other people prefer to write line 7 as

$$Q[s, a] := Q[s, a] + \alpha(r + \gamma \max_{a' \in \mathcal{A}} Q[s', a'] - Q[s, a])$$

which is in *temporal difference form* and lets us see this as making a kind of "error driven" update to our old value for $Q(s, a)$ based on a difference between what just happened and what we expected.

There are two kinds of "learning" going on here simultaneously: we are taking running averages to handle randomness in the outcomes of our actions and we're doing dynamic-programming updates (improving a current value estimate based on other value estimates). That these things can happen in parallel and still result in an algorithm that's provably correct is kind of surprising.

We have to address several problems to make Q-learning work in the real world.

The first is exploration. In line 5, we are just selecting the action that appears to be best given our current Q value estimates, but at the beginning, they will be *terrible*! It's possible we'll get stuck, not trying new things and learning that there are better options in the world. So, we need to at least add some "noise" to our action selection, by, with probability ϵ on each step deciding to execute an action at random instead of following the rule above. More about this in section 5.1.

We must be aware of the foundational assumption behind basically all value-based methods, which is that the things we are calling "states"

are actually states of a Markov decision process. That means that they have to be sufficiently informative about the world so that if you make future predictions based on the current state, they couldn't be made any more accurate by conditioning on the whole history of states and actions. *This is the same idea of state as we talked about when formulating system diagrams.* These methods can tolerate some amount of "non-Markovianity" but exactly how much is not well understood and it can be devastating. We'll discuss this more in section 5.2 on partial observability.

For a finite-state and finite-action MDP with bounded rewards and $0 \leqslant \gamma < 1$, Q learning converges to Q^* with probability one if we visit every state-action pair infinitely often and use learning rates satisfying

$Q^*(s, a)$ is the value of taking action a in state s and then continuing optimally.

$$0 < \alpha_n(s, a) \leqslant 1 \ , \qquad \sum_{n=1}^{\infty} \alpha_n(s, a) = \infty \ , \qquad \sum_{n=1}^{\infty} \alpha_n(s, a)^2 < \infty \ .$$

Here n counts updates to that particular pair. So, for this guarantee, the α in the pseudocode has to decrease appropriately; a fixed learning rate generally won't give exact convergence. A greedy policy for Q^* is optimal![1]

The greedy policy for any Q function is

$$\pi^Q(s) = \arg\max_a Q(s, a) \ .$$

[1] Watkins and Dayan (1992), Q-learning.

2.2 *Efficiency and scaling*

Basic Q learning is incredibly, frustratingly slow: if it takes 10 steps to reach a tasty reward, a forward pass of one-step updates on a simple chain initially propagates that reward back only one step. It can take many visits to learn a good policy. Replaying the transitions in reverse order can propagate the information much faster. One strategy that combines well with Q learning is to keep a *replay buffer*: a stored collection of past transitions (s, a, r, s'), with a flag indicating whether each transition ended the task. As the agent collects new experience, it adds transitions to the buffer. Learning updates use small batches sampled from this collection, so the same experience can contribute to many updates. We recompute the learning targets using our current value estimates; the recorded rewards and transitions stay fixed. This is useful because Q learning is an "off-policy" method: the experience needn't come from the policy we currently want to evaluate or improve. In the tabular setting, the convergence conditions above still apply.

Another big problem is scaling, both in terms of the sizes of the state and action spaces. In robotics, we generally want these to be continuous (e.g., joint angles) and/or high-dimensional (e.g., images). We generally handle both of these problems with neural networks. We'll start with large/continuous $\mathcal{S}$ but still manageable discrete $\mathcal{A}$.

In this case, instead of representing $Q[s, a]$ as an array, we represent it as a neural network with input s, weights θ, and output a vector of

$Q_\theta(s, a)$ for all a! Every time we do an update, we take a squared-loss gradient step while treating the target as fixed:

$$y = r + \gamma \max_{a' \in \mathcal{A}} Q_{\overline{\theta}}(s', a')$$
$$\delta = y - Q_\theta(s, a)$$
$$\theta \leftarrow \theta + \alpha \delta \nabla_\theta Q_\theta(s, a) \quad .$$

For a terminal transition, $y = r$. Here $\overline{\theta}$ are the weights of a separate target network, held fixed for a while and periodically copied from θ. With y fixed, this is gradient descent on $\frac{1}{2}(y - Q_\theta(s, a))^2$. It's called a *semi-gradient* update because we don't differentiate through the bootstrap target, $\overline{\theta}$.

This method, called *deep Q network* (DQN), also uses a replay buffer. It can still be unstable: combining function approximation, bootstrapping, and off-policy learning is tricky. Another problem is that maximizing noisy value estimates can cause overestimation. *Double DQN*[2] addresses that bias by using the online network to select the next action and the target network to evaluate it. We won't have time to go into that.

[2] Van Hasselt, Guez, and Silver, Deep Reinforcement Learning with Double Q-learning.

When the action space is also continuous, we have to make bigger changes. We could potentially store $Q_\theta(s, a)$ with a as an input; but when we need to maximize over a for action selection, we'd have to do a search, which is expensive. In offline RL, we also need to guard against selecting actions whose values are unreliable because they are far from the training data. Better strategies are policy based methods (section 3) or explicit actor-critic methods (section 4).

2.3 *Policy evaluation*

So far, our policies have been deterministic: in state s, the policy chooses action $\pi(s)$. A policy can also give a distribution over actions. We write $\pi(a \mid s)$ for the probability of choosing action a in state s (or a probability density for continuous actions), and execute the policy by drawing an action value a from $\pi(\cdot \mid s)$. This is called a *stochastic policy*. A deterministic policy is the special case that puts all its probability on one action. Sometimes we will be interested in *policy evaluation*: the question is, given a fixed policy π, what is

$$\mathbb{E}_{\tau \sim p_\pi(\mathcal{T} \mid s_0)} \left[\sum_{t=0}^{\infty} \gamma^t r_t \right] \quad ?$$

The expectation now includes randomness in the policy's actions as well as in the environment.

We call this quantity $V^\pi(s_0)$. Unlike Q^*, it tells us how well *this policy* π does, not how well we could possibly do. We can compute this in several ways. We'll highlight two.

Monte Carlo returns The most direct approach to policy evaluation is to run the policy and see what happens! (Sometimes people call this "doing a rollout.") For an episode that terminates at time T, we compute the return from each visited state:

$$g_t = \sum_{k=t}^{T-1} \gamma^{k-t} r_k \ .$$

We use g_t as a training target for $V^\pi(s_t)$. With a table representing V, an incremental update with a small step-size α is

$$V[s_t] \leftarrow V[s_t] + \alpha\big(g_t - V[s_t]\big) \ ;$$

we can also do a gradient update in a neural network. If we can repeatedly start at the same state and run the policy, averaging those returns estimates its value. We can also compute value estimates for states visited along the way. Individual returns may vary a lot, so it may be necessary to average over a large number of runs to get a good estimate.

There is a catch: we have to wait for the return! In a continuing task, the infinite sum is never available. Stopping a rollout after K steps and discarding everything beyond it estimates a truncated return, not the full value. Discounting makes the omitted tail small for large K, but doesn't generally make it zero. For a genuinely terminal state, on the other hand, there are no future rewards to omit.

Recursive temporal-difference estimation We can instead use the same recursive idea that gave us Q-learning. For a fixed policy,

$$V^\pi(s) = \mathbb{E}_{a\sim\pi(A|s)}\Big[R(s,a) + \gamma \mathbb{E}_{s'\sim P(S'|s,a)}\big[V^\pi(s')\big]\Big] \ .$$

For a deterministic policy, $a = \pi(s)$. There is no maximization here: we are evaluating the actions this policy actually chooses.

After observing one transition, we form a target y_t and update:

$$y_t = r_t + \gamma V[s_{t+1}] \ , \qquad V[s_t] \leftarrow V[s_t] + \alpha\big(y_t - V[s_t]\big) \ .$$

This is called TD(0). The difference $y_t - V[s_t]$ is the *TD error*: how much better or worse this step, together with our prediction of what comes next, was than we expected. We use $y_t = r_t$ if the transition ends the task. At an arbitrary rollout cutoff, we keep the next-state value instead.

Now we can learn after every step, without waiting for the end. But we are *bootstrapping*: the target depends on our own current value estimate. An inaccurate estimate of the next state can make the target inaccurate too. Monte Carlo uses an observed return; TD replaces most of that random future with a prediction. This often reduces variance,

at the cost of relying on that prediction. In the finite tabular case, on-policy TD converges to V^π under the usual sufficient-visitation and decreasing-learning-rate conditions.

For large state spaces, we can use a neural network $V_\phi(s)$ instead of a table. For either kind of target, we take a squared-error step:

$$\phi \leftarrow \phi + \alpha\big(y_t - V_\phi(s_t)\big)\nabla_\phi V_\phi(s_t) \ ,$$

where $y_t = g_t$ for Monte Carlo or $y_t = r_t + \gamma V_\phi(s_{t+1})$ for TD, with the terminal-state convention above. We hold the target fixed while taking this derivative, just as we did for DQN.

It is important to note that TD estimation is only guaranteed to work if the Markov property holds; but you can use Monte-Carlo estimation in any domain.

Why learn the value of a policy we already have? Besides measuring how well it performs, knowing the value function can help us improve the policy. It gives us a prediction to compare observed outcomes against. We'll use this as a baseline in section 3.3, and as a critic in section 4.2.

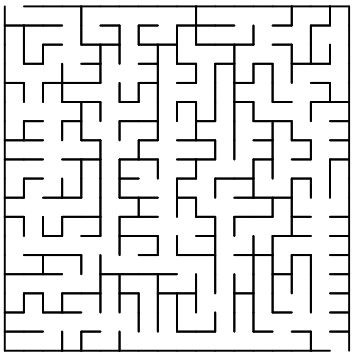
3 *Policy-based methods*

A completely different strategy is to avoid trying to estimate values and instead just do gradient ascent on a parametric representation of the policy to try to improve it directly. We start by picking some parametric class of policies (such as a particular neural-network architecture) with states as input and actions (or distributions over actions) as output. Policy gradient has the advantage of not requiring a Markov state representation: it can use any sort of observations (images, sounds) to optimize a policy in our chosen class. We'll see in section 5.2 that a policy that only sees the current observation can be worse than one that remembers the history. But, as with value-based approaches, when we use a neural network, gradient ascent doesn't guarantee a global optimum.

One concern is that basic policy-gradient estimators are *on policy*: they use data gathered using the policy we are trying to improve. We can't just re-use data from an old policy unchanged.

We'll start with the simplest case to get the intuition and then move to a classic algorithm called REINFORCE.

There are some problems where the policy is a much simpler object than the value function. For example, in a maze with no internal "islands", the policy of always keeping your right (or left) hand on the wall is incredibly simple and will succeed, but estimating the value function has a complex dependence on the size of the maze.



With a cost per step, predicting the remaining return requires accounting for all the detours, even for a simple wall-following policy. Maze by Nandhp, CC BY-SA 3.0.

3.1 *Direct methods*

To start, we assume that we actually know the world dynamics $f(s, a)$, which deterministically returns the next state, s'. We also assume we have a policy $\pi_\theta(s)$, parameterized by weights θ, which produces a as

output. We'll focus on optimizing finite-horizon return, thinking about the distribution of trajectories induced by the policy.

In the case of deterministic world, reward, and policy, our goal is to maximize

$$J(\theta) = \mathbb{E}_{s_0 \sim p_0(S_0)} \left[R(\tau(s_0, \pi_\theta)) \right]$$

where $R(\tau)$ is the sum of the rewards in trajectory τ and the expectation is taken with respect to initial states. Here we're optimizing within our chosen policy class; for a finite horizon, we can also give the policy the number of steps remaining as input.

We can address this problem via stochastic gradient ascent. Assuming the dynamics, reward, and policy are differentiable, and the return gradients are well behaved enough that we can differentiate the average by averaging the derivatives, the starting distribution's independence of θ gives

$$\nabla_\theta J(\theta) = \mathbb{E}_{s_0 \sim p_0(S_0)} \left[\nabla_\theta R(\tau(s_0, \pi_\theta)) \right] \quad .$$

One sufficient condition: for all nearby parameter values, the gradient magnitude is bounded by a function of the initial state whose expectation is finite.

So, we can draw a starting state and roll out π_θ to get a trajectory τ.

Feel free to skip this.

Gradient computation Now we need to compute the gradient of the return, which is the sum of the gradients of the rewards with respect to the parameters:

$$R(\tau) = \sum_{i=0}^{H-1} r_i = \sum_{i=0}^{H-1} R(s_i, \pi_\theta(s_i)) \ , \qquad \nabla_\theta R(\tau) = \sum_{i=0}^{H-1} \nabla_\theta r_i \ .$$

How does r_i depend on θ?

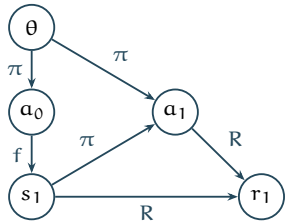
We can make everything one-dimensional: the state s_i, action a_i, and policy parameter θ are scalars, with $a_i = \pi_\theta(s_i)$. We use $d/d\theta$ for the total effect of changing θ, including changes to the states we reach. In $\partial_\theta \pi_\theta(s)$ we hold s fixed; $\partial_s \pi_\theta(s)$ tells us how the action changes when the state changes. Similarly, $\partial_s R$ and $\partial_a R$ differentiate the reward with respect to one input at a time.

We start with r_0. The sampled s_0 doesn't depend on θ, so

$$\frac{dr_0}{d\theta} = \partial_a R(s_0, a_0) \, \partial_\theta \pi_\theta(s_0) \ .$$

Now for r_1, we have to take the dynamics into account:

$$\begin{aligned}
s_1 &= f(s_0, a_0) \\
\frac{ds_1}{d\theta} &= \partial_a f(s_0, a_0) \, \partial_\theta \pi_\theta(s_0) \\
\frac{da_1}{d\theta} &= \partial_\theta \pi_\theta(s_1) + \partial_s \pi_\theta(s_1) \frac{ds_1}{d\theta} \\
\frac{dr_1}{d\theta} &= \partial_s R(s_1, a_1) \frac{ds_1}{d\theta} + \partial_a R(s_1, a_1) \frac{da_1}{d\theta} \ .
\end{aligned}$$



Changing θ changes r_1 through the state, through the action, and through the state's effect on the action. The sampled s_0 is fixed.

The last line has two terms because the reward depends on both state and action. The action derivative itself has two terms: changing θ

changes the policy's output directly, and it changes the state at which we evaluate the policy. These are the three paths in the diagram.

The same chain rule works all the way through the rollout. Starting with $ds_0/d\theta = 0$, we repeatedly apply:

$$\begin{aligned}
\frac{da_i}{d\theta} &= \partial_\theta \pi_\theta(s_i) + \partial_s \pi_\theta(s_i) \frac{ds_i}{d\theta} \\
\frac{dr_i}{d\theta} &= \partial_s R(s_i, a_i) \frac{ds_i}{d\theta} + \partial_a R(s_i, a_i) \frac{da_i}{d\theta} \\
\frac{ds_{i+1}}{d\theta} &= \partial_s f(s_i, a_i) \frac{ds_i}{d\theta} + \partial_a f(s_i, a_i) \frac{da_i}{d\theta} \quad .
\end{aligned}$$

Adding the reward derivatives gives $dR(\tau)/d\theta$. With vector states, actions, and parameters, the same chain rule uses gradients and Jacobians. In practice, automatic differentiation computes the gradient by backpropagating through the rollout.

Summary Whether you compute the gradient by hand or with autodiff, here is a sketch of the policy optimization algorithm.

1: Model-based policy gradient
2: Initialize θ; choose learning rate $\alpha > 0$.
3: loop:
4: draw a start-state value s_0 from p_0
5: "roll out" π_θ for H steps starting in s_0, to get a trajectory
6: $\tau = (s_0, a_0, r_0, \ldots, s_{H-1}, a_{H-1}, r_{H-1}, s_H)$
7: compute $g = \nabla_\theta R(\tau)$ by differentiating through the rollout
8: $\theta \leftarrow \theta + \alpha g$

In this formulation, we used the dynamics and the reward function in order to compute a derivative to improve the policy. What if we don't have them? In low-dimensional parameter spaces we can do finite differencing: we nudge one policy parameter a little, roll out the policy again from the same starting state, and see how much the return changes. Dividing the change in return by the size of the nudge estimates the gradient in that direction. We repeat this for each parameter, then use these estimates to take a gradient-ascent step. We only need to run the system, not differentiate through it, but the extra rollouts get expensive when there are lots of parameters.

This section was just to illustrate a simple thing we'd do if we had a known model or simulator. Now we'll look at more realistic situations, in which we don't.

3.2 REINFORCE

With a deterministic policy, changing a parameter changes the action, so computing a gradient requires knowing how the return changes as a function of the action. A probabilistic policy gives us another option: changing the output *probability distribution* over actions, which we take an expectation over to compute the value of the policy. To differentiate that expectation, we need to know how the weights change—which we can compute—and how good the actions are—which we can estimate by trying them. We never have to differentiate the world dynamics: instead we just draw from the distribution of world-state trajectories our probability induces.

In any given roll-out of a probabilistic policy we get to observe the effects of small deviations from a nominal trajectory; it's a little bit reminiscent of the usefulness of small deviations in the finite-difference method above, but in REINFORCE it will be happening in trajectory space rather than θ space, which is good when our policy is a large neural network.

We assume the policy has the form $\pi_\theta(A \mid s)$: that is, conditioned on a state s, it provides a distribution over actions. We'll start in the case of finite $\mathcal{A}$ so that $\pi_\theta(A \mid s)$ can be a categorical distribution, usually with a softmax parameterization. But we'll see later it can be a parametric continuous distribution (such as a Gaussian) as well. We write $\pi_\theta(a \mid s)$ for its probability (or density) at a particular action value a.

A categorical distribution assigns a probability to each action in a finite set. These probabilities are nonnegative and sum to one. For example, left, right, and stay might have probabilities 0.2, 0.5, and 0.3.

For this derivation, we use trajectories of H steps, with

$$R(\tau) = \sum_{t=0}^{H-1} \gamma^t r_t \ , \qquad J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\mathcal{T})}[R(\tau)] \ .$$

Setting $\gamma = 1$ gives the undiscounted finite-horizon objective from the previous section. The distribution p_θ includes the randomness of the initial state, the policy's actions, and the world.

We'll approach this with gradient ascent, so we need to compute $\nabla_\theta J(\theta)$. We'll do an informal derivation here using sums; integrals play the same role for continuous quantities. We assume the policy probabilities are differentiable and positive on a fixed set of allowed actions, and that we can exchange the gradient and sum. For a fixed trajectory, its return is just a number: changing θ changes the probability that the policy would produce it:

$$\begin{aligned}\nabla_\theta J(\theta) &= \nabla_\theta \sum_\tau p_\theta(\tau) R(\tau) \\ &= \sum_\tau \nabla_\theta p_\theta(\tau) R(\tau) \ .\end{aligned}$$

Now we do what seems like a useless move, multiplying and dividing by $p_\theta(\tau)$:

$$\begin{aligned}&= \sum_\tau p_\theta(\tau) \frac{\nabla_\theta p_\theta(\tau)}{p_\theta(\tau)} R(\tau) \\ &= \sum_\tau p_\theta(\tau) \nabla_\theta \log p_\theta(\tau) R(\tau) \\ &= \mathbb{E}_{\tau \sim p_\theta(\mathcal{T})}[R(\tau) \nabla_\theta \log p_\theta(\tau)] \ .\end{aligned}$$

The last line is an expectation, so we can estimate it by sampling! But don't we need the model to compute $p_\theta(\tau)$?

Here's the nice part. We write $P(s', r \mid s, a)$ for a specific value of the world's joint next-state and reward distribution. For trajectory

$$\tau = (s_0, a_0, r_0, \ldots, s_{H-1}, a_{H-1}, r_{H-1}, s_H) \ ,$$

we have probability

$$p_\theta(\tau) = p_0(s_0) \prod_{t=0}^{H-1} \pi_\theta(a_t \mid s_t) P(s_{t+1}, r_t \mid s_t, a_t) \ .$$

Assuming the initial-state distribution and the world don't themselves depend on θ, taking the log and differentiating gives

$$\nabla_\theta \log p_\theta(\tau) = \sum_{t=0}^{H-1} \nabla_\theta \log \pi_\theta(a_t \mid s_t) \ .$$



No model derivatives!

All the unknown model terms disappear! Yay! We hold the sampled states and actions fixed in this derivative; we don't backpropagate through the world. So

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\mathcal{T})} \left[R(\tau) \sum_{t=0}^{H-1} \nabla_\theta \log \pi_\theta(a_t \mid s_t) \right] \ . \tag{2}$$

To turn this into an algorithm, we estimate the expectation with a batch of trajectories, then take a gradient-ascent step.

REINFORCE

1: Initialize θ; choose learning rate $\alpha > 0$, batch size N, horizon H, and discount γ.
2: loop:
3: collect N independent trajectories $\tau^{(i)}$ of H steps using the current policy π_θ and initial states from p_0
4: compute each trajectory's return

$$R(\tau^{(i)}) = \sum_{t=0}^{H-1} \gamma^t r_t^{(i)}$$

5: estimate the gradient, holding sampled states and actions fixed:

$$\widehat{g} = \frac{1}{N} \sum_{i=1}^{N} R(\tau^{(i)}) \sum_{t=0}^{H-1} \nabla_\theta \log \pi_\theta(a_t^{(i)} \mid s_t^{(i)})$$

6: $\theta \leftarrow \theta + \alpha \widehat{g}$

Why the estimate is unbiased We write $\widehat{G}$ for the random gradient estimator whose realized value is $\widehat{g}$. For a fixed θ, it is unbiased:

$$\mathbb{E}_{\widehat{g} \sim p(\widehat{G} \mid \theta)}[\widehat{g}] = \nabla_\theta J(\theta) \ .$$

Here $p(\widehat{G} \mid \theta)$ is the distribution of gradient estimates from fresh batches sampled under that fixed policy. As we update θ, successive batches estimate gradients at different parameter values.

3.3 REINFORCE *with baseline*

The update above can be very noisy. What we'd really like to know is: was this outcome better or worse than we'd normally expect from here? If we can estimate, even relatively inaccurately, the expected return of a policy from the current state, we can use it as a *baseline*. A well-chosen baseline can reduce the variance of our gradient estimates while *leaving the expected gradient unchanged*; an arbitrary baseline need not reduce variance.[3]

First, let's stop giving an action credit for rewards that arrived *before we took it*. Once we condition on the history up to choosing A_t, those earlier rewards are already fixed, and the conditional mean of $\nabla_\theta \log \pi_\theta(A_t \mid S_t)$ is zero. Their contribution to the expected gradient is therefore zero. We define the *return to go* at time t as

$$G_t = \sum_{k=t}^{H-1} \gamma^{k-t} R_k \ .$$

[3] Tengyu Ma, CS229 lecture notes, Part XV, *Policy Gradient (REINFORCE)*.

What do we mean by variance reduction? Remember that we're trying to move along the gradient of the expected return of our policy. And we've been able to write that as an expectation over the gradient of the log probability of a trajectory times its return. We can't compute that expectation exactly, so we do it by sampling. In the limit of a large number of samples per batch, we recover the expectation. But with a small number of samples per batch, each individual step can have a lot of error compared to the exact gradient direction: the variance of the updates is high. This generally means we need to use a smaller step size, so we need to do more iterations, etc. If we can reduce the variance in the individual gradient samples, learning can be faster and more stable.
It's okay to skip this derivation; the idea and the algorithm are the important parts.

Its value on a sampled trajectory is $g_t = \sum_{k=t}^{H-1} \gamma^{k-t} r_k$. Then equation 2 becomes

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\mathcal{T})} \left[\sum_{t=0}^{H-1} \gamma^t g_t \nabla_\theta \log \pi_\theta(a_t \mid s_t) \right] .$$

Be careful about the outer γ^t: the sum G_t discounts from time t forward, but our objective discounts from time zero, so we need this term to keep the overall discounting correct.

We now choose a baseline $b_t(s)$. It can be *any quantity* that doesn't depend on the action we're about to sample. We hold it fixed when computing the policy gradient.

The key fact is that, at any fixed state s, the average log-probability gradient is zero:

$$\begin{aligned} \mathbb{E}_{a \sim \pi_\theta(A|s)}[\nabla_\theta \log \pi_\theta(a \mid s)] &= \sum_a \pi_\theta(a \mid s) \frac{\nabla_\theta \pi_\theta(a \mid s)}{\pi_\theta(a \mid s)} \\ &= \nabla_\theta \sum_a \pi_\theta(a \mid s) = \nabla_\theta 1 = 0 . \end{aligned}$$

Increasing some action probabilities has to be balanced by decreasing others: they still have to add up to one. Concretely, applied to a baseline,

$$\mathbb{E}_{a \sim \pi_\theta(A|s)}[b_t(s) \nabla_\theta \log \pi_\theta(a \mid s)] = b_t(s)\, 0 = 0 ,$$

so we can subtract it without changing the expected gradient:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\mathcal{T})} \left[\sum_{t=0}^{H-1} \gamma^t \big(g_t - b_t(s_t)\big) \nabla_\theta \log \pi_\theta(a_t \mid s_t) \right] . \quad (3)$$

We can use the sample average of this expression for $\widehat{g}$ and use it in our gradient update $\theta \leftarrow \theta + \alpha \widehat{g}$.

What should we use as a baseline? We want to reduce the variance of the gradient estimate, whose terms multiply $g_t - b_t(s_t)$ by a log-probability gradient. At a fixed state, subtracting a constant does not change the variance of G_t itself. Choosing its conditional mean as the baseline does minimize the expected squared weight, $\mathbb{E}[(G_t - b_t(s))^2 \mid S_t = s]$. So a useful baseline is the expected return from state s under the current policy, $V_t^{\pi_\theta}(s) = \mathbb{E}_{g_t \sim p_{\pi_\theta}(G_t \mid S_t = s)}[g_t]$. Then $g_t - b_t(s_t)$ says how much better or worse this rollout was than expected. This often reduces gradient variance, but is not necessarily the minimum-variance baseline: the log-probability gradient also varies with the sampled action.

We can learn an approximation $b_\phi(s, t)$ with Monte Carlo policy evaluation (section 2.3): we use (s_t, t) as input and the observed g_t as the target, minimizing squared prediction error.

REINFORCE **with a learned baseline**

Problem inputs: environment *env* (including its initial-state distribution p_0), horizon H, and discount factor γ.

1: Choose batch size N and step size $\alpha > 0$; initialize policy weights θ and baseline weights ϕ (a zero baseline is fine).
2: loop:
3: roll out π_θ in *env* for H steps to get N independent trajectories $\tau^{(i)}$, $i = 1, \ldots, N$, each starting from a fresh state drawn from p_0
4: compute returns to go, backward along each trajectory:

$$g_H^{(i)} = 0, \qquad g_t^{(i)} = r_t^{(i)} + \gamma g_{t+1}^{(i)}$$

5: subtract the baseline value estimate from the actual return to go, giving a value *weight* for step t of trajectory i:

$$w_t^{(i)} = g_t^{(i)} - b_\phi(s_t^{(i)}, t)$$

6: estimate the policy gradient, holding all $w_t^{(i)}$ fixed:

$$\widehat{g} = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{H-1} \gamma^t w_t^{(i)} \nabla_\theta \log \pi_\theta(a_t^{(i)} \mid s_t^{(i)})$$

7: $\theta \leftarrow \theta + \alpha \widehat{g}$
8: update the baseline by regressing observed returns to go on state and time, for use on the next batch:

$$\phi \leftarrow \text{regress}\left(\left\{ \left((s_t^{(i)}, t),\ g_t^{(i)} \right) \right\}_{\substack{i=1,\ldots,N \\ t=0,\ldots,H-1}} \right)$$

The constant step size α keeps the pseudocode simple. We can instead use a step-size schedule or an adaptive optimizer such as Adam. These, like the batch size N, are algorithm choices; *env*, H, and γ specify the problem.

Here regress returns weights for $b_\phi(s, t)$ by minimizing squared prediction error on the supplied input–target pairs.

It is important that the baseline be an estimate of what we expected *before seeing this batch's outcomes*. If we first fit it to this batch, a lucky reward can raise both the observed return and the baseline we subtract from it. The baseline then depends indirectly on the sampled action, even though action is not one of its inputs, so the cancellation in our derivation need not hold.

4 *Combined methods*

Often we can combine elements of value-based and policy-based methods to get something better. We can even combine behavior cloning and RL.[4]

[4] Compact but more detailed explanations of all of these methods, with good citations, are available at reinforcement-learning.com.

4.1 *PPO is* REINFORCE *plus baseline plus clipping*

Since we were just talking about REINFORCE with a baseline, we will discuss *proximal policy optimization* (PPO). On the one hand, it would be great if we could squeeze more useful information out of a batch of policy roll-outs by re-using them; but as soon as we make any update to our policy parameters, these roll-outs are no longer a completely accurate representation of the trajectory distribution induced by the policy. PPO tries to get more learning out of each batch of experience without changing the policy so much that the batch stops being useful.[5] Let's call the policy that collected the batch $\pi_{\theta_{\text{old}}}$.

[5] Schulman et al., Proximal Policy Optimization Algorithms.

The *advantage* of taking action a is how much better it is than the policy's average choice at that state:

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s) \quad .$$

Notation reminders: $V^{\pi}(s)$ is the true expected return from state s when we follow policy π; $Q^{\pi}(s,a)$ is the expected return if we first take a in s, then follow π. The neural network $V_{\phi}(s)$, with weights ϕ, approximates $V^{\pi_{\theta_{\text{old}}}}(s)$. In a finite-horizon task, both the value function and its neural-network approximation also depend on the time remaining. We leave that input implicit here. The observed return to go from step t is $g_t = \sum_{k=t}^{H-1} \gamma^{k-t} r_k$.

We don't know these true values, but a simple advantage estimate for a sampled action is

$$\widehat{\text{adv}}_t = g_t - V_{\phi}(s_t) \quad .$$

This is the return-minus-baseline weight from section 3.3.

Assume we collected a batch of data using $\pi_{\theta_{\text{old}}}$ and are trying to use it to approximate the policy gradient for a nearby policy π_{θ}. We can compare the new policy's probability of the sampled action to its old probability:

$$\rho_t(\theta) = \frac{\pi_{\theta}(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} \quad .$$

We saw a similar importance-sampling ratio in the behavior cloning notes: it is a target probability divided by a data-collection probability.

Before clipping, we can form an update by averaging contributions from the recorded steps. A single step contributes

$$\theta \leftarrow \theta + \alpha \gamma^t \nabla_{\theta} \left[\rho_t(\theta) \widehat{\text{adv}}_t \right] \quad .$$

Why is this okay?

When differentiating, we hold the sampled state and action, the old policy's probability, and $\widehat{\text{adv}}_t$ fixed. Using $\nabla_{\theta} \pi_{\theta} = \pi_{\theta} \nabla_{\theta} \log \pi_{\theta}$, we get

$$\begin{aligned}
\nabla_{\theta} \left[\rho_t(\theta) \widehat{\text{adv}}_t \right] &= \widehat{\text{adv}}_t \nabla_{\theta} \rho_t(\theta) \\
&= \widehat{\text{adv}}_t \frac{\nabla_{\theta} \pi_{\theta}(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} \\
&= \widehat{\text{adv}}_t \frac{\pi_{\theta}(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)} \nabla_{\theta} \log \pi_{\theta}(a_t \mid s_t) \\
&= \widehat{\text{adv}}_t \rho_t(\theta) \nabla_{\theta} \log \pi_{\theta}(a_t \mid s_t).
\end{aligned}$$

To check that this is sensible, we can evaluate at $\theta = \theta_{\text{old}}$, where $\rho_t = 1$, and get

$$\nabla_{\theta} \left[\rho_t(\theta) \widehat{\text{adv}}_t \right] \Big|_{\theta = \theta_{\text{old}}} = \widehat{\text{adv}}_t \; \nabla_{\theta} \log \pi_{\theta}(a_t \mid s_t)|_{\theta = \theta_{\text{old}}} \quad .$$

Substituting $\widehat{\text{adv}}_t = g_t - V_\phi(s_t)$ on the right shows that this is the per-step REINFORCE-with-baseline update in equation 3, before multiplying by γ^t and averaging over the batch. So the ratio gives us an objective whose gradient initially agrees with that update.

The ratio only accounts for changing the action probabilities at states already in the batch. A changed policy would also reach different states on later steps. Reweighting the recorded actions doesn't supply experience at those new states, so this objective becomes less reliable as the policy moves away from the one that collected the batch.

If we keep improving this objective on the same batch, we may make some recorded actions much more or less likely and change the policy too far. Clipping limits how much benefit a sample can give us for pushing its probability in the preferred direction. PPO maximizes

$$L_{\text{clip}}(\theta) = \widehat{\mathbb{E}}_{\tau \sim p_{\theta_{\text{old}}}(\mathcal{T})} \left[\sum_t \gamma^t \ell_t(\theta) \right] ,$$

where

$$\ell_t(\theta) = \min\big(\rho_t(\theta)\widehat{\text{adv}}_t, \text{clip}(\rho_t(\theta), 1-\epsilon, 1+\epsilon)\widehat{\text{adv}}_t\big)$$

Here $0 < \epsilon < 1$ sets the clipping range, and $\text{clip}(x, l, u) = \min(\max(x, l), u)$. $\widehat{\mathbb{E}}$ averages over the collected trajectories.

The effect of clipping is to stop rewarding a probability increase past $\rho_t = 1 + \epsilon$ for a positive advantage, or a decrease past $\rho_t = 1 - \epsilon$ for a negative advantage. The minimum still penalizes changes in the wrong direction. Clipping limits the incentive for a large change; it does not impose a hard bound on the updated ratio.

Our implementation of PPO in HW6 uses $\gamma = 1$ and averages the clipped objective over all recorded state–action pairs. For a fixed batch, this only rescales the objective above.

The pseudocode below uses whole-batch updates for simplicity; in practice we usually take gradient steps on minibatches drawn from the same batch. Many implementations also add an entropy bonus (see section 4.3) to encourage the policy to keep some randomness in its actions. PPO is still usually called on-policy because it reuses a recent batch for a limited time, rather than learning from an arbitrary collection of old experience.

PPO with complete-episode returns

Problem inputs: environment *env* (including its initial-state distribution p_0), horizon H, and discount factor γ.

1: Choose batch size N, clipping tolerance ϵ, step size $\alpha > 0$, and number of policy updates per batch M; initialize policy weights θ and value-network weights ϕ.
2: loop:
3: $\quad \theta_{\text{old}} \leftarrow \theta$
4: $\quad$ roll out $\pi_{\theta_{\text{old}}}$ in *env* to collect N complete H-step episodes, each starting from a fresh state drawn from p_0
5: $\quad$ compute g_t backward from $g_H = 0$ using $g_t = r_t + \gamma g_{t+1}$
6: $\quad$ store $p_t^{\text{old}} = \pi_{\theta_{\text{old}}}(a_t \mid s_t)$ and $\widehat{\text{adv}}_t = g_t - V_\phi(s_t)$
7: $\quad$ repeat M times:
8: $\quad\quad \theta \leftarrow \theta + \alpha \nabla_\theta L_{\text{clip}}$, using the fixed p_t^{old} and advantages
9: $\quad$ fit $V_\phi(s_t)$ to the fixed targets g_t by squared-error regression
10: $\quad$ discard the batch and collect fresh experience next time

Estimating advantages without waiting for a complete episode The algorithm above uses observed returns all the way to the end of each episode. These can be noisy, and in a continuing task there is no episode end to wait for. PPO usually uses *generalized advantage estimation* (GAE), which combines observed rewards with predictions from the value network.[6] This value network is also called the *critic*, because it evaluates how well the policy is doing. In the formulas below, all value predictions come from the network before this batch's updates, $V_{\phi_{\text{old}}}$.

[6] Schulman et al., High-Dimensional Continuous Control Using Generalized Advantage Estimation.

We first compute the one-step prediction errors from section 2.3, then add them with decreasing weights:

$$\delta_t = r_t + \gamma V_{\phi_{\text{old}}}(s_{t+1}) - V_{\phi_{\text{old}}}(s_t) \ ,$$
$$\widehat{\text{adv}}_t = \sum_{\ell=0}^{K-1} (\gamma\lambda)^\ell \delta_{t+\ell} \ .$$

Here K is the number of steps from t to the end of the collected rollout segment, stopping at any episode boundary, and $0 \leqslant \lambda \leqslant 1$. We use zero for the next-state value at a true terminal state; if we merely stop collecting before the task ends, we use the network's prediction for the remaining return instead.

With $\lambda = 0$, we use one reward and the network's prediction of what follows. With $\lambda = 1$ and a complete episode, the intermediate

value predictions cancel, leaving $g_t - V_\phi(s_t)$. Intermediate values trade reliance on the network against noisy long returns; an inaccurate prediction can introduce bias. We compute these advantages before updating either network, then keep them fixed in the PPO updates. For value-network regression, a common fixed target is $\widehat{\text{adv}}_t + V_{\phi_{\text{old}}}(s_t)$, using the value prediction saved when the batch was collected.

The PPO implementation used in HW6 differs from this in some implementation aspects; it is described in detail in appendix A.

4.2 Off-policy actor-critic methods

The idea of an "actor-critic" method is that we have two networks: the "actor" network represents a policy and the "critic" network represents its value function. PPO is a kind of actor-critic method: its learned value function helps estimate which actions were better than expected. But PPO uses recent rollouts, then collects a new batch, so it's *on-policy*. Off-policy actor-critic methods can keep learning from a replay buffer of experience collected by earlier policies.

We saw value-based methods in section 2 but observed that for continuous actions, maximizing $Q(s, a)$ over a can be difficult. We can learn a policy to do this job! With a deterministic actor $a = \pi_\theta(s)$, we train $Q_\phi(s, a)$ to predict the return from taking action a and then following the actor. The two networks have different training objectives: the critic minimizes prediction error, while the actor maximizes the return predicted by the critic.

This idea underlies *deep deterministic policy gradient* (DDPG; Lillicrap et al.). Vanilla DDPG is sensitive to tuning and critic errors. TD3 improves its robustness; SAC, introduced next, uses a stochastic actor and an entropy bonus. TD3 and SAC are generally more useful starting points for applications today.

Critic: minimize prediction error in ϕ We sample recorded transitions (s, a, r, s') from the replay buffer $\mathcal{D}$ and form a target from the observed reward and a prediction of the remaining return:

$$y = r + \gamma \bar{Q}(s', \bar{\pi}(s')) \ .$$

Here $\bar{Q}$ and $\bar{\pi}$ are fixed target versions of the critic and actor. For a true terminal transition, the bootstrap term is zero, so $y = r$. Holding these targets fixed, we minimize

$$L_Q(\phi) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}} \left[(Q_\phi(s, a) - y)^2 \right] \ .$$

The critic update changes ϕ to make its predictions agree better with these targets; it does not differentiate through y. The target networks are updated separately as learning proceeds.

Actor: maximize predicted return in θ Holding the critic fixed, we adjust the actor by gradient ascent on

$$L_\pi(\theta) = \mathbb{E}_{s\sim\mathcal{D}}[Q_\phi(s, \pi_\theta(s))] \ ,$$

where $s \sim \mathcal{D}$ means sampling a stored state. We differentiate through the critic's action input and the actor, using the chain rule. The critic's

weights ϕ stay fixed during this update, but its predicted value changes as the actor changes its action. The learned critic supplies the derivative; we don't need to differentiate through the world. We alternate these critic and actor updates.

Reusing experience can substantially reduce the amount of interaction needed. The catch is that the actor may find actions where the critic is optimistically wrong, and bootstrapping can amplify those errors. We still need exploration and adequate coverage of useful states and actions.

Policy search has an important advantage when the observations are not Markov, even if we restrict ourselves to a *memoryless* policy $\pi_\theta(a \mid o)$. We can estimate its return gradient from complete rollouts and search for the best policy in that class without assuming that o is a Markov state. Standard value-based methods that apply Bellman updates directly to o lose that justification: the expected future can depend on history that o does not capture, so their updates need not lead to the best memoryless policy. Adding memory can improve what is achievable, but is not required for this policy-search argument.

This makes policy search, including PPO, attractive in non-Markov settings. PPO is also useful when simulation is cheap and highly parallel; off-policy methods such as TD3 and SAC are often attractive when a Markov state representation is available and experience is expensive, as on a physical robot. Neither approach is uniformly better.[7]

The unbiased-gradient argument applies to return-based REINFORCE, including an action-independent baseline. PPO is a practical extension, but clipping and bootstrapped GAE introduce approximations; a critic using non-Markov observations can bias GAE. Neither method guarantees a global optimum.

[7] Practical algorithm-selection guidance appears in Stable Baselines3. Also this repo is a great place to go for reference implementations and hyper-parameter settings for many standard RL methods.

4.3 Maximum entropy RL

For a finite MDP with the usual expected discounted-return objective, there is an optimal *deterministic* policy. The Markov property alone is not enough to guarantee this for every problem formulation. Since deterministic policies are simpler, and since we're not trying to deal with training data from a non-deterministic multi-modal demonstrator (as in BC), then it might seem like we should restrict our attention to deterministic policies when we can. In policy search, making our policies stochastic can smooth the expected-return objective and provide exploration during optimization. For similar reasons, even doing value-based RL in a Markovian environment where we know there is a deterministic optimum, it can be useful to operate in a probabilistic policy space.

Maximum entropy RL rewards a policy both for obtaining high returns and for keeping options open. We define the entropy of policy π at state s as

$$\mathcal{H}(\pi(A \mid s)) = -\mathbb{E}_{a \sim \pi(A \mid s)}[\log \pi(a \mid s)] \ ,$$

Entropy measures how uncertain the policy's action is. For a finite action set, it is zero when the policy always chooses one action, and higher when probability is spread more evenly among actions. An entropy bonus encourages the policy to keep several options open.

Optimizing in a larger space than strictly necessary can sometimes help, as with over-parameterized neural networks. This is an analogy, not a general guarantee of avoiding poor local optima.

For continuous actions this is differential entropy.

and the overall objective, including both reward and entropy, as

$$J_\beta(\pi) = \mathbb{E}_{\tau \sim p_\pi(\mathcal{T})}\left[\sum_{t=0}^{\infty} \gamma^t \left(r_t + \beta \mathcal{H}(\pi(\cdot \mid s_t))\right)\right] .$$

The temperature $\beta > 0$ controls the tradeoff; we use β so it won't be confused with the learning rate α. This is a change to the objective, not just temporary exploration noise. If one action is much better, reward can outweigh entropy; if several are comparably good, the policy has a reason to keep using them.

Soft actor-critic (SAC) is an effective off-policy actor-critic method for this objective.[8] A common version uses a stochastic actor policy, a replay buffer, and two critics.

[8] Haarnoja et al., Soft Actor-Critic Algorithms and Applications.

Summary: value, policy, and combined methods The main tradeoffs are how much experience we can reuse, what we assume about the observations, and how much we rely on learned value predictions.

Method	**Experience, strengths, and limitations**
Value-based (e.g., Q-learning)	Off-policy replay reuses experience. Bellman updates need a Markov state; maximizing over continuous actions can be difficult.
Policy search (REINFORCE, PPO)	On-policy batches; PPO reuses each batch for a limited time. Return-based gradients work even for memoryless policies with non-Markov observations, but collecting fresh rollouts can be expensive.
Off-policy actor-critic (e.g., TD3, SAC)	Replay reuses experience, and an actor learns continuous actions. The actor can exploit critic errors, which bootstrapping can amplify; these value updates rely on a Markov state representation.

Cheap, parallel simulation makes fresh PPO rollouts practical; expensive interaction makes replay attractive when we have a Markov state representation. These categories overlap: PPO is also an actor-critic method. Its clipping and bootstrapped GAE are approximations, and a critic using non-Markov observations can bias GAE. The return-based policy-gradient argument still applies without Markov observations; neither approach guarantees finding a global optimum.

4.4 Combining RL and BC

Behavior cloning can take immediate, direct advantage of some expert demonstrations, but has trouble when policy execution goes outside the

part of the state space covered in the data. RL can be good at exploring and learning what to do in a larger part of the space, but especially when the reward is "sparse" (really, distant from the state where the important decision needs to be made) it can have trouble finding an even remotely good way of behaving.

Behavior cloning can give us a useful starting point for RL: the robot already has some idea of what successful behavior looks like. RL lets it practice, recover from its own mistakes, and optimize returns on novel states. We'll explore some concrete strategies for combining them.

RL with BC-guided exploration Exploration is a significant difficulty in RL, especially with sparse rewards and an initial random policy. One thing to do with a BC policy is simply use it to guide exploration in early phases of an off-policy RL algorithm: start by mostly executing the BC policy, and then gradually mix in actions chosen by the learned policy (or by maximizing the learned action-value function) and random exploration.

Initializing with BC, improving with RL The next strategy is to train on demonstrations first, then use the cloned policy to initialize PPO, SAC, or another RL method that has an explicit policy representation. A policy that can occasionally succeed may be easier to improve with a sparse success reward than one that has never come close. But early RL updates can sometimes destroy the useful behavior encoded in the weights of the BC policy, so we often keep some imitation training in the loop.

One simple objective is to select a policy to maximize a weighted combination of reward and the likelihood assigned to the demo data by that policy.

$$\max_{\theta} \; J_{RL}(\theta) + \lambda_{BC} \mathbb{E}_{(s,a^*)\sim\mathcal{D}_{demo}}[\log \pi_\theta(a^* \mid s)] \quad .$$

The RL term rewards outcomes; the BC term keeps the policy practicing demonstrated actions. Reducing λ_{BC} over time gives RL more freedom to exceed the performance of the demonstrator. Too much imitation can preserve the teacher's mistakes; too little can cause the policy to forget how to get near a reward. Demonstration-augmented policy gradient (DAPG) is one example of this general strategy.[9]

[9] Rajeswaran et al., Learning Complex Dexterous Manipulation with Deep Reinforcement Learning and Demonstrations.

Demonstrations can also provide the first entries in an off-policy replay buffer, the collection of stored transitions introduced in section 2.2, if we have transitions and can supply their rewards.

Keeping the BC policy and learning a correction With continuous actions, another option is a *residual policy*:

$$a_t = \pi_{BC}(s_t) + u_\theta(s_t) \quad .$$

We freeze the cloned policy π_{BC} and initialize the correction u_θ near zero. RL learns adjustments to the policy. This process can preserve a useful skill while reducing how much RL has to learn.[10] Of course, if the base policy needs a completely different strategy, small corrections may not be enough.

[10] Alakuijala et al., Residual Reinforcement Learning from Demonstrations.

5 *Important concerns*

So far we've mostly worried about how to update the weights in value and/or policy networks. On a robot, a lot of the work is in arranging for those updates to have something useful to learn from. What experience can we collect? What can the robot actually observe? And how much of what it learns in simulation will survive contact with the real world?

5.1 *Exploration*

For a peg-insertion task, the reward might be 1 for putting the peg in the hole and 0 otherwise. If the robot never gets the peg near the hole, all its experience says "nothing good happened." There isn't much information about how to improve. This is a *sparse reward*: useful feedback occurs rarely. Randomly wiggling every joint is unlikely to ever, even accidentally, solve the problem.

In robotics, we usually give exploration some help:

- We can use demonstrations, a BC policy, or a hand-written controller to get the robot into states where small changes in its actions sometimes succeed, so there is some signal to learn from.
- We can begin with the peg close to the hole, then move it farther away as the robot improves. This is a *curriculum*: a progression of training problems. In simulation, we can also start from intermediate states along a successful demonstration.
- We can reward progress toward an objective, such as approaching the object, grasping it, and moving it toward the target. These extra rewards are called *reward shaping*. They can make learning much easier, but unless designed carefully, may warp the objective function in ways that produce undesired behavior.

Ng, Harada, and Russell, Policy invariance under reward transformations: Theory and application to reward shaping (1999).

The action representation may have a substantial effect on exploration. Small changes in a requested gripper position, with a controller underneath that moves the arm, can be much more useful and informative than small changes in raw joint torques.

On hardware, we also face the problem of resetting the environment. If the robot can reach states (stuck pegs, broken pegs, dropped objects) that it can't recover from, then humans have to pick up after the robot (or design elaborate mechanisms for resetting).

5.2 Partial observability

A robot rarely sees the full environment state. We can't accurately judge a novel object's mass from an image. We call the sensor input an *observation*, o_t, to distinguish it from the world state s_t. Two states that need different actions can produce the same observation. A box might look the same whether it is empty or full, but require a different force to lift.

In general, optimal behavior under partial observability requires the ability to condition the choice of the next action on the whole previous history of actions and observations. So, for example, after lifting a box, the robot has some information about its mass. We can give a policy a short window of observations and past actions as its input, or use a *recurrent network* to represent the policy and/or value function. Training such networks is often difficult. Memory doesn't automatically recover a Markov state. Policy search can, in principle, learn to gather information: looking behind an obstacle, touching a surface, or trying a gentle push before a hard one. But the benefit may become apparent much later than the action, so discovering that behavior can be hard.

Learn about POMDPs! Take 6.4110.

5.3 Sim to Real

Simulation lets us collect enormous amounts of experience without damaging anything and reset failed attempts cheaply. However, RL, like other optimization methods, will readily exploit any attractive "loopholes" in the objective. So a policy learned in simulation may rely on details that the simulator gets wrong: friction, contact forces, motor delays, or the appearance of an object. The difference between simulation and reality is often called the *reality gap*.

One way to address this problem is through *domain randomization*, which means varying the simulated world during training. We can vary object masses and friction, camera positions and lighting, sensor noise and action delays. The hope is that the policy learns behavior that works in expectation over all these variations, marginalizing over the unobserved domain properties, including aspects of the real robot.[11] The variations should cover plausible uncertainty: too little variation leaves the policy brittle, but too much can make learning harder. If no single policy can succeed in all those environments, optimizing average performance necessarily involves compromises.

[11] Tobin et al., Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World.

Another useful trick is *teacher–student training*. We first train a teacher policy in simulation with access to information such as exact object poses, contact forces, or friction. This extra information is called *privileged information*: it is available during training but won't be available to the deployed robot. Policies are generally much easier to learn with complete observations, which often constitute a Markov state, so

we hope to learn a good teacher policy, possibly using a value-based method. We then train a student to match the teacher's actions using only the robot's actual observations, possibly with memory, using supervised imitation learning. The student can subsequently be improved with policy-gradient learning. Training a student to imitate a teacher is called *distillation*.[12]

[12] Rusu et al., Policy Distillation.

6 Model-based RL

So far, we have studied methods that answer "what should I do?" (policies) and "how good is this action in the long term?" (value functions). Given an action-value function Q, choosing an action can be as simple as comparing its values, although maximizing over continuous actions can still be difficult. Another approach is to learn to answer the question "what would happen if I did this?" We can use the answers to this question to compare possible futures before acting. In *model-based RL*, predictions from a model of the environment help us learn or improve behavior. The model can be given, learned from experience, or a combination of the two.

One driving motivation behind model-based methods is that we would like to be able to learn about the environment in a way that generalizes over the particular "task" that the robot is supposed to solve. Policies and value functions trained for a single task have the reward function "baked in": changing R generally requires updating them, although we needn't start learning from scratch. A transition model can be reused when the reward changes but the dynamics do not.

We'll start with some ideas that aren't exactly "model-based," but that still generalize over (some aspects of) R. Then we'll move into even more general model-based methods, using a transition model T and a reward function R, either of which may be learned or supplied.

6.1 Goal conditioning and distance metrics

One important subclass of RL problems consists of *stochastic shortest path* (SSP) problems: the setting is that the robot's objective is to reach some element of a goal set $\mathcal{G} \subset \mathcal{S}$, and to incur as little expected total cost as possible, assuming there is a cost function $C(s, a)$ indicating the cost for taking action a in state s. These are undiscounted MDPs ($\gamma = 1$) that terminate at the goal. For simplicity, we assume each nonterminal step costs at least some $c_{\min} > 0$, and that a policy can reach the goal with finite expected total cost. Thus, avoiding the goal forever has infinite cost. RL methods can solve these problems, but they have special structure we can exploit.

In particular, we are interested in the class of problems where the robot won't know what the goal is until it's deployed: you definitely might want to be able to tell your robot what you want it to do!

Goal-conditioned RL In *goal-conditioned RL*, the goal g is another input: we learn $\pi(a \mid s, g)$ and $Q(s, a, g)$. A goal might be a desired object pose, a location, or an image of a completed task. Let $\mathcal{G}(g)$ be the set of states that satisfy goal g; for an exact state goal, $\mathcal{G}(g) = \{g\}$. Then we can define

$$R(s, a, g) = \begin{cases} 0, & s \in \mathcal{G}(g), \\ -C(s, a), & s \notin \mathcal{G}(g). \end{cases}$$

Reaching $\mathcal{G}(g)$ terminates the episode. We pay for the action that takes us into the goal set, and receive no further rewards after arrival. Maximizing the undiscounted return therefore minimizes the expected cost of reaching the goal.

We could just do regular RL where the state space is understood to be (s, g), but in fact, we observe that experience we get when trying to achieve one goal g_1 also teaches us about how to achieve goal g_2, as long as the physical dynamics stay the same. So, if we're using an *off-policy* RL method, a transition $((s, g_1), a, r, (s', g_1))$ can also be relabeled as

$$((s, g_2), a, R(s, a, g_2), (s', g_2)).$$

We also recompute whether s' is terminal for g_2: if $s' \in \mathcal{G}(g_2)$, the learning target has no continuation value. Transitions starting in $\mathcal{G}(g_2)$ are omitted, since that task would already have ended. In practice, we sample alternative goals rather than enumerate the whole goal space.

Hindsight experience replay stores additional versions of an experience with a goal that was actually reached, recomputing the rewards for that goal.[13] In stochastic environments, choosing goals based on the observed outcomes can bias learning toward lucky outcomes.

[13] Andrychowicz et al., Hindsight Experience Replay.

See Li, Pinto, and Abbeel, Generalized Hindsight for Reinforcement Learning, section 4.4.

Quasimetric learning For exact state goals with the same dynamics and step costs, we can think of the *minimum expected total cost* of reaching g from s as a kind of "distance", $d(s, g) = -V^*(s, g)$. This quantity behaves a little like a mathematical distance, but needn't be symmetric. A robot can drop an object off a table much more easily than it can get it back. On a set of states where each can be reached from every other with finite expected cost, these distances satisfy, for all intermediate states u,

$$d(s, g) \leqslant d(s, u) + d(u, g) \quad :$$

going directly as cheaply as possible can't cost more than the cheapest path constrained to go through an intermediate state u. A nonnegative distance that is zero exactly when the two states coincide and obeys this

In a deterministic problem this is the cost of a shortest path; in a stochastic problem we optimize a policy, not a single realized path.

triangle inequality, but need not be symmetric, is called a *quasimetric*. If a goal cannot be reached with finite expected cost, its distance is infinite. This structure can help us learn relationships among many goals and choose intermediate goals for planning.

Quasimetric RL builds this structure into learned representations of goal-reaching costs, as described by Wang et al.

Wang and collaborators applied quasimetric RL to simulated Fetch-arm reaching, pushing, and sliding tasks, including reaching and pushing from images. More recently, Zheng, Myers, Eysenbach, and Levine used multistep quasimetric learning for real robot manipulation, combining parts of different recorded trajectories to reach goals requiring longer sequences of actions.[14]

[14] Zheng et al., Multistep Quasimetric Learning for Scalable Goal-conditioned Reinforcement Learning.

6.2 *Learning models*

If we know a reward model $R(s, a)$ and a transition model $T(s, a, s') = P(s' \mid s, a)$ then we can solve for an optimal policy (using value iteration, policy iteration, or RL in a simulation made with R and T). These methods yield a complete policy $\pi(a \mid s)$. It's also possible to use inference-time reasoning to compute the next a on the fly.

Sigh. The word "model" has become almost useless in our field. Webster's 1913 dictionary gives "A miniature representation of a thing, with the several parts in due proportion." But now it can stand for a neural network (which might represent a transition model or a policy, or something else entirely). And now we have "world model" which sometimes means a representation of a world state s and sometimes a transition model $P(s' \mid s, a)$.

We will generally say "neural network" when we mean that, and reserve "model" as a thing that stands for something else.

We'll start with model learning and then talk about how to use the models we learn.

6.2.1 *Reward models*

We use a reward model to assign immediate utility to states (or state-action pairs), representing what we want the robot to do. Sometimes we can write the reward directly, using a negative distance to a known target or a penalty for using energy. But "set the table nicely" or "hand me the cup comfortably" is harder to turn into a numerical score. It may be easier for a person to recognize or demonstrate good behavior than to program a function that recognizes it.

We can learn a reward function from human feedback by asking a person to label images as successful or unsuccessful, or to compare two attempts. We then train a network R_ψ to predict the labels or scores by adjusting its parameters ψ.

Learning from comparisons We'll start with a person comparing two final states, s_A and s_B, for the same task. How can we use this information to train a reward network $R_\psi(s)$ to produce reward values? For short, we write

$$r_A = R_\psi(s_A), \qquad r_B = R_\psi(s_B).$$

We can relate the reward scores to the person's choice using the *Bradley–Terry model*:

$$p = p_\psi(A \succ B) = \frac{e^{r_A}}{e^{r_A} + e^{r_B}} = \frac{1}{1 + e^{-(r_A - r_B)}}.$$

Here $A \succ B$ means that the person expresses a preference for A over B. The probability increases with the difference in preference scores; it is $1/2$ when the scores are equal.

We'll work with N comparisons, each expressing a strict preference between two states. We store the preferred state first in each pair, giving the data set

$$\mathcal{D} = \{(s_A^{(i)}, s_B^{(i)})\}_{i=1}^N, \qquad s_A^{(i)} \succ s_B^{(i)}.$$

For comparison i, we write

$$\Delta^{(i)} = R_\psi(s_A^{(i)}) - R_\psi(s_B^{(i)}), \qquad p^{(i)} = \frac{1}{1 + e^{-\Delta^{(i)}}}.$$

We treat the choices as conditionally independent given the compared states and ψ, and choose ψ to maximize their likelihood:

$$\hat{\psi} = \arg\max_\psi \prod_{i=1}^N p^{(i)} = \arg\max_\psi \sum_{i=1}^N \log p^{(i)}.$$

By taking the logarithm, we replace the product with a sum without changing the maximizing parameters. Equivalently, we can minimize the average negative log-likelihood,

$$\mathcal{L}(\psi) = -\frac{1}{N} \sum_{i=1}^N \log p^{(i)} = \frac{1}{N} \sum_{i=1}^N \log(1 + e^{-\Delta^{(i)}}).$$

To derive a stochastic gradient update, we'll consider one comparison and drop the index i. Its loss is

$$\ell(\psi) = -\log p = \log(1 + e^{-\Delta}), \qquad \Delta = r_A - r_B.$$

First, we differentiate with respect to the score difference:

$$\begin{aligned} \frac{\partial \ell}{\partial \Delta} &= \frac{-e^{-\Delta}}{1 + e^{-\Delta}} \\ &= \frac{1}{1 + e^{-\Delta}} - 1 = p - 1. \end{aligned}$$

Since $\partial\Delta/\partial r_A = 1$ and $\partial\Delta/\partial r_B = -1$, we get loss derivatives with opposite signs for the two scores:

$$\frac{\partial \ell}{\partial r_A} = p - 1, \qquad \frac{\partial \ell}{\partial r_B} = 1 - p.$$

Both scores depend on the same parameters ψ. We apply the chain rule through both evaluations of the reward network to obtain

$$\begin{aligned} \nabla_\psi \ell &= \frac{\partial \ell}{\partial r_A} \nabla_\psi R_\psi(s_A) + \frac{\partial \ell}{\partial r_B} \nabla_\psi R_\psi(s_B) \\ &= (p - 1)\big[\nabla_\psi R_\psi(s_A) - \nabla_\psi R_\psi(s_B)\big]. \end{aligned}$$

We can compute the two score gradients by backpropagation and take a gradient-descent step with learning rate $\alpha > 0$:

$$\begin{aligned}\psi &\leftarrow \psi - \alpha \nabla_\psi \ell \\ &= \psi + \alpha(1-p)\big[\nabla_\psi R_\psi(s_A) - \nabla_\psi R_\psi(s_B)\big].\end{aligned}$$

All quantities on the right are evaluated before the update. We move in the direction of $\nabla_\psi \Delta$, scaled by the positive factor $\alpha(1-p)$. With a sufficiently small step, we increase the preferred state's score relative to the other state's score, provided that gradient is nonzero. The factor $1-p$ is large when the network assigns low probability to the person's choice, and small when it already predicts that choice confidently. If we sample a comparison uniformly from $\mathcal{D}$, its loss gradient is an unbiased estimate of $\nabla_\psi \mathcal{L}$. For a minibatch, we average the individual loss gradients before updating ψ.[15]

[15] Christiano et al., Deep Reinforcement Learning from Human Preferences.

Inverse reinforcement learning We already studied learning from demonstrations via *behavior cloning*, where we directly learned a policy mapping states to actions. When the demonstrator's embodiment differs from the robot's, or when we want to generalize to very different scenarios, we may be unable to clone the behavior directly. We can instead infer a reward model and combine it with other methods to generate behavior.

In *inverse reinforcement learning* (IRL), we fit a reward that could explain the demonstrations. We generally cannot identify a unique reward from demonstrations alone, but we can use a model of how a person chooses behavior to learn which rewards are plausible.

We can connect IRL to the comparison model above by changing the set of alternatives available to the person. In a comparison, we present two alternatives explicitly. In a demonstration, we interpret the observed behavior as a choice from the feasible behaviors the person could perform. We follow Jeon, Milli, and Dragan's account of both kinds of feedback as reward-rational choices.[16]

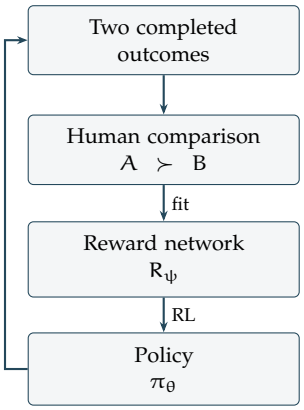


Figure 1: Two learning problems: we train the reward network using human comparisons, then train the policy using the network's terminal scores. We collect further comparisons of new policy outcomes, including outcomes absent from the reward network's training data.

[16] Jeon, Milli, and Dragan (2020), Reward-rational (implicit) choice: A unifying formalism for reward learning, Sections 2–3.

We'll use a per-step reward $R_\psi(s, a)$ and score a trajectory by its total reward,

$$g_\psi(\tau) = \sum_{t=0}^{H-1} R_\psi(s_t, a_t).$$

For a finite set of alternatives $\mathcal{C}$, we model the person's choice using

$$p_\psi(\tau \mid \mathcal{C}) = \frac{\exp[g_\psi(\tau)]}{\sum_{\tau' \in \mathcal{C}} \exp[g_\psi(\tau')]}.$$

With $\mathcal{C} = \{\tau_A, \tau_B\}$, we recover the Bradley–Terry model, now using trajectory returns as the two scores. For a demonstration, we instead use the set $\mathcal{F}$ of all feasible trajectories for the task. Under the model,

a higher-return trajectory is more likely to be demonstrated, but a lower-return trajectory still has positive probability.

For the computation below, we'll assume a finite state space, finite sets of available actions $\mathcal{A}(s)$, known deterministic dynamics $s_{t+1} = f(s_t, a_t)$, a fixed initial state s_{init}, and a fixed horizon H. We abbreviate the demonstration likelihood as

$$p_\psi(\tau) = \frac{\exp[g_\psi(\tau)]}{Z(\psi)}, \qquad Z(\psi) = \sum_{\tau' \in \mathcal{F}} \exp[g_\psi(\tau')].$$

We call $Z(\psi)$ the *partition function*. It is the sum of the unnormalized weights of all alternatives. Dividing by $Z(\psi)$ makes the trajectory probabilities sum to one. We use Ziebart et al.'s maximum-entropy IRL model and partition-function computation, specialized here to a fixed horizon and deterministic dynamics. [17]

[17] Ziebart, Maas, Bagnell, and Dey (2008), Maximum Entropy Inverse Reinforcement Learning, Equations 2 and 6 and Algorithm 1; corrected author-hosted version.

We'll fit the reward to N demonstrated trajectories, $\mathcal{D}_{\text{demo}} = \{\tau^{(i)}\}_{i=1}^{N}$, by minimizing the average negative log-likelihood:

$$\begin{aligned}\mathcal{L}(\psi) &= -\frac{1}{N}\sum_{i=1}^{N} \log p_\psi(\tau^{(i)}) \\ &= -\frac{1}{N}\sum_{i=1}^{N} g_\psi(\tau^{(i)}) + \log Z(\psi).\end{aligned}$$

We need both terms: increasing a demonstration's return also changes the normalizer, and may affect the returns of competing trajectories.

To make the gradient concrete, we'll use a linear reward:

$$R_\psi(s, a) = \psi^\top \phi(s, a), \qquad \Phi(\tau) = \sum_{t=0}^{H-1} \phi(s_t, a_t), \qquad g_\psi(\tau) = \psi^\top \Phi(\tau).$$

The known features $\phi(s, a)$ might measure energy use or proximity to an obstacle; we learn their reward weights ψ. The vector $\Phi(\tau)$ contains the total features along the trajectory. IRL does not require a linear reward, but the linear case makes each term in the update easy to interpret. We write the average feature vector in the demonstrations as

$$\bar{\Phi}_D = \frac{1}{N}\sum_{i=1}^{N} \Phi(\tau^{(i)}).$$

The gradient of the first loss term is $-\bar{\Phi}_D$. For the normalizer, we differentiate the sum of exponentials:

$$\begin{aligned}\nabla_\psi \log Z(\psi) &= \frac{1}{Z(\psi)} \nabla_\psi Z(\psi) \\ &= \frac{1}{Z(\psi)} \sum_{\tau \in \mathcal{F}} e^{\psi^\top \Phi(\tau)} \Phi(\tau) \\ &= \sum_{\tau \in \mathcal{F}} p_\psi(\tau) \Phi(\tau) \\ &= \mathbb{E}_{\tau \sim p_\psi}[\Phi(\tau)].\end{aligned}$$

Combining the two derivatives, we get

$$\nabla_\psi \mathcal{L} = \mathbb{E}_{\tau \sim p_\psi}[\Phi(\tau)] - \bar{\Phi}_D.$$

We can now take a gradient-descent step:

$$\psi \leftarrow \psi - \alpha \nabla_\psi \mathcal{L} = \psi + \alpha\big(\bar{\Phi}_D - \mathbb{E}_{\tau \sim p_\psi}[\Phi(\tau)]\big).$$

If a feature occurs more often in the demonstrations than the model predicts, we increase its reward weight. If it occurs less often, we decrease its weight. For example, with one indicator feature per state, we increase the rewards of states that the model visits too rarely relative to the demonstrations.

6.2.1.1 Detailed derivation

We can skip this derivation on a first reading and continue with the worked example in Section 6.2.1.2.

We can compute $\bar{\Phi}_D$ directly from the data. To compute the model's expected features, we would apparently need to enumerate all feasible trajectories. We can avoid that enumeration by sharing computations among trajectories with the same state and remaining horizon.

We define $Z_t(s)$ to be the sum of exponentiated remaining returns over all feasible continuations from state s at time t. We hold ψ fixed during the computation and omit it from this notation. At time H, there is one empty continuation, with zero remaining reward, so

$$Z_H(s) = e^0 = 1.$$

At an earlier time, we group continuations by their first action. After action a, the next state is $f(s, a)$, and each continuation's weight factors into $e^{R_\psi(s,a)}$ times the weight of its suffix. We therefore define the total weight of continuations beginning with a as

$$Z_t(s, a) = e^{R_\psi(s,a)} Z_{t+1}(f(s, a)).$$

We then add the weights for all available first actions:

$$Z_t(s) = \sum_{a \in \mathcal{A}(s)} Z_t(s, a).$$

Working backward from $t = H-1$ to $t = 0$, we obtain the partition function for the full task, $Z(\psi) = Z_0(s_{\text{init}})$.

We can also recover the probability of the next action under the trajectory model. Among all continuation weights counted in $Z_t(s)$, the portion beginning with action a is $Z_t(s,a)$, so

$$\pi_{\psi,t}(a \mid s) = \frac{Z_t(s,a)}{Z_t(s)} = \frac{e^{R_\psi(s,a)} Z_{t+1}(f(s,a))}{Z_t(s)}.$$

We use this ratio for states with $Z_t(s) > 0$, where at least one feasible continuation exists. Along a feasible trajectory, the successive partition-function factors cancel:

$$\begin{aligned}\prod_{t=0}^{H-1} \pi_{\psi,t}(a_t \mid s_t) &= \exp\left[\sum_{t=0}^{H-1} R_\psi(s_t, a_t)\right] \frac{Z_H(s_H)}{Z_0(s_{\text{init}})} \\ &= \frac{\exp[g_\psi(\tau)]}{Z(\psi)} = p_\psi(\tau).\end{aligned}$$

Thus, if we follow these action probabilities, we sample trajectories from exactly the Boltzmann model used in the likelihood.

We now compute the model's expected feature counts. We write $D_t(s)$ for the probability of being in state s at time t, starting from s_{init} and following $\pi_{\psi,t}$. Initially,

$$D_0(s) = \begin{cases} 1, & s = s_{\text{init}}, \\ 0, & \text{otherwise.} \end{cases}$$

For each state s with positive probability, we distribute its probability among its actions and send each portion to the corresponding next state:

$$D_{t+1}(s') = \sum_{s,a:\, f(s,a)=s'} D_t(s)\pi_{\psi,t}(a \mid s).$$

Here we sum over available actions at states with $D_t(s) > 0$. The probability of visiting (s,a) at time t is $D_t(s)\pi_{\psi,t}(a \mid s)$. Multiplying by its features and summing over time, states, and actions gives

$$\mathbb{E}_{\tau\sim p_\psi}[\Phi(\tau)] = \sum_{t=0}^{H-1} \sum_{s} \sum_{a\in\mathcal{A}(s)} D_t(s)\pi_{\psi,t}(a \mid s)\phi(s,a),$$

where states with $D_t(s) = 0$ contribute zero and can be skipped.

6.2.1.2 Worked example

For a small worked example, we'll use the two-step task in Figure 2. From s_{init}, we can go left to s_L or right to s_R. At s_L, we can pass through a red or a blue gate; at s_R, only a red gate is available. Each gate leads to a terminal state. We use two features, counting red and blue gates

respectively. The first action has feature vector $(0, 0)$; a red-gate action has $(1, 0)$, and a blue-gate action has $(0, 1)$. We'll start with reward weights

$$\psi = \begin{bmatrix} \log 2 \\ 0 \end{bmatrix}.$$

The immediate rewards are therefore 0 for going left or right, $\log 2$ for a red gate, and 0 for a blue gate.

We begin the backward pass with $Z_2(g_j) = 1$ for each terminal state. At time 1, the red and blue action weights are $e^{\log 2} = 2$ and $e^0 = 1$, so

$$Z_1(s_L) = 2 + 1 = 3, \qquad Z_1(s_R) = 2.$$

Since both first actions have zero reward, we get

$$Z_0(s_{\text{init}}, \text{left}) = e^0 \cdot 3 = 3, \qquad Z_0(s_{\text{init}}, \text{right}) = e^0 \cdot 2 = 2,$$

and $Z_0(s_{\text{init}}) = 3 + 2 = 5$. Dividing action weights by state weights gives

$$\begin{aligned} \pi_{\psi,0}(\text{left} \mid s_{\text{init}}) &= \frac{3}{5}, \quad \pi_{\psi,0}(\text{right} \mid s_{\text{init}}) = \frac{2}{5}, \\ \pi_{\psi,1}(\text{red} \mid s_L) &= \frac{2}{3}, \qquad \pi_{\psi,1}(\text{blue} \mid s_L) = \frac{1}{3}, \\ \pi_{\psi,1}(\text{red} \mid s_R) &= 1. \end{aligned}$$

The first actions have equal immediate rewards, but going left is more probable because we can choose between two continuations there.

In the forward pass, $D_1(s_L) = 3/5$ and $D_1(s_R) = 2/5$. We get the three trajectory probabilities by multiplying along each path: $p_\psi(\text{left}, \text{red}) = 2/5$, $p_\psi(\text{left}, \text{blue}) = 1/5$, and $p_\psi(\text{right}, \text{red}) = 2/5$. Each path passes through one gate, so the expected feature counts are

$$\mathbb{E}_{\tau \sim p_\psi}[\Phi(\tau)] = \frac{2}{5}\begin{bmatrix} 1 \\ 0 \end{bmatrix} + \frac{1}{5}\begin{bmatrix} 0 \\ 1 \end{bmatrix} + \frac{2}{5}\begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 4/5 \\ 1/5 \end{bmatrix}.$$

We'll use five demonstrations: two pass through red gates and three through the blue gate. Their average feature vector is $\bar{\Phi}_D = (2/5, 3/5)^\top$. With learning rate $\alpha = 1/2$, one reward update gives

$$\begin{aligned} \psi_{\text{new}} &= \begin{bmatrix} \log 2 \\ 0 \end{bmatrix} + \frac{1}{2}\left(\begin{bmatrix} 2/5 \\ 3/5 \end{bmatrix} - \begin{bmatrix} 4/5 \\ 1/5 \end{bmatrix}\right) \\ &= \begin{bmatrix} \log 2 - 1/5 \\ 1/5 \end{bmatrix}. \end{aligned}$$

We decrease the red-gate reward and increase the blue-gate reward, since the model predicts too many red gates and too few blue gates. After recomputing the partition functions, the probability of the blue-gate trajectory rises from $1/5$ to

$$\frac{e^{1/5}}{2e^{\log 2 - 1/5} + e^{1/5}} \approx 0.272,$$

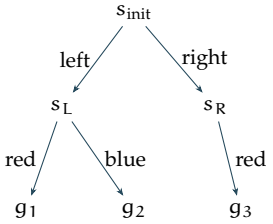


Figure 2: Three feasible trajectories in a two-step task. We count red and blue gates as features; all terminal rewards are zero.

moving toward its demonstrated frequency of 3/5.

We repeat the backward pass, forward pass, and reward update until we finish fitting the model. Each reward update changes the trajectory distribution, so we recompute the expected features using the new parameters. For large problems, we may need approximate inference or sampling. Once we learn a reward, we can use it with planning or RL to generate behavior.

6.2.2 *Transition models*

If we can observe the state, a simple dynamics model takes (s_t, a_t) as input and predicts s_{t+1}. Each recorded transition provides a supervised training example. For continuous state vectors, we can fit a deterministic model f_θ by regression on a dataset $\mathcal{D} = \{(s_i, a_i, s'_i)\}_{i=1}^N$:

$$\mathcal{L}_{\text{reg}}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \|s'_i - f_\theta(s_i, a_i)\|^2.$$

The derivation here assumes deterministic dynamics. With stochastic dynamics, we must distinguish a person's decisions from random outcomes they cannot control. Ziebart, Bagnell, and Dey (2010), Modeling Interaction via the Principle of Maximum Causal Entropy, develop an extension for sequential decisions under uncertainty.

With stochastic dynamics, squared loss fits the conditional mean, which need not itself be a likely outcome.

A probabilistic model $p_\theta(s' \mid s, a) \approx T(s, a, s')$ represents a distribution over possible outcomes. We first choose a distribution family: for discrete states, a categorical distribution; for continuous states, we might use a Gaussian with a learned mean and covariance matrix. Maximum likelihood minimizes

$$\mathcal{L}_{\text{NLL}}(\theta) = -\frac{1}{N} \sum_{i=1}^{N} \log p_\theta(s'_i \mid s_i, a_i).$$

Negative log-likelihood is a *strictly proper scoring rule*: its expected loss is uniquely minimized by the true conditional distribution, when that distribution is available in the model family.

For complicated distributions, including images, conditional diffusion or flow models offer richer alternatives, as we saw in behavior cloning. Here the conditioning variables are (s, a) and the generated sample is s'.

An *encoder-decoder* can instead model the dynamics in a smaller latent space. An encoder E maps an observation o_t to z_t, a transition model predicts z_{t+1}, and a decoder D predicts the next observation:

$$z_t = E(o_t), \qquad z_{t+1} \sim p_\theta(\cdot \mid z_t, a_t), \qquad \hat{o}_{t+1} = D(z_{t+1}).$$

Reconstruction and transition-prediction losses train the representation and dynamics together. The latent state must retain what is needed to predict future observations and rewards; reconstruction alone does not ensure this. Under partial observability, the encoder may need a history of observations and actions, rather than one image.

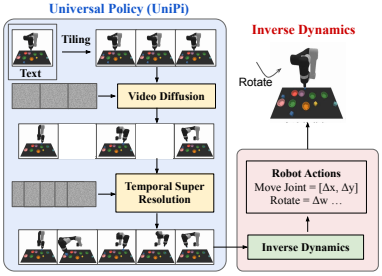


Figure 3: UniPi generates a video plan, then converts frames to robot actions with inverse dynamics. Figure 2 from Du et al. (2023), cropped from the paper; CC BY 4.0.

Video world models can also generate a whole visual plan conditioned on the current image o_0 and a goal description g. An *inverse dynamics* model then predicts actions from successive frames:

$$o_{1:H} \sim p_\theta(\cdot \mid o_0, g), \qquad a_t \sim q_\psi(\cdot \mid o_t, o_{t+1})$$

The inverse model is trained on recorded (o_t, a_t, o_{t+1}) triples; unlabeled video alone does not specify the robot's control commands. UniPi uses this video-generation and inverse-dynamics approach (Figure 3). Generated frames must describe physically achievable transitions: a plausible-looking video need not be an executable plan.

6.3 Planning with learned models

Once we learn a world model, we can use it as a simulator and do RL! We generate transitions with the learned dynamics and evaluate them using the task reward or a learned reward model. The RL algorithms we studied earlier can then train a policy from this simulated experience, either for a fixed reward or conditioned on a goal.

For a differentiable deterministic model and policy, with differentiable rewards, we can also compute policy gradients analytically by backpropagating through simulated trajectories. The computation is the same as in Section 3.1, where we derived policy gradients with a known model; we now use the learned model in its place. We hold the model parameters fixed while updating the policy parameters. In either approach, we move computation into training so that choosing an action at execution time is cheap.

An exact optimal policy specifies optimal actions throughout the state space; an optimal goal-conditioned policy must also cover the goals. Computing these policies can be intractable when the spaces are large. A learned policy can generalize without explicitly solving every state–goal pair, but its training may not cover all possible situations we could encounter.

We might imagine a pre-compiled policy for all the things that could go wrong on an international air trip!

With *inference-time planning*, we instead concentrate computation on the current state s and goal g, using the models to choose the next action. We execute it, observe the outcome, and plan again.

The geometric motion planners we studied earlier in the course exploit a special structure: edges directly connect configurations, with known collision geometry and usually no explicit dynamics. Now we will consider the more general case in which actions may be torques or discrete decisions, and transitions may be stochastic.

In the worst case, choosing even one optimal action can require reasoning about every reachable state and the actions we would take there. Planning from the current state does not remove worst-case complexity.

We'll first describe the planning objective and the loop that connects planning to execution. Then we'll look at methods that search for an action sequence, including kinodynamic RRT, which builds a tree of

partial trajectories. Finally, we'll study Monte Carlo tree search, where future actions can depend on the outcomes of earlier ones.

6.3.1 The planning objective and replanning

In *model predictive control* (MPC), we limit each planning call to a horizon of H steps for computational efficiency. Here H is a choice about how far we can afford to search. It need not be the task horizon introduced in Section 1.2: the task might continue for thousands of steps, or indefinitely, while we plan only ten steps ahead.

We'll initially search for an action sequence $\mathbf{a} = (a_0, \ldots, a_{H-1})$ from the current state estimate s_0. With reward model R and discount factor γ, we score it by

$$J(\mathbf{a}) = \mathbb{E}\left[\sum_{k=0}^{H-1} \gamma^k R(s_k, a_k) + \gamma^H \hat{V}(s_H)\right],$$

The expectation is over model rollouts with transitions sampled from $p_\theta(s' \mid s, a)$. For a deterministic model, we instead use $s_{k+1} = f_\theta(s_k, a_k)$.

The *terminal value estimate* $\hat{V}(s_H)$ estimates the return from continuing *optimally* after the search ends. The state s_H need not be terminal in the actual task. We might learn $\hat{V}$ from experience, or construct it using knowledge of the problem. If the goal lies beyond the search horizon and all intermediate rewards are zero, a useful $\hat{V}$ can distinguish progress toward the goal from moving away. Setting $\hat{V} = 0$ simply ignores rewards beyond the cutoff.

The terminal value plays the role of a search heuristic: we combine the return along the simulated path with an estimate of what remains. In a cost-minimization problem, a heuristic $h(s)$ estimates remaining cost; here $\hat{V}(s)$ estimates remaining reward.

The ideal sequence optimizer returns

$$\mathbf{a}^* = \arg\max_{\mathbf{a} \in \mathcal{A}^H} J(\mathbf{a}),$$

where $\mathcal{A}$ is the allowed action set. In practice, we seek a good sequence within a computation budget. The sequence is *open loop*: its later actions do not depend on which simulated outcomes occur. Even when we use many rollouts to estimate its expected return, we plan to apply the same sequence in every rollout.

We execute only the first action, observe what actually happens, and plan again. The resulting *receding-horizon* loop provides feedback, even when each individual plan is open loop.

Model predictive control

1: inputs: model p_θ, reward R, terminal estimate $\hat{V}$, planning horizon H, discount γ
2: $s \leftarrow$ state estimate from the current observation
3: while the task continues:
4: $\quad \mathbf{a} \leftarrow \text{Plan}(s, p_\theta, R, \hat{V}, H, \gamma)$
5: $\quad a \leftarrow a_0$ ▷ *only the first planned action*
6: $\quad o \leftarrow \text{ExecuteAndObserve}(a)$
7: $\quad s \leftarrow \text{UpdateStateEstimate}(s, a, o)$
8: $\quad$ ▷ *the next iteration plans from the actual outcome*

Here Plan approximately maximizes J, and ExecuteAndObserve acts in the real environment. We can use the unused suffix of the old plan to initialize the next search, appending a new final action.

Replanning serves two purposes. First, we adapt to unintended outcomes caused by disturbances, stochastic transitions, or model errors. Second, we move the computational horizon forward: after one real step, the next search can see one step farther into the future. We can therefore carry out a task much longer than H. A useful terminal estimate can help make up for the short-horizon approximation.

Other approximations may still be needed. We can sample next states from a transition distribution, or use *determinization* to replace it with a single predicted outcome. We can sample candidate actions from a continuous action space. Under partial observability, we also need to represent uncertainty about the current state and reason about future observations. The methods below differ in how they spend the available search budget.

6.3.2 *Approximate path search methods*

Path search methods optimize a single action sequence. With deterministic dynamics, each sequence generates one predicted state path; with stochastic dynamics, we can evaluate the same sequence on several sampled paths. Each algorithm below is a possible implementation of the Plan call in the MPC loop. We hold the current state, models, horizon, discount, and terminal value estimate fixed during that call; the outer loop executes only the first action.

Random shooting samples M candidate action sequences from an arbitrary distribution q over action sequences of length H:

$$\mathbf{a}^{(i)} = (a_0^{(i)}, \ldots, a_{H-1}^{(i)}), \qquad i = 1, \ldots, M.$$

We simulate each candidate and select the one with the largest estimated return. With K rollouts per candidate, we write $s_k^{(i,j)}$ for the state at

step k of rollout j for candidate i. Every rollout starts at $s_0^{(i,j)} = s_0$ and uses the actions $a_k^{(i)}$. The score is

$$\hat{J}(\mathbf{a}^{(i)}) = \frac{1}{K} \sum_{j=1}^{K} \left[\sum_{k=0}^{H-1} \gamma^k R(s_k^{(i,j)}, a_k^{(i)}) + \gamma^H \hat{V}(s_H^{(i,j)}) \right].$$

For a deterministic model, one rollout per candidate suffices.

Random shooting

1: inputs: horizon H, distribution q over $\mathcal{A}^H$, candidate count M, rollout count K
2: for $i = 1, \ldots, M$:
3: $\quad \mathbf{a}^{(i)} \sim q$
4: $\quad \hat{J}(\mathbf{a}^{(i)}) \leftarrow$ average return of K model rollouts
5: return $\displaystyle\arg\max_{\mathbf{a} \in \{\mathbf{a}^{(1)}, \ldots, \mathbf{a}^{(M)}\}} \hat{J}(\mathbf{a})$

We only need a simulator, and we can evaluate candidates in parallel. Random shooting works with discrete or continuous actions, but finding a good sequence by chance becomes difficult as the horizon and action dimension grow.

The *cross-entropy method* (CEM) adapts the distribution q we use to sample possible solutions. We call q a *proposal distribution* because it proposes candidates for evaluation. After each batch, CEM keeps a set of the highest-scoring candidates (called *elites*), fits a new sampling distribution to them, and repeats. For continuous actions, we often use a Gaussian over the whole action sequence:

$$q_{\mu,\Sigma} = \mathcal{N}(\mu, \Sigma), \qquad \mathbf{a}^{(i)} \sim q_{\mu,\Sigma}.$$

Here μ is the mean action sequence and Σ is its covariance. If E indexes the elites, the new mean is

$$\mu_{\text{new}} = \frac{1}{|E|} \sum_{i \in E} \mathbf{a}^{(i)}.$$

We also estimate the covariance from the elites, often retaining only its diagonal. All elites receive equal weight; other samples receive zero weight. Successive batches concentrate sampling near promising sequences. A minimum variance can prevent the proposal distribution from collapsing too soon. [18]

In the sketch below, L is the number of elites, and Score computes $\hat{J}$ using the rollout average above. We write S for all candidates sampled during the search and return the best one. An implementation only needs to retain the best candidate and its score between batches.

[18] Chua et al. (2018), Deep Reinforcement Learning in a Handful of Trials using Probabilistic Dynamics Models, use CEM with learned probabilistic dynamics.

Cross-entropy method

1: inputs: horizon H, initial μ, Σ, batch size M, elite count L
2: repeat for the chosen number of batches:
3: $\quad \mathbf{a}^{(i)} \sim \mathcal{N}(\mu, \Sigma)$ for $i = 1, \ldots, M$
4: $\quad \hat{J}(\mathbf{a}^{(i)}) \leftarrow \text{Score}(\mathbf{a}^{(i)})$ for each i
5: $\quad E \leftarrow$ indices of the L highest-scoring candidates
6: $\quad (\mu, \Sigma) \leftarrow$ mean and covariance of the elites
7: $\quad$ keep the highest-scoring sequence seen so far
8: return $\arg\max_{\mathbf{a} \in \mathcal{S}} \hat{J}(\mathbf{a})$

Model predictive path integral control (MPPI) also adapts the distribution q we use to sample possible solutions. We use the same Gaussian form $q_{\mu,\Sigma} = \mathcal{N}(\mu, \Sigma)$, but keep the covariance Σ fixed. Each batch updates the mean using smoothly varying weights for all candidates, in place of CEM's equal weights for the elites and zero weights for the rest. An importance-sampling form of the update is

$$\begin{aligned} \tilde{w}_i &= \exp\!\left(\hat{J}(\mathbf{a}^{(i)})/\eta\right) \frac{q_0(\mathbf{a}^{(i)})}{q_{\mu,\Sigma}(\mathbf{a}^{(i)})}, \\ w_i &= \frac{\tilde{w}_i}{\sum_{j=1}^{M} \tilde{w}_j}, \qquad \mu_{\text{new}} = \sum_{i=1}^{M} w_i \mathbf{a}^{(i)}. \end{aligned} \tag{4}$$

Here $\eta > 0$ is a temperature and q_0 is a fixed reference distribution over action sequences. The ratio corrects for sampling from the current proposal distribution; implementations often absorb it into the rollout cost. With exact scores, the weighted samples approximate a distribution proportional to $q_0(\mathbf{a}) \exp(J(\mathbf{a})/\eta)$, favoring high return while retaining a preference for the reference controls. Smaller η concentrates weight on better rollouts.[19]

We compute all scores and weights using the same proposal distribution before updating its mean. The covariance Σ stays fixed during these updates, and the final mean is the action sequence we return.

[19] Williams et al., Information Theoretic Model Predictive Control: Theory and Applications to Autonomous Driving.

Model predictive path integral control

1: inputs: horizon H, initial mean $\boldsymbol{\mu}$, fixed covariance Σ, temperature η, reference q_0, batch size M
2: repeat for the chosen number of batches:
3: $\quad \mathbf{a}^{(i)} \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$ for $i = 1, \ldots, M$
4: $\quad \hat{J}(\mathbf{a}^{(i)}) \leftarrow \text{Score}(\mathbf{a}^{(i)})$ for each i
5: $\quad$ compute $\tilde{w}_i$ and w_i using Equation (4)
6: $\quad \boldsymbol{\mu} \leftarrow \sum_{i=1}^{M} w_i \mathbf{a}^{(i)}$
7: return $\boldsymbol{\mu}$

The Gaussian CEM and MPPI updates above require actions in a vector space where averaging actions makes sense. We average the action vectors at corresponding time steps; any constraints on the allowed actions still need to be respected.

Even when averaging actions makes sense, averaging good plans can produce a bad plan. We'll use a point robot with position s_k and displacement action a_k, with dynamics $s_{k+1} = s_k + a_k$. Two four-step sequences move from $(-2, 0)$ to $(2, 0)$, one above a rock and one below it (Figure 4). Both routes avoid the rock, but their equally weighted mean has

$$\bar{a}_k = \tfrac{1}{2}\left(a_k^{(+)} + a_k^{(-)}\right) = (1, 0), \qquad k = 0, \ldots, 3.$$

The mean sequence leads straight through the rock! CEM would center its next proposal distribution on this unsafe mean, although our CEM algorithm still returns the best sampled sequence. Our MPPI algorithm returns the mean itself. Good sampled plans do not guarantee a good average.

Figure 5 compares one batch of random shooting, CEM, and MPPI using the same samples and scores. Random shooting selects one sequence. CEM uses an equally weighted elite set to refit a distribution. MPPI uses smoothly varying weights to update the mean of its proposal distribution, keeping the covariance fixed in the version shown here. The diagram compares the updates; performance also depends on the task, sampling budget, and settings. In all three cases, the MPC loop executes only the first action before planning again.

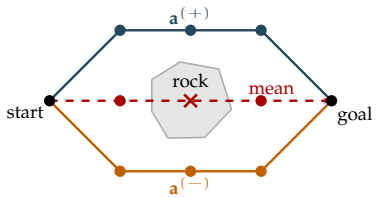


Figure 4: Averaging two routes around a rock. The first sequence uses actions $(1, 1), (1, 0), (1, 0), (1, -1)$; the second reverses the vertical components. Their mean uses $(1, 0)$ at every step and hits the rock. Dots mark the states after each action under $s_{k+1} = s_k + a_k$. Original illustration.

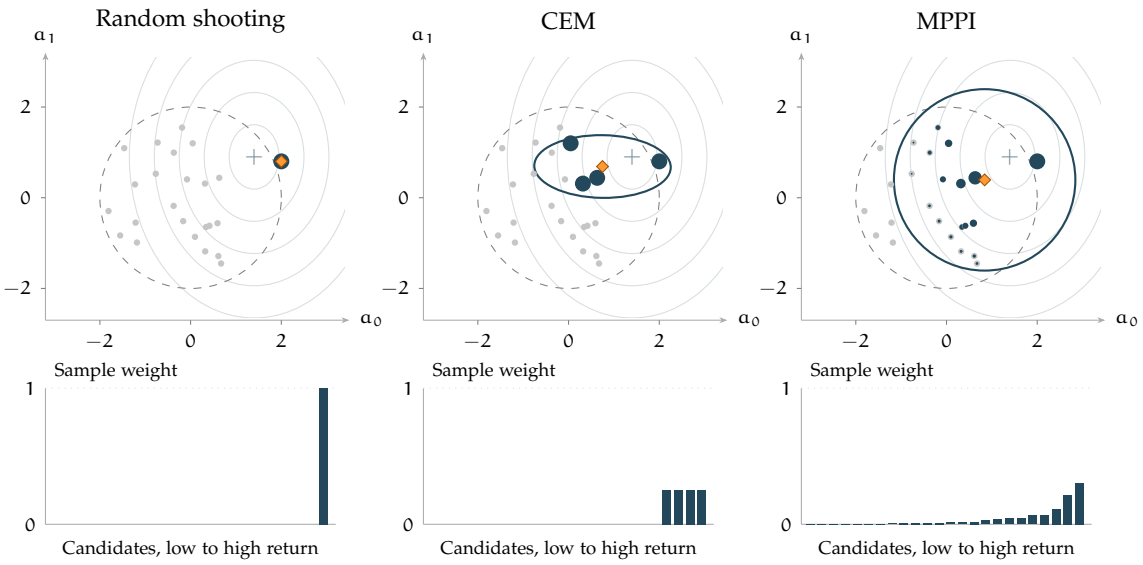


Figure 5: One update using the same 24 sequences $\mathbf{a} = (a_0, a_1)$. Pale contours show return, increasing toward the $+$. Random shooting keeps one sample; CEM uses the best four equally; MPPI weights all samples. Blue point areas indicate relative weights within each panel; bars share a common weight scale. Dashed gray and solid blue ellipses show the initial and updated proposal distributions (two standard deviations). Orange diamonds mark the selected sequence or new mean. For MPPI, $\eta = 1$ and initially $q_0 = q_{\mu,\Sigma}$, so the importance ratio is one. Original illustration with computed scores and updates.

Gradient-based trajectory optimization uses derivatives to improve a continuous action sequence. With a differentiable deterministic model, we can differentiate J through the rollout:

$$\mathbf{a} \leftarrow \mathbf{a} + \alpha \nabla_{\mathbf{a}} J.$$

The dynamics model f_θ must be differentiable with respect to both its state and action inputs, as must the reward and terminal value functions along the rollout. We hold the model parameters fixed and optimize the actions, as in the trajectory optimization we saw earlier. We can enforce action bounds by projecting each update back into the allowed set, denoted by $\Pi_{\mathcal{A}^H}$ below.

Gradient-based trajectory optimization

1: inputs: initial sequence $\mathbf{a}$, step size α
2: repeat for the chosen number of gradient steps:
3: simulate $s_1, \ldots, s_H$ using f_θ and $\mathbf{a}$
4: compute $J(\mathbf{a})$, including the terminal value
5: $g \leftarrow \nabla_{\mathbf{a}} J$ by backpropagation through the rollout
6: $\mathbf{a} \leftarrow \Pi_{\mathcal{A}^H}(\mathbf{a} + \alpha g)$
7: return $\mathbf{a}$

Every gradient step uses a fresh rollout from the same current state s_0, with the updated action sequence. Good initialization matters because the objective is generally nonconvex. Optimization can also

adversarially exploit errors in the learned model, finding actions with excellent predicted return that perform poorly in the real system.

6.3.3 *Kinodynamic RRT*

We can also adapt the rapidly-exploring random trees (RRTs) we studied earlier to plan with dynamics. In *kinodynamic RRT*, a vertex represents a full state, including velocity when it affects future motion. Edges are trajectories generated by applying controls under the dynamics. Two states with the same position but different velocities may require very different braking maneuvers![20]

For a deterministic learned model $s' = f_\theta(s, a)$, we can grow the tree by forward simulation. We sample a target state, find a nearby tree vertex, and try short control sequences from that vertex. One simple version holds each sampled control constant for a sampled number of steps. We keep the valid trajectory whose endpoint lies closest to the target. The target guides exploration; the new edge need not reach it exactly. No model derivatives or procedure for connecting two specified states is required.

Kinodynamic RRT: a simple forward-simulation version

1: inputs: start s_0, goal region $\mathcal{G}$, model f_θ, state distance d, trajectory validity test, search budget
2: initialize a tree T with root s_0
3: repeat until the budget runs out:
4: sample a target state s_{target}, occasionally in $\mathcal{G}$
5: $s_{\text{near}} \leftarrow$ nearest vertex in T under d
6: sample several admissible controls and short durations
7: simulate each candidate from s_{near} using f_θ
8: discard trajectories that violate collision or other constraints
9: if none remain, continue to the next iteration
10: add the endpoint closest to s_{target} and its trajectory to T
11: if the endpoint is in $\mathcal{G}$, return its root-to-goal control sequence
12: report that no path was found within the budget

The tree reuses partial trajectories, and an extension can start from any stored vertex. We still need a useful state distance and a way to check constraints along each predicted trajectory. Basic RRT seeks a feasible path to a goal; it does not optimize our general return objective. Its output is one action sequence, even though the search uses a tree.

[20] LaValle and Kuffner (2001), Randomized Kinodynamic Planning; LaValle (2006), *Planning Algorithms*, Section 14.4.3.

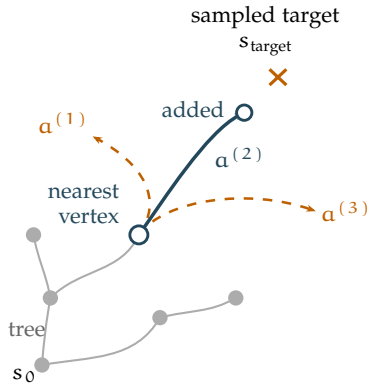


Figure 6: One kinodynamic RRT extension. From the nearest tree vertex, three controls generate candidate rollouts. We add the blue rollout because its endpoint is closest to the sampled target. Gray edges belong to the existing tree.

With stochastic dynamics, storing one sampled outcome per edge would not tell us what to do after other outcomes. We'll now turn to a tree search that explicitly represents those choices.

6.3.4 *Monte Carlo tree search*

When we know a probabilistic transition model (it's enough to be able to sample from $P(s' \mid s, a)$—we don't need to be able to evaluate probabilities or densities), we can consider all (or many) possible outcomes of an action, and select different actions depending on what happens. *Monte Carlo tree search* (MCTS) organizes possible futures into a tree and allocates simulations to promising branches. We can use it for deterministic problems, stochastic problems, and games. In a deterministic problem, an action has one successor state. In a stochastic problem, it may have several possible successors. In a game, the other player's move creates branches that we do not control, even when the rules themselves are deterministic.

In the latter two cases, the tree includes possible responses to outcomes other than the one we expect. Its solution represents a partial *contingent policy*: which action to take at each explored decision state. We ultimately execute one root action, but we evaluate that action using possible later choices throughout the tree. A single path cannot represent these contingencies.

For a minimal example, we'll use a two-step task with $\gamma = 1$. At the start, a safe route ends the task with reward 6. A shortcut has zero initial reward and reveals whether the way ahead is clear (probability 0.99) or blocked (probability 0.01). We observe the result before choosing whether to continue or retreat. Continuing earns 10 if the way is clear and -1000 if it is blocked; retreating earns 0 in either case (Figure 7).

Searching only the likely path makes the shortcut followed by continuing look attractive: it earns 10. But the expected return of committing to those two actions is

$$0.99(10) + 0.01(-1000) = -0.1.$$

A sequence optimizer that correctly averages over both outcomes therefore prefers the safe route, worth 6; committing to retreat after the shortcut would earn only 0. Tree search can evaluate the plan "take the shortcut, then continue if clear and retreat if blocked," whose expected return is

$$0.99 \max\{10, 0\} + 0.01 \max\{-1000, 0\} = 9.9.$$

The difference comes from choosing the second action after seeing the outcome. Replanning an open-loop sequence can react to a blocked route, but the initial sequence optimization does not credit that future ability to react. The tree can account for it when choosing between the

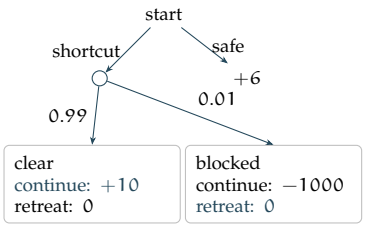


Figure 7: We observe the shortcut's outcome before the second decision. A contingent plan continues when the way is clear and retreats when it is blocked. Blue text marks these choices.

safe route and the shortcut. MCTS still needs enough simulations to discover the rare branch; a finite search can miss it.

MCTS builds such a tree incrementally (Figure 8). At a decision node with state s, one common selection rule is

$$a = \arg\max_a \left[\bar{Q}(s,a) + c\sqrt{\frac{\log N(s)}{N(s,a)}} \right].$$

Here $\bar{Q}(s,a)$ is the average simulated return for that edge, $N(s,a)$ counts its visits, $N(s) = \sum_a N(s,a)$, and $c > 0$ controls exploration. Unvisited actions are tried first. The rule, called *UCT*, trades off high estimated return against limited sampling.[21]

Each simulation selects actions in the existing tree, samples their outcomes from the transition model, and expands a new node. At a chance branch, we sample according to its probability; we do not choose whichever outcome looks best. From a new leaf, a simulated continuation or $\hat{V}$ supplies an estimate of the remaining return. For an MDP, backing up $G \leftarrow R(s,a) + \gamma G$ along the simulated path updates visit counts and average returns. After the search budget is exhausted, we execute a root action, commonly the most visited one, and search again after observing the outcome.

In a two-player zero-sum game, we also account for whose turn it is. At the opponent's decision nodes, selection favors their payoff, equivalently the negative of ours, while still exploring less-visited moves. We do not assume that the opponent will cooperate by choosing the continuation that gives us the highest return.

For continuous actions, *progressive widening* grows the candidate set as a node receives more visits. One rule allows at most

$$m(s) = \max\{1, \lceil k_a N(s)^\beta \rceil\}, \qquad k_a > 0, \quad 0 < \beta < 1$$

action children. Below this limit we propose a new action; otherwise UCT selects among existing actions. Sublinear growth leaves time to evaluate each candidate repeatedly. With continuous stochastic next states, *double progressive widening* also limits the number of outcome children per action, reusing sampled outcomes according to their empirical probabilities.[22] Tree search is most effective when useful branches can be proposed and revisited; raw, high-dimensional motor commands often favor the sequence-optimization methods above.

6.4 Combining learning and planning

A learned policy and value function can make planning more effective; planning can, in turn, provide better training data for both. This creates a loop in which learning improves planning and planning improves learning.

[21] Kocsis and Szepesvári (2006), Bandit Based Monte-Carlo Planning.

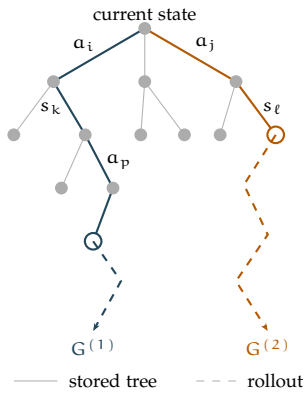


Figure 8: Two MCTS simulations with alternating action and outcome branches. Solid branches belong to the stored tree; each highlighted path reaches a newly expanded leaf (open circle), then continues with a deeper rollout (dashed). The sampled returns are backed up through the visited tree nodes. We need not store every rollout state in the tree.

[22] Couëtoux et al. (2011), Continuous Upper Confidence Trees.

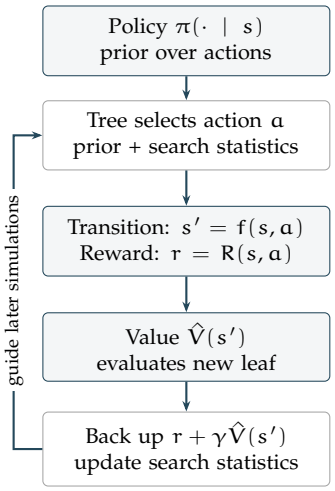


Figure 9: Where MuZero's learned components enter MCTS. The last edge of a simulation is shown: s is a latent search state and s' a new leaf. Rewards and the leaf value are backed up along the selected path; root visit counts determine the real action. Based on Schrittwieser et al., MuZero.

AlphaZero is a famous example. The learned policy provides a prior over moves for MCTS, concentrating search on promising branches. The learned value function evaluates leaf positions without searching all the way to the end of the game. Together they let a limited search budget produce stronger decisions. In the other direction, MCTS produces an improved move distribution from its visit counts, which becomes the policy's training target. Games played using these search decisions produce outcomes that train the value function. Better networks then support stronger search in the next round of self-play.[23] AlphaZero is given the game rules: its transition model is exact.

[23] Silver et al., Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm.

MuZero learns the dynamics as well (figure 9). An encoder maps observation history to a latent state; a learned dynamics network predicts the next latent state and reward given an action. Policy and value predictions at these latent states guide MCTS, just as in AlphaZero. The model is trained to predict observed rewards, return targets, and search move distributions over multiple steps. It need not reconstruct observations: its latent dynamics need to support accurate predictions for planning.

Dreamer uses a learned world model to train an actor and critic on imagined trajectories. *DayDreamer* applies this approach to physical robots for locomotion, manipulation, and navigation (figure 10). Real observations, actions, and rewards enter a replay buffer. An encoder combines observations with the previous latent state and action to estimate the current latent state. A recurrent dynamics model predicts subsequent latent states given actions; a decoder reconstructs observations, and a reward head predicts rewards. Reconstruction, reward prediction, and agreement between inferred and predicted latent states are components of the loss function for learning the world model.

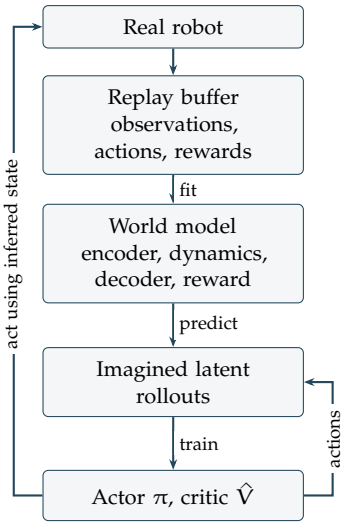


Figure 10: The DayDreamer learning loop. Real experience trains the world model; imagined experience trains behavior. Based on Wu et al., DayDreamer.

Starting from states inferred from replay, the actor chooses actions and the model generates short latent rollouts, without rendering images. Predicted rewards and bootstrapped critic values form return targets: the critic fits these targets, and the actor maximizes them. For continuous actions, actor gradients pass through the learned dynamics; the world-model parameters stay fixed during these behavior updates. On the robot, the encoder updates the latent state and the actor selects an action, with no online tree search. Data collection runs alongside learning, so new real experience continually corrects the model used to train the policy.

7 *Other classes to take!*

6.8200: Sensorimotor Learning Pulkit Agrawal's class goes deeper into learning for control, including reinforcement learning, imitation learning, self-supervised learning, and learned models. Hands-on exercises and research projects emphasize formulating sensorimotor problems and choosing methods that work. This is a natural continuation for building learning-based robot systems.

6.7920: Reinforcement Learning: Foundations and Methods Cathy Wu and Munther Dahleh develop the mathematical foundations of RL, from finite- and infinite-horizon dynamic programming through Monte Carlo methods, TD learning, Q-learning, and policy optimization. The class connects algorithms to stochastic approximation, bandits, and finite-sample analysis, with computational exercises and a project. It provides a more systematic foundation for the methods surveyed here.

6.4110: Representation, Inference, and Reasoning in AI Leslie Pack Kaelbling's class develops representations and algorithms for reasoning and planning: constraint satisfaction, logical inference, probabilistic graphical models, search, and planning with MDPs and POMDPs. It gives a broader view of how an agent can use a model to make decisions, including when the state is only partially observed.

9.522: Statistical Reinforcement Learning and Decision Making Alexander (Sasha) Rakhlin's course studies the statistical and algorithmic foundations of RL. Topics include multi-armed and contextual bandits, exploration versus exploitation, and RL with function approximation. Frequentist and Bayesian viewpoints help explain when efficient learning is possible and how much interaction it requires.

A The PPO version in HW6

HW6 uses PPO with clipping and GAE to improve a cautious pushing controller learned by behavior cloning. The PPO machinery is general, but we deliberately give its actor only one parameter to learn. This makes it easier to see how reward-based learning can improve on imitation. Here are the choices that specialize the algorithm in section 4.1.

A discrete choice and a continuous command An action chooses a block face $f \in \{0, 1, 2, 3\}$ and a stick travel distance d. The BC network supplies face probabilities $p_{BC}(f \mid s)$ and a separate latent distance mean $\mu_{BC,f}(s)$ for each face. We first sample a face, then a Gaussian latent z, and finally map z into the allowed distance range:

$$\begin{aligned} f &\sim p_{BC}(F \mid s), \\ z &\sim \mathcal{N}\big(\mu_{\theta,f}(s), \sigma_f(s)^2\big), \\ d &= d_{max}\,\mathrm{sigmoid}(z), \qquad d_{max} = 0.125 \text{ m}. \end{aligned}$$

The Gaussian is conditional on the selected face; we do not average the four means. Finite latent values give $0 < d < d_{max}$. The physical command is executed by supplied planning and control, so one RL step is a whole approach–push–retract–settle cycle.

HW6 freezes the BC network and adds a *residual*, or correction, to its latent mean. If $e_f(s)$ is the body-axis goal error associated with face f, measured in meters, define

$$\begin{aligned} h(s, f) &= \mathrm{clip}\left(\frac{|e_f(s)| - 0.07 \text{ m}}{0.05 \text{ m}}, 0, 1\right), \\ \mu_{\theta,f}(s) &= \mu_{BC,f}(s) + \theta h(s, f), \\ \sigma_f(s) &= 0.2 + 0.3h(s, f). \end{aligned}$$

Only the signed scalar θ, initialized to zero, and a separate, freshly initialized critic are learned by PPO. Face probabilities, BC weights, and the state-dependent exploration standard deviation stay fixed. Positive θ increases the commanded distance for the same latent noise when $h > 0$; negative θ decreases it. The correction vanishes within 7 cm of the goal along the selected axis and has full strength beyond 12 cm. At $\theta = 0$ the latent means match BC, although the exploration standard deviation is set by the formula above. This is a restricted policy family: it cannot learn a different face selection rule. Section 4.4 discusses residual policies more generally.

Which probability goes into the ratio? For the stored face and latent sample, the joint log probability/density is

$$q_t(\theta) = \log p_{BC}(f_t \mid s_t) + \log p_\theta(z_t \mid s_t, f_t),$$
$$\rho_t(\theta) = \exp\big(q_t(\theta) - q_t(\theta_{\text{old}})\big).$$

We always evaluate the *same recorded* (s_t, f_t, z_t) under the current and old policies; we do not draw a new action to compute the ratio. Adding the two log terms implements the product of the face probability and the conditional Gaussian density. The frozen face probability cancels in this particular residual policy's ratio.

Why can we use the latent Gaussian density when the robot executes d? For $d = d_{\max}\,\text{sigmoid}(z)$, change of variables gives

$$p_\theta(f, d \mid s) = \frac{p_{BC}(f \mid s)\, p_\theta(z \mid s, f)}{d_{\max}\,\text{sigmoid}(z)(1 - \text{sigmoid}(z))} \quad .$$

The Jacobian factor cancels between new and old densities because the transform is fixed and we evaluate both at the same action. It is not equal to one, and would matter for an absolute physical-action log density. Storing z_t also avoids numerically inverting a nearly saturated sigmoid.

Two backward calculations: advantages and critic targets HW6 uses $\gamma = 1$ and $\lambda = 0.95$. Rollouts end on success or physical failure, or stop at a computational cutoff of 32 commands. The cutoff is not a task deadline: the underlying task could continue. The state has no countdown input. Let T be the number of recorded steps in one such episode or episode prefix, and set

$$b_T = \begin{cases} 0, & \text{success or physical failure,} \\ V_{\phi_{\text{old}}}(s_T), & \text{computational cutoff.} \end{cases}$$

Use this b_T for the final next-state value in the TD errors. For earlier steps use the saved old value of the actual next state. Compute GAE backward within this episode:

$$\widehat{\text{adv}}_T = 0,$$
$$\widehat{\text{adv}}_t = \delta_t + 0.95\,\widehat{\text{adv}}_{t+1}.$$

Even at a cutoff the final *advantage accumulator* is zero; the continuation value has already entered through the last TD error. Never carry this recurrence into the next episode after a reset. Concatenate all episodes' estimates, then standardize once over the whole batch as described below.

The critic has a different regression target, which we denote by y_t:

$$y_T = b_T, \qquad y_t = r_t + y_{t+1} = \sum_{j=t}^{T-1} r_j + b_T \quad .$$

This is the observed return to the end of the recorded segment, plus the old critic's estimate of any continuation. At a true terminal it is simply the observed return to go. HW6 does *not* use $\widehat{\text{adv}}_t + V_{\phi_{\text{old}}}(s_t)$ as its critic target with $\lambda = 0.95$. Those expressions agree when $\lambda = 1$, by telescoping, but generally not otherwise.

For example, take rewards $(0, 1)$, old values $(0.2, 0.4)$, and a true terminal after the second reward. The TD errors are $(0.2, 0.6)$, so the first GAE estimate is $0.2 + 0.95(0.6) = 0.77$, while the critic target is $y_0 = 1$. If instead this is a cutoff with continuation $b_T = 0.5$, the last TD error becomes 1.1, the first GAE estimate becomes 1.245, and $y_0 = 1.5$. Neither calculation uses a value from the reset state. Evaluation reports only rewards actually observed; it does not add this training-time continuation estimate.

Optional advantage standardization Some implementations, including HW6, *standardize* the advantage estimates before using them as weights in the policy objective. In HW6 these are the GAE estimates computed above. For n recorded transitions, define

$$\bar{A} = \frac{1}{n}\sum_i \widehat{\text{adv}}_i, \qquad s_A = \sqrt{\frac{1}{n}\sum_i (\widehat{\text{adv}}_i - \bar{A})^2},$$
$$\widetilde{\text{adv}}_i = \frac{\widehat{\text{adv}}_i - \bar{A}}{s_A + 10^{-8}}.$$

Dividing by the standard deviation keeps the weights on a comparable numerical scale across batches; the small constant prevents division by zero. Subtracting the mean removes a common offset that may remain when the critic is imperfect. It also changes the interpretation: a positive weight now means *above this batch's average estimated advantage*, not simply better than the critic predicted. For example, centering estimates $(1, 2, 3)$ gives $(-1, 0, 1)$ before division, even though all three outcomes were better than predicted.

This is an optional optimization convention, not a requirement of PPO. Changing signs can change which branch of the clipped objective applies, so it can change the update; it is not just a change in learning rate. The initial agreement with the return-minus-baseline gradient derived in section 4.1 refers to the unstandardized weights. HW6 computes the standardized weights once over the whole collected batch and holds them fixed through its minibatch updates. Only the actor uses these weights: rewards and critic regression targets stay in their original units.

The loss and the minibatch loop For a minibatch B, let $\widetilde{\mathrm{adv}}_t$ be the already standardized weights. The implementation minimizes

$$\mathcal{L}_{\text{actor}}(\theta) = -\frac{1}{|B|}\sum_{t\in B}\min\big(\rho_t\widetilde{\mathrm{adv}}_t, \mathrm{clip}(\rho_t, 0.9, 1.1)\widetilde{\mathrm{adv}}_t\big),$$
$$\mathcal{L}_{\text{value}}(\phi) = \frac{1}{|B|}\sum_{t\in B}\big(V_\phi(s_t) - y_t\big)^2,$$
$$\mathcal{L}(\theta,\phi) = \mathcal{L}_{\text{actor}}(\theta) + \tfrac{1}{4}\mathcal{L}_{\text{value}}(\phi).$$

The minus sign converts actor maximization to loss minimization. The actual update uses $\epsilon = 0.1$; a hand-calculation question in HW6 separately uses $\epsilon = 0.2$. There is no value-loss clipping, no demonstration loss during PPO, and no entropy bonus. Exploration comes from sampling the frozen face distribution and the Gaussian with its prescribed standard deviation.

The short notebook experiment uses up to four shuffled passes (*epochs*) over its one batch, with minibatches of at most 64 transitions. Adam uses learning rate 0.01 for θ and 0.0003 for the critic. Before each proposed optimizer step, it checks the minibatch estimate

$$\widehat{D}_B = \frac{1}{|B|}\sum_{t\in B}\big(\rho_t - 1 - \log\rho_t\big).$$

For actions sampled from the old policy, its expectation at a fixed state is $D_{KL}(\pi_{\text{old}}\|\pi_\theta)$: the mean of $\rho - 1$ is zero, and the remaining term is the old-to-new log ratio. If $\widehat{D}_B > 0.02$, the loop skips that step and stops all remaining epochs for the batch. This is an additional check on policy movement, not a hard KL constraint or a guarantee of improvement. Otherwise it backpropagates the combined loss, clips the combined actor/critic gradient norm to 0.5, and takes the optimizer step. Gradient-norm clipping limits the gradient magnitude; it is separate from clipping the probability ratio in the actor objective.

One HW6 PPO iteration

1. Collect a batch with the fixed old policy; save states, faces, latent samples, rewards, old log probabilities, old values, and whether each episode ended or was cut off.
2. Compute episode-local GAE and the separate return targets y_t; standardize advantages once over the entire batch.
3. Keep old log probabilities, standardized advantages, and critic targets fixed throughout optimization.

4: For up to four epochs, shuffle and split into minibatches; recompute current log probabilities and current value predictions.
5: Stop if the minibatch KL estimate exceeds 0.02; otherwise minimize the combined loss with one Adam step as described above.
6: For another online iteration, collect fresh data with the updated policy and recompute all saved quantities and targets.

The critic can be updated in the same minibatch loop as the actor because the advantages and targets have already been saved as fixed numbers. Gradients flow through current log probabilities and current value predictions, not through the saved quantities or the simulator.

For the short exercise, collection has already been done: the notebook loads 185 recorded on-policy transitions and the exact checkpoint that generated them, and checks their old log probabilities. Repeating the exercise restarts from that checkpoint; it is not another online iteration. The supplied full training runs do collect fresh rollouts. Neither teacher demonstrations nor arbitrary later instructor batches can simply replace those fresh rollouts after the actor changes. PPO's clipping does not turn arbitrary old data into on-policy data.