CS285 (Deep Reinforcement Learning) Notes [Fall 2020]

Patrick Yin

Updated August 11, 2021

Contents

1 Note **3**
1.1 Note 3

2 Imitation Learning **4**
2.1 Imitation Learning 4
2.2 Goal-Conditioned Behavorial Cloning 5

3 Reinforcement Learning **6**
3.1 Definitions 6
3.2 RL Algorithm Anatomy 7
3.3 Value Functions 7
3.4 Types of Algorithms 8
3.5 Tradeoffs Between Algorithms 9

4 Policy Gradient **11**
4.1 Direct Policy Differentiation 11
4.2 Understanding Policy Gradients 12
4.3 Reducing Variance 13
4.4 Off-Policy Policy Gradients 15
4.5 Covariant/Natural Policy Gradient 16

5 Actor-Critic Algorithms **17**
5.1 Policy Evaluation 17
5.2 Actor-Critic 19
5.3 Actor-Critic Design Decisions 20
5.4 Critics as Baselines 21

6 Value Function Methods **23**
6.1 Policy Iteration 23
6.2 Fitted Value Iteration & Q-Iteration 24
6.3 Q-Learning 25
6.4 Value Functions in Theory 25

7 Deep RL with Q-Functions **27**
7.1 Replay Buffers . . . 27
7.2 Target Networks . . . 27
7.3 Improving Q-Learning . . . 28
7.4 Implementation Tips . . . 30

8 Advanced Policy Gradient **31**
8.1 Policy Gradient as Policy Iteration . . . 31
8.2 Bounding the Distribution Change . . . 32
8.3 Policy Gradients with Constraints . . . 33
8.4 Natural Gradient . . . 34

9 Model-Based Planning **36**
9.1 Optimal Planning and Control . . . 36
9.2 Open-Loop Planning . . . 37
9.3 Trajectory Optimization with Derivatives . . . 39
9.4 LQR for Stochastic and Nonlinear Systems . . . 41

10 Model-Based Reinforcement Learning **44**
10.1 Model-Based RL Basics . . . 44
10.2 Uncertainty in Model-Based RL . . . 45
10.3 Uncertainty-Aware Neural Net Models . . . 45
10.4 Planning With Uncertainty, Examples . . . 46
10.5 Model-Based RL with Images . . . 47

Chapter 1

Note

1.1 Note

These course notes are my notes from CS 285 : Deep Reinforcement Learning taught by Professor Sergey Levine. The course is linked here. I did not formally take this course, but self-studied the material via the lectures posted published on YouTube. These notes are currently in progress.

Chapter 2

Imitation Learning

2.1 Imitation Learning

Imitation learning has the issue of distributional drift. We can solve this in two ways.

The first way is just to mimic the expert so accurately so that it doesn't drift. Failing to fit the expert accurately could be due to non-markovian and/or multimodal behavior. For the former problem, we can consider history with an RNN. For the later problem, we can output a MoG, use latent variable models, or use autoregressive discretization.

The second way is to generate more data so that the training distribution matches the policy trajectory distribution. DAgger, Dataset Aggregation, does this by collecting data from $p_{\pi_\theta}(o_t)$ instead of $p_{data}(o_t)$.

Algorithm 1 DAgger

```
train $\pi_\theta(a_t|o_t)$ from human data $\mathcal{D} = \{o_1, a_1, ..., o_N, a_N\}$
run $\pi_\theta(a_t|o_t)$ to get dataset $\mathcal{D}_\pi = \{o_1, ..., o_M\}$
ask human to label $\mathcal{D}_\pi$ with actions $a_t$
aggregate: $\mathcal{D} \leftarrow \mathcal{D} \bigcup \mathcal{D}_\pi$
```

Let us prove theoretically why the error with DAgger is an order of magnitude lower than that of traditional behaviorial cloning. Define

$$c(s, a) = \begin{cases} 0 & a = \pi^*(s) \\ 1 & otherwise \end{cases}$$

Assuming that $\pi_\theta(a \neq \pi^*(s)|s) \leq \epsilon$ for all s in $\mathcal{D}_{train}$,

$$\mathbb{E}[\sum_t c(s_t, a_t)] \leq \epsilon T + (1 - \epsilon)\epsilon T + (1 - \epsilon)^2 \epsilon T + ... = \mathcal{O}(\epsilon T^2)$$

More generally, let us instead assume that $\pi_\theta(a \neq \pi^*(s)|s) \leq \epsilon$ for all s $\sim p_{train}(s)$. With DAgger, since $p_{train}(s) \to p_\theta(s)$,

$$\mathbb{E}[\sum_t c(s_t, a_t)] \leq \epsilon T$$

With behavorial cloning, we can compute the probability distribution of the a state under the current policy in terms of the probability distribution of training data as such:

$$p_\theta(s_t) = (1-\epsilon)^t p_{train}(s_t) + (1-(1-\epsilon)^t)p_{mistake}(s_t)$$

so,

$$|p_\theta(s_t) - p_{train}(s_t)| = (1-(1-\epsilon)^t)|p_{mistake}(s_t) - p_{train}(s_t)| \leq 2(1-(1-\epsilon)^t) \leq 2\epsilon t$$

Then, we know that

$$\begin{aligned}\sum_t \mathbb{E}_{p_\theta(s_t)}[c_t] = \sum_t \sum_{s_t} p_\theta(t) c_t(s_t) &\leq \sum_t \sum_{s_t} p_{train}(s_t) c_t(s_t) + |p_\theta(s_t) - p_{train}(s_t)| c_{max} \\ &\leq \sum_t \epsilon + 2\epsilon t = \mathcal{O}(\epsilon T^2)\end{aligned}$$

It turns out we can get the same bounds with the looser assumption that $\mathbb{E}_{p_{train}(s)}[\pi_\theta(a \neq \pi^*(s)|s)] \leq \epsilon$, but we won't prove this here.

2.2 Goal-Conditioned Behavorial Cloning

For a policy to reach any goal p, which may not be in the training dataset, we can condition our policy on p. In other words, we collect data and train a goal conditioned policy with the last state being the goal state. In "Learning to Reach Goals via Iterated Supervised Learning", the authors start with a random policy, collect data with random goals, treat this data as demonstrations for the goals that were reached, used this to improve the policy, and repeated.

Chapter 3

Reinforcement Learning

3.1 Definitions

MDP is defined by $\mathcal{M} = \{\mathcal{S}, \mathcal{A}, \mathcal{T}, r\}$, and POMDP is defined by $\mathcal{M} = \{\mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{T}, \mathcal{E}, r\}$ where $\mathcal{E}$ gives the emission probability $p(o_t|s_t)$. Let us define

$$p_\theta(\tau) := p_\theta(s_1, a_1, ..., s_T, a_T) = p(s_1) \prod_{t=1}^{T} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t)$$

Then the optimal RL policy is

$$\theta^* = \arg\max_\theta \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_t r(s_t, a_t) \right]$$

For convenient, let us rewrite this expression in terms of state-action marginals:

$$\theta^* = \arg\max_\theta \sum_{t=1}^{T} \mathbb{E}_{(s_t, a_t) \sim p_\theta(s_t, a_t)} [r(s_t, a_t)]$$

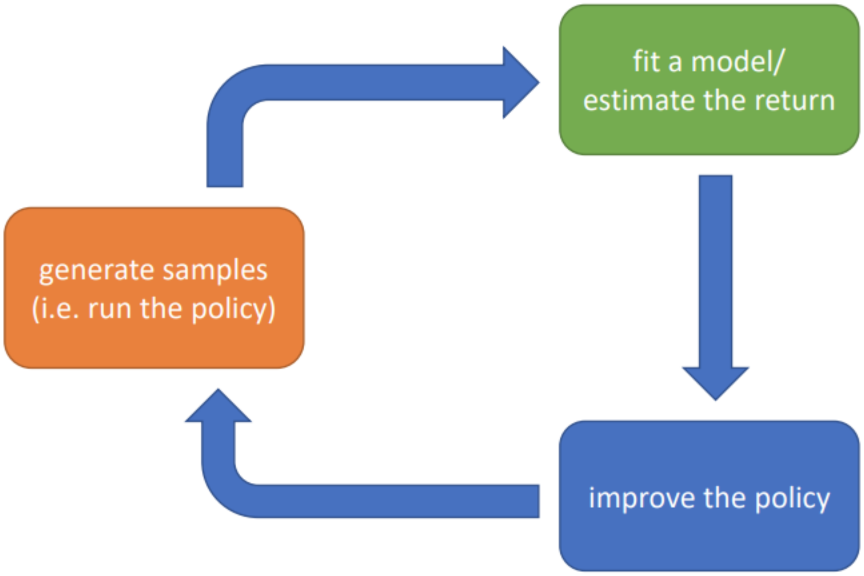
where $p_\theta(s_t, a_t)$ is our state-action marginal. Our "new" MC now has the transition probability $p((s_{t+1}, a_{t+1})|(s_t, a_t)) = p(s_{t+1}|s_t, a_t)\pi_\theta(a_{t+1}|s_{t+1})$.

In the infinite horizon case (i.e. when $T = \infty$), $p(s_t, a_t)$ converges to a stationary distribution if it is ergodic. If so, then the stationary distribution μ satisfies $\mu = \mathcal{T}\mu$, so $\mu := p_\theta(s, a)$ is the eigenvector of $\mathcal{T}$ with eigenvalue 1. If rewards are positive, the reward sum can go to infinity, so we also divide by T. Thus, in the infinite horizon case, our optimal RL policy is

$$\theta^* = \arg\max_\theta \frac{1}{T} \sum_{t=1}^{T} \mathbb{E}_{(s_t, a_t) \sim p_\theta(s_t, a_t)} [r(s_t, a_t)] \to \mathbb{E}_{(s,a) \sim p_\theta(s,a)} [r(s, a)]$$

Also note that we care about expectations over the reward accrued because it is smooth in θ.

3.2 RL Algorithm Anatomy



3.3 Value Functions

Remember that our objective function is

$$\mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\sum_t r(s_t,a_t)\right]$$

This can rewritten as

$$\mathbb{E}_{s_1\sim p(s_1)}\left[\mathbb{E}_{a_1\sim\pi(a_1|s_1)}\left[r(s_1,a_1)+\mathbb{E}_{s_1\sim p(s_2|s_1,a_1)}\left[\mathbb{E}_{a_2\sim\pi(a_2|s_1)}\left[r(s_2,a_2)+...|s_2\right]|s_1,a_1\right]|s_1\right]\right]$$

If we define

$$Q(s_1,a_1)=r(s_1,a_1)+\mathbb{E}_{s_1\sim p(s_2|s_1,a_1)}\left[\mathbb{E}_{a_2\sim\pi(a_2|s_1)}\left[r(s_2,a_2)+...|s_2\right]|s_1,a_1\right]$$

then

$$\mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\sum_t r(s_t,a_t)\right]=\mathbb{E}_{s_1\sim p(s_1)}\left[\mathbb{E}_{a_1\sim\pi(a_1|s_1)}\left[Q(s_1,a_1)|s_1\right]\right]$$

This "Q-function" is very useful as if we know it, we can easily improve the policy. So, we will define a Q-function as such

$$Q^\pi(s_t,a_t):=\sum_{t'=t}^{T}\mathbb{E}_{\pi_\theta}\left[r(s_{t'},a_{t'})|s_t,a_t\right]$$

and a value function as such

$$V^{\pi}(s_t) := \sum_{t'=t}^{T} \mathbb{E}_{\pi_\theta}\big[r(s_{t'}, a_{t'})|s_t\big] = \mathbb{E}_{a_t \sim \pi(a_t|s_t)}\big[Q^{\pi}(s_t, a_t)\big]$$

Note that $\mathbb{E}_{s_1 \sim p(s_1)}\big[V^{\pi}(s_1)\big]$ is the RL objective.
Using Q-functions and value functions, we have two high-level ideas of how we can improve our policy π over time:

1. Set $\pi'(a|s) = 1$ if $a = \arg\max_a Q^{\pi}(s, a)$. This is at least as good as π.

2. If $Q^{\pi}(s, a) > V^{\pi}(s)$, then a is better than average. Modify $\pi(a|s)$ to increase probability of a if $Q^{\pi}(s, a) > V^{\pi}(s)$.

3.4 Types of Algorithms

RL Algorithm Types			
RL Type	Description	Fit model and/or estimate return	Improve policy
Policy gradient	directly differentiate the above objective	evaluate returns $R_\tau = \sum_t r(s_t, a_t)$	$\theta \leftarrow \theta + \alpha \nabla_\theta \mathbb{E}[\sum_t r(s_t, a_t)]$
Value-based	estimate value function or Q-function of the optimal policy (no explicit policy)	fit $V(s)$ or $Q(s, a)$	set $\pi(s) = \arg\max_a Q(s, a)$
Actor-critic	estimate value function or Q-function of the current policy, use it to improve policy	fit $V(s)$ or $Q(s, a)$	$\theta \leftarrow \theta + \alpha \nabla_\theta \mathbb{E}[Q(s_t, a_t)]$
Model-based	estimate transition model and use it for planning, to improve a policy, etc.	learn $p(s_{t+1}|s_t)$	A few ways: 1. Just use the model to plan (no policy), such as trajectory optimization/optimal control (e.g. backprop in continuous space or Monte Carlo tree search in discrete space) 2. Backprop gradients into policy 3. Use model to learn a value function with DP or by generating simulated experience for model-free learner

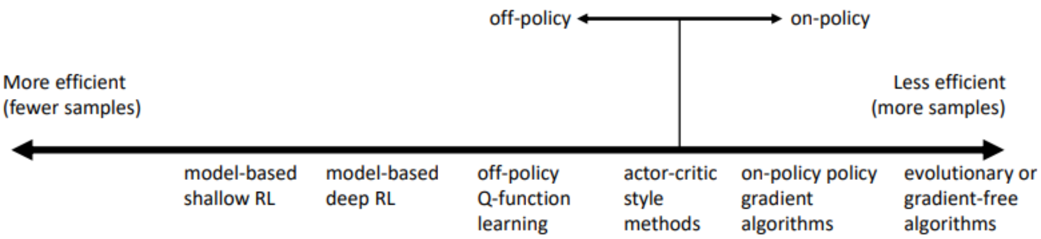
3.5 Tradeoffs Between Algorithms

There are many RL algorithms because there are

- Different tradeoffs: sample efficiency, stability and ease of use
- Different assumptions: stochastic or deterministic, continuous or discrete, episodic or infinite horizon
- DIfferent things are easy or hard in different settings: easier to represent the policy, easier to represent the model

3.5.1 Sample Efficiency

Sample efficiency spectrum:


off-policy
on-policy
More efficient
(fewer samples)
Less efficient
(more samples)
model-based shallow RL
model-based deep RL
off-policy Q-function learning
actor-critic style methods
on-policy policy gradient algorithms
evolutionary or gradient-free algorithms

Off-policy means being able to improve the policy without generating new samples from the policy. On-policy means that each time the policy is changed, even a little bit, we need to generate new samples. Note that wall clock time is not the same as sample efficiency.

3.5.2 Stability and Ease of Use

Unlike supervised learning, RL often does not use gradient descent so there are not as much convergence guarantees:

- Value function fitting: At best, Bellman error (i.e. error of fit) is minimized. At worst, nothing is optimized (many not guaranteed to converge to anything in nonlinear case)
- Model-based RL: Model minimizes error of fit. But no guarantee better model = better policy.
- Policy gradient: The only one that actually performs gradient descent (or ascent) on the true objective.

3.5.3 Common Assumptions

1. Full observability
    (a) Generally assumed by value function fitting methods
    (b) Can be mitigated by adding recurrence
2. Episodic learning
    (a) Often assumed by pure policy gradient methods
    (b) Assumed by some model-based RL methods
3. Continuity/Smoothness
    (a) Assumed by some continuous value function learning methods
    (b) Often assumed by some model-based RL methods

Chapter 4

Policy Gradient

4.1 Direct Policy Differentiation

Remember that in RL

$$\theta^* = \arg\max_\theta \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_t r(s_t, a_t) \right]$$

For simplicity, define

$$J(\theta) := \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_t r(s_t, a_t) \right] \approx \frac{1}{N} \sum_i \sum_t r(s_{i,t}, a_{i,t})$$

and define

$$r(\tau) = \sum_{t=1}^{T} r(s_t, a_t)$$

so, then

$$J(\theta) := \mathbb{E}_{\tau \sim p_\theta(\tau)}[r(\tau)] = \int p_\theta(\tau) r(\tau) d\tau$$

A convenient identity we will use quite a bit for the future is that

$$p_\theta(\tau) \nabla_\theta \log p_\theta(\tau) = p_\theta(\tau) \frac{\nabla_\theta p_\theta(\tau)}{p_\theta(\tau)} = \nabla_\theta p_\theta(\tau)$$

Using this identity, we find that

$$\nabla_\theta J(\theta) = \int \nabla_\theta p_\theta(\tau) r(\tau) d\tau = \int p_\theta(\tau) \nabla_\theta \log p_\theta(\tau) r(\tau) d\tau = \mathbb{E}_{\tau \sim p_\theta(\tau)}[\nabla_\theta \log p_\theta(\tau) r(\tau)]$$

We know that

$$\begin{aligned} p_\theta(\tau) &= p(s_1)\prod_{t=1}^{T}\pi_\theta(a_t|s_t)p(s_{t+1}|s_t,a_t) \\ \log p_\theta(\tau) &= \log p(s_1) + \sum_{t=1}^{T}\log\pi_\theta(a_t|s_t) + \log p(s_{t+1}|s_t,a_t) \\ \nabla_\theta \log p_\theta(\tau) &= \sum_{t=1}^{T}\nabla_\theta\log\pi_\theta(a_t|s_t) \end{aligned}$$

Now, we have a nicer expression for $\nabla_\theta J(\theta)$:

$$\begin{aligned} \nabla_\theta J(\theta) &= \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\left(\sum_{t=1}^{T}\nabla_\theta\log\pi_\theta(a_t|s_t)\right)\left(\sum_{t=1}^{T}r(s_t,a_t)\right)\right] \\ &\approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta\log\pi_\theta(a_{i,t}|s_{i,t})\right)\left(\sum_{t=1}^{T}r(s_{i,t},a_{i,t})\right) \end{aligned}$$

Now if we simply do gradient ascent using this gradient, we have our most basic policy gradient algorithm REINFORCE:

Algorithm 2 REINFORCE

```
loop
    sample $\{\tau^i\}$ from $\pi_\theta(a_t|s_t)$ (run the policy)
    $\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta\log\pi_\theta(a_{i,t}|s_{i,t})\right)\left(\sum_{t=1}^{T}r(s_{i,t},a_{i,t})\right)$
    $\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta)$
end loop
```

4.2 Understanding Policy Gradients

In policy gradients, the gradient is

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta\log\pi_\theta(a_{i,t}|s_{i,t})\right)\left(\sum_{t=1}^{T}r(s_{i,t},a_{i,t})\right)$$

In MLE, the gradient is

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta\log\pi_\theta(a_{i,t}|s_{i,t})\right)$$

Intuitively, REINFORCE is then just trial and error learning where good stuff is made more likely and bad stuff is made less likely.

Also note that if we follow the same derivation with partial observability, we get

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}|o_{i,t}) \right) \left(\sum_{t=1}^{T} r(s_{i,t}, a_{i,t}) \right)$$

so we can use policy gradient on POMDPs without modification.

4.3 Reducing Variance

The current policy gradient won't work in practice because the variance of the gradient is too big. We can implement some tricks to reduce the variance of the gradient while keeping the estimate unbiased.

4.3.1 Causality

Through some derivation, we can show that

$$\mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \left(\sum_{t'=1}^{t} r(s'_t, a'_t) \right) \right] = 0$$

Intuitively, this makes sense since the policy at time t' cannot affect reward at time t when $t < t'$. So,

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) \left(\sum_{t'=t}^{T} r(s_{i,t'}, a_{i,t'}) \right)$$

is the same as our original approximation in expectation and now has smaller variance. We will define the right-hand expression as our "reward to go" $\hat{Q}_{i,t} = \sum_{t'=t}^{T} r(s_{i,t'}, a_{i,t'})$.

4.3.2 Baselines

Average Reward

It turns out that

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta \log p_\theta(\tau)[r(\tau) - b]$$

is also a valid approximation where

$$b = \frac{1}{N} \sum_{i=1}^{N} r(\tau)$$

This is because

$$\begin{aligned}
\mathbb{E}[\nabla_\theta \log p_\theta(\tau)b] &= \int p_\theta(\tau)\nabla_\theta \log p_\theta(\tau) b d\tau \\
&= \int \nabla_\theta p_\theta(\tau) b d\tau \\
&= b\nabla_\theta \int p_\theta(\tau) d\tau \\
&= b\nabla_\theta 1 = 0
\end{aligned}$$

In other words, subtracting a baseline is also unbiased in expectation. Note that average reward is not the best baseline, but it's still pretty good.

Optimal Baseline

The best baseline minimizes the variance of the gradient, which is

$$\text{Var} = \mathbb{E}_{\tau\sim p_\theta(\tau)}[(\nabla_\theta \log p_\theta(\tau)(r(\tau)-b))^2] - \mathbb{E}_{\tau\sim p_\theta(\tau)}[\nabla_\theta \log p_\theta(\tau)(r(\tau)-b)]^2$$

For convenience, let

$$g(\tau) := \nabla_\theta \log p_\theta(\tau)$$

To minimize this variance, we take a derivative of the variance and set it to zero. Note that the latter part of the variance expression is just $\mathbb{E}_{\tau\sim p_\theta(\tau)}[\nabla_\theta \log p_\theta(\tau) r(\tau)]$ because baselines are unbiased in expectation, so its derivative in terms of b is just zero.

$$\begin{aligned}
\frac{\partial \text{Var}}{\partial b} &= \frac{\partial}{\partial b}\mathbb{E}[g(\tau)^2(r(\tau)-b)^2] \\
&= \frac{\partial}{\partial b}(\mathbb{E}[g(\tau)^2 r(\tau)^2] - 2\mathbb{E}[g(\tau)^2 r(\tau) b] + b^2\mathbb{E}[g(\tau)^2]) \\
&= -2\mathbb{E}[g(\tau)^2 r(\tau)] + 2b\mathbb{E}[g(\tau)^2] = 0 \\
b &= \frac{\mathbb{E}[g(\tau)^2 r(\tau)]}{\mathbb{E}[g(\tau)^2]}
\end{aligned}$$

So the optimal baseline is just the expected reward, but weighted by gradient magnitudes.

4.4 Off-Policy Policy Gradients

In off-policy policy gradients, we want to improve our policy with data generated from a different policy. We can do this with importance sampling:

$$\begin{aligned}\mathbb{E}_{x\sim p(x)}[f(x)] &= \int p(x)f(x)dx \\ &= \int \frac{q(x)}{q(x)}p(x)f(x)dx \\ &= \int q(x)\frac{p(x)}{q(x)}f(x)dx \\ &= \mathbb{E}_{x\sim q(x)}\left[\frac{p(x)}{q(x)}f(x)\right]\end{aligned}$$

In our case, we want to improve policy θ' with data generated from policy θ. So,

$$\begin{aligned}\nabla_{\theta'}J(\theta') &= \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\frac{p_{\theta'}(\tau)}{p_\theta(\tau)}\nabla_{\theta'}\log\pi_{\theta'}(\tau)r(\tau)\right] \\ &= \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\left(\prod_{t=1}^{T}\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)}\right)\left(\sum_{t=1}^{T}\nabla_{\theta'}\log\pi_{\theta'}(a_t|s_t)\right)\left(\sum_{t=1}^{T}r(s_t,a_t)\right)\right] \\ &= \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\sum_{t=1}^{T}\nabla_{\theta'}\log\pi_{\theta'}(a_t|s_t)\left(\prod_{t'=1}^{t}\frac{\pi_{\theta'}(a_{t'}|s_{t'})}{\pi_\theta(a_{t'}|s_{t'})}\right)\left(\sum_{t'=t}^{T}r(s_{t'},a_{t'})\left(\prod_{t''=t}^{t'}\frac{\pi_{\theta'}(a_{t''}|s_{t''})}{\pi_\theta(a_{t''}|s_{t''})}\right)\right)\right]\end{aligned}$$

If we ignore the importance weight ratios over t'', we get a policy iteration algorithm (more on this later), so we can simply the gradient to

$$\nabla_{\theta'}J(\theta') = \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\sum_{t=1}^{T}\nabla_{\theta'}\log\pi_{\theta'}(a_t|s_t)\left(\prod_{t'=1}^{t}\frac{\pi_{\theta'}(a_{t'}|s_{t'})}{\pi_\theta(a_{t'}|s_{t'})}\right)\left(\sum_{t'=t}^{T}r(s_{t'},a_{t'})\right)\right]$$

Note that the importance weight ratios are exponential in T, so its variance will grow exponentially in T. Instead, let us write the objective in terms of state-action marginals, so

$$\begin{aligned}\nabla_{\theta'}J(\theta') &\approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\frac{\pi_{\theta'}(s_{i,t},a_{i,t})}{\pi_\theta(s_{i,t},a_{i,t})}\nabla_{\theta'}\log\pi_{\theta'}(a_{i,t}|s_{i,t})\hat{Q}_{i,t} \\ &\approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\frac{\pi_{\theta'}(s_{i,t})}{\pi_\theta(s_{i,t})}\frac{\pi_{\theta'}(a_{i,t}|s_{i,t})}{\pi_\theta(a_{i,t}|s_{i,t})}\nabla_{\theta'}\log\pi_{\theta'}(a_{i,t}|s_{i,t})\hat{Q}_{i,t}\end{aligned}$$

If we ignore the ratio of the state priors, we have

$$\nabla_{\theta'}J(\theta') \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\frac{\pi_{\theta'}(a_{i,t}|s_{i,t})}{\pi_\theta(a_{i,t}|s_{i,t})}\nabla_{\theta'}\log\pi_{\theta'}(a_{i,t}|s_{i,t})\hat{Q}_{i,t}$$

This is no longer unbiased, but its error can be bounded in terms of the difference between θ and θ' (more on this later). Under this approximation, we get rid of the exponentially growing variance.

4.5 Covariant/Natural Policy Gradient

One issue with policy gradients is that most likely than not the gradients dominate in terms of some parameters, leading it to be hard to converge to optimal parameters. So instead, we need to rescale the gradient so that this doesn't happen.

Recall that with vanilla gradient ascent, we have

$$\theta_{k+1} \leftarrow \theta_k + \alpha \nabla_{\theta_k} J(\theta_k)$$

This is equivalent to

$$\theta_{k+1} \leftarrow \arg\max_{\theta} \alpha \nabla_{\theta_k} J(\theta_k)^T \theta - \frac{1}{2}\|\theta - \theta_k\|^2$$

This makes sense since the above expression is just maximizing the linear approximation of $J(\theta)$ at θ_k with a quadratic regularization. Using Lagrangian form, we can rewrite the above expression as

$$\theta' \leftarrow \arg\max_{\theta'} (\theta' - \theta)^T \nabla_\theta J(\theta) \text{ s.t. } \|\theta' - \theta\| \leq \epsilon$$

Currently the constraint is in parameter-space. As a result, it doesn't account for the fact that some parameters influence the policy more than others. So we would like a constraint in policy-space. One parameterization-independent divergence measure is KL-divergence: $\mathcal{D}_{KL}(\pi_{\theta'}\|\pi_\theta) = \mathbb{E}_{\pi_{\theta'}}[\log \pi_\theta - \log \pi_{\theta'}]$.

$$\theta' \leftarrow \arg\max_{\theta'} (\theta' - \theta)^T \nabla_\theta J(\theta) \text{ s.t. } \mathcal{D}_{KL}(\pi_{\theta'}\|\pi_\theta) \leq \epsilon$$

We can approximate the KL-divergence with its second-order Taylor approximation around $\theta' = \theta$,

$$\mathcal{D}_{KL}(\pi_{\theta'}\|\pi_\theta) \approx (\theta' - \theta)^T \mathbf{F} (\theta' - \theta)$$

$\mathbf{F}$ is the Fischer-information matrix, where

$$\mathbf{F} = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(\mathbf{a}|\mathbf{s}) \nabla_\theta \log \pi_\theta(\mathbf{a}|\mathbf{s})^{\mathbf{T}}]$$

which can be estimated with samples taken form π_θ.

After writing out the Lagrangian and solving for the optimal solution, we find that the update rule is

$$\theta \leftarrow \theta + \alpha \mathbf{F^{-1}} \nabla_\theta \mathbf{J}(\theta)$$

The natural policy gradient selects α. Trust region policy optimization (TRPO) selects ϵ and then derives α using conjugate gradient.

Chapter 5

Actor-Critic Algorithms

5.1 Policy Evaluation

Recall that in policy gradient, we have

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})\hat{Q}_{i,t}^{\pi}$$

$\hat{Q}_{i,t}$ is an estimate of expected reward if we take action $a_{i,t}$ in state $s_{i,t}$. So far, we use a single-sample estimate with $\hat{Q}_{i,t} = \sum_{t'=t}^{T} r(s_{i,t'}, a_{i,t'})$. If we can instead approximate the true expected reward-to-go, $Q(s_t, a_t) = \sum_{t'=t}^{T} \mathbb{E}_{\pi_\theta}[r(s_{t'}, a_{t'})|s_t, a_t]$, then we will have a lower variance estimate. So can could instead approximate the Q-function and calculate the gradient as

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})Q(s_{i,t}, a_{i,t})$$

We can also add in our baseline to reduce variance, so

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})(Q(s_{i,t}, a_{i,t}) - b)$$

One idea for the baseline is to average Q-values: $b_t = \frac{1}{N}\sum_i Q(s_{i,t}, a_{i,t})$. We can reduce variance even more by averaging over actions for a specific state; this is exactly the value function: $V(s_t) = \mathbb{E}_{a_t \sim \pi_\theta(a_t|s_t)}[Q(s_t, a_t)]$. Our gradient then becomes

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})(Q(s_{i,t}, a_{i,t}) - V(s_{i,t}))$$

In terms of notation, we denote the Q-function, or the total reward from taking a_t in s_t, as

$$Q^\pi(s_t, a_t) = \sum_{t'=t}^{T} \mathbb{E}_{\pi_\theta}[r(s_{t'}, a_{t'})|s_t, a_t]$$

We denote value function, or the total reward from s_t, as

$$V^\pi(s_t) = \mathbb{E}_{a_t \sim \pi_\theta(a_t|s_t)}[Q^\pi(s_t, a_t)]$$

We define the advantage function, or how much better a_t is as

$$A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t)$$

With this notation we can write our gradient as

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) A^\pi(s_{i,t}, a_{i,t})$$

Now the question is whether we fit Q^π and/or V^π and/or A^π. We know that

$$\begin{aligned}
Q^\pi(s_t, a_t) &= r(s_t, a_t) + \sum_{t'=t+1}^{T} \mathbb{E}_{\pi_\theta}[r(s_{t'}, a_{t'})|s_t, a_t] \\
&= r(s_t, a_t) + \mathbb{E}_{s_{t+1} \sim p(s_{t+1}|s_t, a_t)}[V^\pi(s_{t+1})] \\
&\approx r(s_t, a_t) + V^\pi(s_{t+1})
\end{aligned}$$

In the last line, we approximate the Q-function with a single-sample estimate from the transition dynamic $p(s_{t+1}|s_t, a_t)$ at the cost of increasing the variance of our Q-value approximation. With this approximation, we now have

$$A^\pi(s_t, a_t) \approx r(s_t, a_t) + V^\pi(s_{t+1}) - V^\pi(s_t)$$

Thus, we can just fit the value function and use it to calculate the advantage.

With Monte Carlo policy evaluation, we can estimate the value function as

$$V^\pi(s_t) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t'=t}^{T} r(s_{t'}, a_{t'})$$

However, we can't do this in most model-free settings since we can't reset the simulator at any state. So instead, we will create a function approximation of the value function with a network. This isn't as good as Monte Carlo, but still pretty good. In a function approximation setting, our training data for the value network will be $\{(s_{i,t}, y_{i,t})\}$ where $y_{i,t} = \sum_{t'=t}^{T} r(s_{i,t'}, a_{i,t'})$. We then train supervised regression with the loss $\mathcal{L}(\phi) = \frac{1}{2} \sum_i \|\hat{V}_\phi^\pi(s_i) - y_i\|^2$. By using a network, similar states will map to similar actions.

We can further reduce the variance with bootstrapping. Recall that our ideal target is

$$y_{i,t} = \sum_{t'=t}^{T} \mathbb{E}_{\pi_\theta}[r(s_{t'}, a_{t'})|s_{i,t}] \approx r(s_{i,t}, a_{i,t}) + V^\pi(s_{i,t+1}) \approx r(s_{i,t}, a_{i,t}) + \hat{V}^\pi_\phi(s_{i,t+1})$$

In the first approximation, we increase the variance of our estimate by using a single-sample estimate of the expectation under $p(s_{t+1}|s_t, a_t)$ (as we did before). In the second approximation, we increase the variance of our estimate by using bootstrapped estimate, where we directly use our previous fitted value function. With bootstrapping, our training data now is $\{(s_{i,t}, y_{i,t})\}$ where $y_{i,t} = r(s_{i,t}, a_{i,t}) + \hat{V}^\pi_\phi(s_{i,t+1})$, and we train supervised regression on the value network with loss function $\mathcal{L}(\phi) = \frac{1}{2}\sum_i \|\hat{V}^\pi_\phi(s_i) - y_i\|^2$.

5.2 Actor-Critic

Algorithm 3 Batch Actor-Critic Algorithm (Without Discounts)

```
loop
    sample $\{s_i, a_i\}$ from $\pi_\theta(a|s)$
    fit $\hat{V}^\pi_\phi(s)$ to sampled reward sums
    evaluate $\hat{A}^\pi(s_i, a_i) = r(s_i, a_i) + \hat{V}^\pi_\phi(s'_i) - \hat{V}^\pi_\phi(s_i)$
    $\nabla_\theta J(\theta) \approx \sum_i \nabla_\theta \log \pi_\theta(a_i|s_i)\hat{A}^\pi(s_i, a_i)$
    $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$
end loop
```

We can fit $\hat{V}^\pi_\phi(s)$ to sampled reward sums with the loss $\mathcal{L}(\phi) = \frac{1}{2}\sum_i \|\hat{V}^\pi_\phi(s_i) - y_i\|^2$ where $y_{i,t} = r(s_{i,t}, a_{i,t}) + \hat{V}^\pi_\phi(s_{i,t+1})$.

If the episode length, T, is ∞, $\hat{V}^\pi_\phi$ can get infinitely large in many cases. So we will add a discount factor $\gamma \in [0, 1]$ (0.99 works well). Adding γ changes the MDP. There is a new state we can call the death state and at each state, there is a $1 - \gamma$ probability of transitioning to the death state. The original transition dynamics now get multiplied by a factor of γ (i.e. $\gamma p(s'|s, a)$). Our new target becomes $y_{i,t} = r(s_{i,t}, a_{i,t}) + \gamma \hat{V}^\pi_\phi(s_{i,t+1})$. With actor-critic, our gradient is

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})(r(s_{i,t}, a_{i,t}) + \gamma \hat{V}^\pi_\phi(s_{i,t+1}) - \hat{V}^\pi_\phi(s_{i,t}))$$

With Monte Carlo policy gradients, we have two options:

$$\begin{aligned}
\text{Option 1: } \nabla_\theta J(\theta) &\approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})(\sum_{t'=t}^{T}\gamma^{t'-t} r(s_{i,t'}, a_{i,t'})) \\
\text{Option 2: } \nabla_\theta J(\theta) &\approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})(\sum_{t'=1}^{T}\gamma^{t'-1} r(s_{i,t'}, a_{i,t'})) \\
&\approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})(\sum_{t'=t}^{T}\gamma^{t'-1} r(s_{i,t'}, a_{i,t'})) \\
&\approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\gamma^{t-1}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})(\sum_{t'=t}^{T}\gamma^{t'-t} r(s_{i,t'}, a_{i,t'}))
\end{aligned}$$

We use option 1 over option 2 because we want a policy that does well at every timestep, not just earlier timesteps.

Algorithm 4 Batch Actor-Critic Algorithm

```
loop
    sample $\{s_i, a_i\}$ from $\pi_\theta(a|s)$
    fit $\hat{V}^\pi_\phi(s)$ to sampled reward sums
    evaluate $\hat{A}^\pi(s_i, a_i) = r(s_i, a_i) + \gamma \hat{V}^\pi_\phi(s'_i) - \hat{V}^\pi_\phi(s_i)$
    $\nabla_\theta J(\theta) \approx \sum_i \nabla_\theta \log \pi_\theta(a_i|s_i) \hat{A}^\pi(s_i, a_i)$
    $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$
end loop
```

We can also create a fully online version of actor-critic.

Algorithm 5 Online Actor-Critic Algorithm

```
loop
    take action $a \sim \pi_\theta(a|s)$, get $(s, a, s', r)$
    update $\hat{V}^\pi_\phi$ using target $r + \gamma \hat{V}^\pi_\phi(s')$
    evaluate $\hat{A}^\pi(s, a) = r(s, a) + \gamma \hat{V}^\pi_\phi(s') - \hat{V}^\pi_\phi(s)$
    $\nabla_\theta J(\theta) \approx \nabla_\theta \log \pi_\theta(a|s) \hat{A}^\pi(s, a)$
    $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$
end loop
```

5.3 Actor-Critic Design Decisions

We can either use a separate network for both $\hat{V}^\pi_\phi(s)$ and $\pi_\theta(a|s)$ or just a shared network design. Single-sample backpropagation updates in online actor-critic is typically not stable due to the high variance of policy gradients. We could instead take 8-16 steps in the environment and update our network on that batch. However, these data points are highly correlated. Instead, we can use parallel workers. In synchronized parallel actor-critic, each worker is initialized in a separate seed, and, at each timestep, all the workers take a step in the environment and return (s, a, s', r), which are batched up and used to update θ. In asynchronous parallel actor-critic, the workers collect data at whatever rate they are able to, and once they collect a transition, they send it to a central parameter server. The parameter server can then make an update and send the updated parameters back to the workers. In the meantime, the workers aren't waiting but are collecting more samples. One might note that workers might be taking steps with an older version of the policy even though the parameter server might have a newer version that hasn't deployed yet. This tends to be ok because the old and new parameters are similar enough.

5.4 Critics as Baselines

5.4.1 Critics as State-Dependent Baselines

In actor-critic,

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})(r(s_{i,t}, a_{i,t}) + \gamma \hat{V}_\phi^\pi(s_{i,t+1}) - \hat{V}_\phi^\pi(s_{i,t}))$$

This has lower variance (due to critic), but is not unbiased if critic is not perfect. On the other hand, in policy gradient,

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})((\sum_{t'=t}^{T}\gamma^{t'-t} r(s_{i,t}, a_{i,t})) - b$$

This has no bias, but has higher variance because it is a single-sample estimate. One way to balance this tradeoff is to use

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})((\sum_{t'=t}^{T}\gamma^{t'-t} r(s_{i,t}, a_{i,t})) - \hat{V}_\phi^\pi(s_{i,t}))$$

This has no bias and has lower variance than the policy gradient approach since the baseline is closer to rewards.

5.4.2 Control Variates: Action-Dependent Baselines

Currently, our advantage is

$$\hat{A}^\pi(s_t, a_t) = \sum_{t'=t}^{\infty}\gamma^{t'-t} r(s_t, a_t) - \hat{V}_\phi^\pi(s_t)$$

This has no bias but has higher variance due to it being a single-sample estimate. Instead, if we use

$$\hat{A}^{\pi}(s_t, a_t) = \sum_{t'=t}^{\infty} \gamma^{t'-t} r(s_t, a_t) - \hat{Q}_{\phi}^{\pi}(s_t, a_t)$$

This goes to zero in expectation if the critic is correct, but is not exactly correct. To be unbiased, we need to add an error term

$$\nabla_{\theta} J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \left(\nabla_{\theta} \log \pi_{\theta}(a_{i,t}|s_{i,t})(\hat{Q}_{i,t} - Q_{\phi}^{\pi}(s_{i,t}, a_{i,t})) + \nabla_{\theta} \mathbb{E}_{a \sim \pi_{\theta}(a_t|s_t)}[Q_{\phi}^{\pi}(s_{i,t}, a_t)] \right)$$

Provided that the second term can be evaluated (which it can in this case), we have an unbiased estimate.

5.4.3 Eligibility Traces & N-Step Returns

A bootstrapped estimate has lower variance but higher bias if the estimate is wrong (it always is). A Monte Carlo estimate has no bias but has higher variance because it is a single-sample estimate. We can control the bias/variance tradeoff by switching from Monte Carlo to bootstrapping at some timestep $n > 1$.

$$\hat{A}_n^{\pi}(s_t, a_t) = \sum_{t'=t}^{t+n} \gamma^{t'-t} r(s_{t'}, a_{t'}) - \hat{V}_{\phi}^{\pi}(s_t) + \gamma^n \hat{V}_{\phi}^{\pi}(s_{t+n})$$

Later steps along Monte Carlo has bigger variance, so we using bootstrapping for later on.

5.4.4 Generalized Advantage Estimation

GAE is a weighted combination of n-step returns.

$$\hat{A}_{GAE}^{\pi}(s_t, a_t) = \sum_{n=1}^{\infty} w_n \hat{A}_n^{\pi}(s_t, a_t)$$

Since we prefer cutting earlier since there is less variance early, we can use an exponential falloff. Define $\delta_{t'} = r(s_{t'}, a_{t'}) + \gamma \hat{V}_{\phi}^{\pi}(s_{t'+1}) - \hat{V}_{\phi}^{\pi}(s_{t'})$.

$$\begin{aligned}
\hat{A}_{GAE}^{\pi}(s_t, a_t) &= (1-\lambda)(\hat{A}_1^{\pi}(s_t, a_t) + \lambda \hat{A}_2^{\pi}(s_t, a_t) + \lambda^2 \hat{A}_3^{\pi}(s_t, a_t) + ...) \\
&= (1-\lambda)(\delta_t(1 + \lambda + \lambda^2 + ...) + \gamma \delta_{t+1}(\lambda + \lambda^2 + ...) + ...) \\
&= \sum_{t'=t}^{\infty} (\gamma\lambda)^{t'-t} \delta_{t'}
\end{aligned}$$

This has a very similar effect to discounts, also implying that discounts have a role in the bias-variance tradeoff.

Chapter 6

Value Function Methods

6.1 Policy Iteration

Instead of using policy gradient to train a policy, we could define

$$\pi'(a_t|s_t) = \begin{cases} 1 & a_t = \arg\max_{a_t} A^\pi(s_t, a_t) \\ 0 & \textit{otherwise} \end{cases}$$

Since $A^\pi(s,a) = r(s,a) + \gamma\mathbb{E}[V^\pi(s')] - V^\pi(s)$, we can just evaluate $V^\pi(s)$. For now, assume we know $p(s'|s,a)$, and s and a are both discrete (and small). Since our policy is deterministic, our bootstrapped update is

$$V^\pi(s) \leftarrow r(s, \pi(s)) + \gamma\mathbb{E}_{s'\sim p(s'|s,\pi(s))}[V^\pi(s')]$$

With our policy and policy evaluation procedure defined, we now have a policy iteration algorithm using dynamic programming:

Algorithm 6 Policy Iteration

```
loop
    Evaluate $V^\pi(s)$ with bootstrapped estimate
    set $\pi \leftarrow \pi'$ where $\pi'$ is defined up above
end loop
```

Note that $\arg\max_{a_t} A^\pi(s_t, a_t) = \arg\max_{a_t} Q^\pi(s_t, a_t)$. Since $\arg\max_a Q(s,a)$ is our policy, we can compute Q-values instead of a policy.

Algorithm 7 Value Iteration

```
loop
    set $Q(s,a) \leftarrow r(s,a) + \gamma\mathbb{E}[V(s')]$
    set $V(s) \leftarrow \max_a Q(s,a)$
end loop
```

6.2 Fitted Value Iteration & Q-Iteration

Representing states in a big table for DP becomes unrealistic for larger state spaces such as images. Instead, we should do regression on the value function

$$\mathcal{L}(\phi) = \frac{1}{2}\|V_\phi(s) - \max_a Q^\pi(s,a)\|^2$$

With this loss, we can do value iteration with a regression value network.

Algorithm 8 Fitted Value Iteration

```
loop
    set $y_i \leftarrow \max_{a_i}(r(s_i, a_i) + \gamma\mathbb{E}[V_\phi(s'_i)])$
    set $\phi \leftarrow \arg\min_\phi \frac{1}{2}\sum_i \|V_\phi(s_i) - y_i\|^2$
end loop
```

However, in order to do this, we need to know outcomes for different actions. Instead, let us iterate on the Q-function instead of the value function. For policy evaluation, instead of

$$V^\pi(s) \leftarrow r(s, \pi(s)) + \gamma\mathbb{E}_{s'\sim p(s'|s,\pi(s))}V^\pi(s')$$

we iterate on the Q-values

$$Q^\pi(s,a) \leftarrow r(s,a) + \gamma\mathbb{E}_{s'\sim p(s'|s,a)}[Q^\pi(s', \pi(s'))]$$

This way, we don't have to take a max over a_i when computing target values.

Algorithm 9 Fitted Q-Iteration

```
loop
    set $y_i \leftarrow r(s_i, a_i) + \gamma\mathbb{E}[V_\phi(s'_i)]$
    $\phi \leftarrow \arg\min_\phi \frac{1}{2}\sum_i \|Q_\phi(s_i, a_i) - y_i\|^2$
end loop
```

We can now write out the full fitted Q-iteration algorithm:

Algorithm 10 Full Fitted Q-Iteration

```
loop
    collect dataset $\{(s_i, a_i, s'_i, r_i)\}$ using some policy
    for K iterations do
        set $y_i \leftarrow r(s_i, a_i) + \gamma \max_{a'_i} Q_\phi(s'_i, a'_i)$
        $\phi \leftarrow \arg\min_\phi \frac{1}{2}\sum_i \|Q_\phi(s_i, a_i) - y_i\|^2$
    end for
end loop
```

This works for off-policy samples, uses only one network, and has no high-variance policy gradient. However, there are no convergence guarantees for non-linear function approximation (more on this later). Now, we have a Q-network with takes in state and action and outputs a number. In the discrete action case as we have right now, we can also structure the network to just take in state and output a head for each action.

6.3 Q-Learning

Fitted Q-iteration is off-policy because given s and a, the transition is independent of π. The bellman error

$$\mathcal{E} = \frac{1}{2}\mathbb{E}_{(s,a)\sim\beta}\left[\left(Q_\phi(s,a) - [r(s,a) + \gamma \max_{a'} Q_\phi(s',a')]\right)^2\right]$$

is approximated by $\sum_i \|Q_\phi(s_i,a_i) - y_i\|^2$ at each step of fitted Q-iteration. If $\mathcal{E} = 0$, then $Q_\phi(r,a) = r(s,a) + \gamma \max_{a'} Q_\phi(s',a')$. This is an optimal Q-function, corresponding to the optimal policy π'. However, most guarantees are lost when we leave the tabular case.

We can convert fitted Q-iteration into an online analogue, which we call Q-learning.

Algorithm 11 Online Q-Iteration

```
loop
    take some action $a_t$ and observe $(s_i, a_i, s'_i, r_i)$
    $y_i = r(s_i, a_i) + \gamma \max_{a'} Q_\phi(s'_i, a'_i)$
    $\phi \leftarrow \phi - \alpha \frac{\partial Q_\phi}{\partial \phi}(s_i, a_i)(Q_\phi(s_i, a_i) - y_i)$
end loop
```

We call $(Q_\phi(s_i,a_i) - y_i)$ the temporal difference (TD) error. In Q-learning, if we explore just based on just the argmax policy, we may be stuck in a subset of the environment and miss out on states and actions that give larger rewards. Some exploration policies are epsilon-greedy

$$\pi(a_t|s_t) = \begin{cases} 1-\epsilon & a_t = \arg\max_{a_t} Q_\phi(s_t,a_t) \\ \epsilon/(|\mathcal{A}|-1) & otherwise \end{cases}$$

and Boltzmann exploration

$$\pi(a_t|s_t) \propto \exp\left(Q_\phi(s_t,a_t)\right)$$

We'll discuss exploration in detail later.

6.4 Value Functions in Theory

We will look at convergence guarantees for these algorithms.

6.4.1 Tabular Value Iteration

Tabular value iteration can be concisely described with the bellman backup operator $\mathcal{B} : \mathcal{B}V = \max_{\mathbf{a}} r_{\mathbf{a}} + \gamma \mathcal{T}_{\mathbf{a}} V$ where $\mathcal{T}_{\mathbf{a},i,j} = p(\mathbf{s}' = i | \mathbf{s} = j, \mathbf{a})$, $r_{\mathbf{a}}$ is a stacked vector of rewards for all states for action $\mathbf{a}$, and we are doing an element-wise max. Note that V^* is a fixed point of $\mathcal{B}$ because $V^* = \mathcal{B}V^*$, so it is unique and always corresponds to the optimal policy. Value iteration reaches V^* because $\mathcal{B}$ is a contraction: for any V and $\bar{V}$, we have $\|\mathcal{B}V - \mathcal{B}\bar{V}\|_\infty \le \gamma \|V - \bar{V}\|_\infty$ (not proved here), so $\|\mathcal{B}V - V^*\|_\infty \le \gamma \|V - V^*\|_\infty$. Here, we have shown that we converge on the optimal policy with tabular value iteration.

6.4.2 Non-Tabular Value Iteration

Let's introduce a new operator $\Pi : \Pi V = \arg\min_{V' \in \Omega} \frac{1}{2} \sum \|V'(s) - V(s)\|^2$, which is a projection onto ω in terms of ℓ_2 norm. Fitted value iteration can be described as $V \leftarrow \Pi \mathcal{B} V$. $\mathcal{B}$ is a contraction w.r.t ∞-norm and Π is a contraction w.r.t ℓ_2-norm, but $\Pi\mathcal{B}$ is not a contraction of any kind, so it does not converge in general and often not in practice.

6.4.3 Fitted Q-Iteration

Similarly, if we define $\Pi : \Pi Q = \arg\min_{Q' \in \Omega} \frac{1}{2} \sum \|Q'(s,a) - Q(s,a)\|^2$, we can describe fitted Q-iteration as $Q \leftarrow \Pi \mathcal{B} Q$. Again, we see that $\Pi\mathcal{B}$ is not a contraction of any kind, so it does not converge in general and often not in practice. This also applies to online Q-learning. Note that Q-learning is not gradient descent because there is no gradient through the target value.

6.4.4 Actor-Critic

In actor-critic, we also have $\mathcal{B}$ without the max and Π in fitting the value function, so fitted bootstrapped policy evaluation also does not converge for the same reason.

Chapter 7

Deep RL with Q-Functions

7.1 Replay Buffers

In online Q-learning, sequential states are strongly correlated and the target value is always changing. Since sequential states are strongly correlated, it is possible for our Q-network to overfit to different chunks along a training trajectory. We could use synchronized parallel Q-learning or asynchronous parallel Q-learning as we did with actor-critic. Another solution is to use replay buffers, which stores a dataset of the agent's most recent trajectories (old trajectories are thrown away when the replay buffer hits a threshold limit).

Algorithm 12 Full Q-Learning with Replay Buffer

```
while some stop condition is not satisfied do
    collect dataset $\{(s_i, a_i, s'_i, r_i)\}$ using some policy, add it to $\mathcal{B}$
    for K iterations do
        sample a batch $(s_i, a_i, s'_i, r_i)$ from $\mathcal{B}$
        $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi}{d\phi}(s_i, a_i)(Q_\phi(s_i, a_i) - [r(s_i, a_i) + \gamma \max_{a'} Q_\phi(s'_i, a'_i)])$
    end for
end while
```

7.2 Target Networks

We have one more issue in that our Q network changes every gradient step, so our target changes every gradient step. As a result, it is possible that this type of "gradient descent" won't converge, since our network is sort of "chasing its own tail". The solution to this is to save an old version of the model for gradient descent and take multiple gradient steps before updating a newer version of the model for gradient descent. We call this old version of the model to be used in the loss function for gradient descent the target network.

Algorithm 13 Q-Learning with Replay Buffer and Target Network

```
while some stop condition is not satisfied do
  save target network parameters: $\phi' \leftarrow \phi$
  for N iterations do
    collect dataset $\{(s_i, a_i, s'_i, r_i)\}$ using some policy, add it to $\mathcal{B}$
    for K iterations do
      sample a batch $(s_i, a_i, s'_i, r_i)$ from $\mathcal{B}$
      $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi}{d\phi}(s_i, a_i)(Q_\phi(s_i, a_i) - [r(s_i, a_i) + \gamma \max_{a'} Q_{\phi'}(s'_i, a'_i)])$
    end for
  end for
end while
```

The classic DQN is essentially Q-Learning with Replay Buffer and Target Network with $K = 1$:

Algorithm 14 Classic Deep Q-Learning (DQN)

```
1: while some stop condition is not satisfied do
2:   take some action $a_i$ and observe $(s_i, a_i, s'_i, r_i)$, add it to $\mathcal{B}$
3:   sample mini-batch $(s_j, a_j, s'_j, r_j)$ from $\mathcal{B}$ uniformly
4:   compute $y_j = r_j + \gamma \max_{a'_j} Q_{\phi'}(s'_j, a'_j)$ using target network $Q_{\phi'}$
5:   $\phi \leftarrow \phi - \alpha \sum_j \frac{dQ_\phi}{d\phi}(s_j, a_j)(Q_\phi(s_j, a_j) - y_j)$
6:   update $\phi'$: copy $\phi$ every $N$ steps
7: end while
```

A popular alternative target network is using Polyak averaging, where in line 6 of DQN we instead set $\phi' : \phi' \leftarrow \tau\phi' + (1-\tau)\phi$. $\tau = 0.999$ works well in practice. The intuition here is to avoid the maximal lag that takes place for update $N - 1 \pmod{N}$ compared to to update $0 \pmod{N}$ where there is no lag.

7.3 Improving Q-Learning

7.3.1 Double Q-Learning

Recall that in Q-learning, our target value is $y_j = r_j + \gamma \max_{a'_j} Q_{\phi'}(s'_j, a'_j)$. However, $Q_{\phi'}(s'_j, a'_j)$ is noisy and thus $\max_{a'_j} Q_{\phi'}(s'_j, a'_j)$ overestimates the next value because for two random variables X_1 and X_2, $\mathbb{E}[\max(X_1, X_2)] \geq \max(\mathbb{E}[X_1], \mathbb{E}[X_2])$. Note that $\max_{a'} Q_{\phi'}(s', a') = Q_{\phi'}(s', \arg\max_{a'} Q_{\phi'}(s', a'))$. If the value calculated and the best action selected were decorrelated, then this problem goes away. This is where double Q-learning comes in, which involves two networks:

$$Q_{\phi_A}(s, a) \leftarrow r + \gamma Q_{\phi_B}(s', \arg\max_{a'} Q_{\phi_A}(s', a'))$$

$$Q_{\phi_B}(s, a) \leftarrow r + \gamma Q_{\phi_A}(s', \arg\max_{a'} Q_{\phi_B}(s', a'))$$

By using a different function approximator for selecting the best action and calculating the value, it is unlikely that the action will be overestimated. In practice, we can use our current network to find the best action and our target network to evaluate its value.

7.3.2 Multi-Step Returns

Again, recall that in Q-learning, our target value is

$$y_{j,t} = r_{j,t} + \gamma \max_{a_{j,t+1}} Q_{\phi'}(s_{j,t+1}, a_{j,t+1})$$

Like in actor-critic, we can instead use N-step return estimators:

$$y_{j,t} = \sum_{t'=t}^{t+N-1} \gamma^{t-t'} r_{j,t'} + \gamma^N \max_{a_{j,t+N}} Q_{\phi'}(s_{j,t+N}, a_{j,t+N})$$

This estimator has less biased target values when Q-values are inaccurate (i.e. in the beginning of training) and typically learn faster early on. However, it is only actually correct when learning on-policy because we need transitions $s_{j,t'}, a_{j,t'}, s_{j,t'+1}$ to come from π for $t' - t < N - 1$ when $N > 1$. To fix this, we could ignore the problem (often works well in practice), cut the trace by dynamically choosing N to get only on-policy data (works well when data is mostly on-policy and action space is small), or do importance sampling.

7.3.3 Q-Learning with Continuous Actions

For continuous actions, we have trouble finding $\max_{a_{j,t+1}} Q_{\phi'}(s_{j,t+1}, a_{j,t+1})$. We have three options:

1. Optimization: we could do gradient based optimization such as SGD. However, this is a bit slow in the inner loop. Instead, we can do stochastic optimization. A simple and parallelizable solution is to sample actions from some distribution (e.g. uniform) and take the max over sampled actions. This is not very accurate however. More accurate solutions are cross-entropy method (CEM), a simple iterative stochastic optimization, and CMA-ES, which is less simple.

2. Maximizable Q-functions: we could use a function class that is easy to optimize such as Normalized Advantage Functions (NAF):

$$Q_\phi(s,a) = -\frac{1}{2}(a - \mu_\phi(s))^T P_\phi(s)(a - \mu_\phi(s)) + V_\phi(s)$$

   where $\arg\max_a Q_\phi(s,a) = \mu_\phi(s)$ and $\max_a Q_\phi(s,a) = V_\phi(s)$. This efficient option requires no change to the algorithm, but representation power is lost.

3. Approximate Maximizer: we could train another network such that $\mu_\theta(s) \approx \arg\max_a Q_\phi(s,a)$ with backpropagation: $\frac{dQ_\phi}{d\theta} = \frac{da}{d\theta}\frac{dQ_\phi}{da}$. Then our new target becomes $y_j = r_j + \gamma Q_{\phi'}(s'_j, \mu_\theta(s'_j))$. The Deep Deterministic Policy Gradient (DDPG) algorithm is as follows

Algorithm 15 DDPG

```
while some stop condition is not satisfied do
    collect dataset $\{(s_i, a_i, s'_i, r_i)\}$ using some policy, add it to $\mathcal{B}$
    sample mini-batch $(s_i, a_i, s'_i, r_i)$ from $\mathcal{B}$ uniformly
    compute $y_j = r_j + \gamma Q_{\phi'}(s'_j, \mu_{\theta'}(s'_j))$ using target nets $Q_{\phi'}$ and $\mu_{\theta'}$
    $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi}{d\phi}(s_i, a_i)(Q_\phi(s_i, a_i) - y_i)$
    $\theta \leftarrow \theta + \beta \sum_j \frac{d\mu}{d\theta}(s_j)\frac{dQ_\phi}{da}(s_j, \mu(s_j))$
    update $\phi'$ and $\theta'$ (e.g. Polyak averaging)
end while
```

7.4 Implementation Tips

- Q-learning takes some care to stabilize, so test on easy, reliable tasks first to make sure your implementation is correct
- Large replay buffers help improve stability
- It takes time
- Start with high exploration (epsilon) and gradually reduce
- Bellman error gradients can be big; clip gradients or user Huber loss
- Double Q-learning helps a lot in practice and has no downsides
- N-step returns help a lot, but have some downsides
- Schedule exploration and learning rates high to low, Adam optimizer can help too
- Run multiple random seeds, it's very inconsistent between runs

Chapter 8

Advanced Policy Gradient

8.1 Policy Gradient as Policy Iteration

Remember that in policy gradient, we want to maximize

$$J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)}\left[\sum_t \gamma^t r(s_t, a_t)\right]$$

We want to find

$$\begin{aligned}
J(\theta') - J(\theta) &= J(\theta') - \mathbb{E}_{s_0 \sim p(s_0)}[V^{\pi_\theta}(s_0)] \\
&= J(\theta') - \mathbb{E}_{\tau \sim p_{\theta'}(\tau)}[V^{\pi_\theta}(s_0)] \\
&= J(\theta') - \mathbb{E}_{\tau \sim p_{\theta'}(\tau)}\left[\sum_{t=0}^{\infty} \gamma^t V^{\pi_\theta}(s_t) - \sum_{t=1}^{\infty} \gamma^t V^{\pi_\theta}(s_t)\right] \\
&= J(\theta') + \mathbb{E}_{\tau \sim p_{\theta'}(\tau)}\left[\sum_{t=0}^{\infty} \gamma^t (\gamma V^{\pi_\theta}(s_{t+1}) - V^{\pi_\theta}(s_t))\right] \\
&= \mathbb{E}_{\tau \sim p_{\theta'}(\tau)}\left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t)\right] + \mathbb{E}_{\tau \sim p_{\theta'}(\tau)}\left[\sum_{t=0}^{\infty} \gamma^t (\gamma V^{\pi_\theta}(s_{t+1}) - V^{\pi_\theta}(s_t))\right] \\
&= \mathbb{E}_{\tau \sim p_{\theta'}(\tau)}\left[\sum_{t=0}^{\infty} \gamma^t (r(s_t, a_t) + \gamma V^{\pi_\theta}(s_{t+1}) - V^{\pi_\theta}(s_t))\right] \\
&= \mathbb{E}_{\tau \sim p_{\theta'}(\tau)}\left[\sum_{t=0}^{\infty} \gamma^t A^{\pi_\theta}(s_t, a_t)\right] \\
&= \sum_{t=0}^{\infty} \mathbb{E}_{s_t \sim p_{\theta'}(s_t)}\left[\mathbb{E}_{a_t \sim \pi_{\theta'}(a_t|s_t)}\left[\gamma^t A^{\pi_\theta}(s_t, a_t)\right]\right] \\
&= \sum_{t=0}^{\infty} \mathbb{E}_{s_t \sim p_{\theta'}(s_t)}\left[\mathbb{E}_{a_t \sim \pi_\theta(a_t|s_t)}\left[\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)} \gamma^t A^{\pi_\theta}(s_t, a_t)\right]\right] \text{ (by importance sampling)}
\end{aligned}$$

Can we instead use $p_\theta(s_t)$? In other words, we want this follow statement to be true:

$$\begin{aligned}
&\sum_{t=0}^{\infty} \mathbb{E}_{s_t \sim p_{\theta'}(s_t)} \left[\mathbb{E}_{a_t \sim \pi_\theta(a_t|s_t)} \left[\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)} \gamma^t A^{\pi_\theta}(s_t, a_t) \right] \right] \\
\approx &\sum_{t=0}^{\infty} \mathbb{E}_{s_t \sim p_\theta(s_t)} \left[\mathbb{E}_{a_t \sim \pi_\theta(a_t|s_t)} \left[\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)} \gamma^t A^{\pi_\theta}(s_t, a_t) \right] \right] = \bar{A}(\theta')
\end{aligned}$$

If this is true, then we can simply update with $\theta' \leftarrow \arg\max_{\theta'} \bar{A}(\theta')$ and use $\hat{A}^\pi(s_t, a_t)$ to get improved policy π'. We claim that $p_\theta(s_t)$ is close to $p_{\theta'}(s_t)$ when π_θ is close to $\pi_{\theta'}$, which we will prove.

8.2 Bounding the Distribution Change

We claim that $p_\theta(s_t)$ is close to $p_{\theta'}(s_t)$ when π_θ is close to $\pi_{\theta'}$. In the simple case, assume that π_θ is a deterministic policy $a_t = \pi_\theta(s_t)$. By definition, $\pi_{\theta'}$ is close to π_θ if $\pi_{\theta'}(a_t \neq \pi_\theta(s_t)|s_t) \leq \epsilon$. Then

$$\begin{aligned}
p_{\theta'}(s_t) &= (1-\epsilon)^t p_\theta(s_t) + (1-(1-\epsilon)^t) p_{\text{mistake}}(s_t) \\
|p_{\theta'}(s_t) - p_\theta(s_t)| &= (1-(1-\epsilon)^t)|p_{\text{mistake}}(s_t) - p_\theta(s_t)| \\
&\leq 2(1-(1-\epsilon)^t) \\
&\leq 2\epsilon t \text{ since } (1-\epsilon)^t \geq 1 - \epsilon t \text{ for } \epsilon \in [0,1]
\end{aligned}$$

In the general case, assume that π_θ is an arbitrary distribution. By definition, $\pi_{\theta'}$ is close to π_θ if $|\pi_{\theta'}(a_t|s_t) - \pi_\theta(a_t|s_t)| \leq \epsilon$ for all s_t. A useful lemma here is that

$$\begin{aligned}
&\text{if } |p_X(x) - p_Y(x)| = \epsilon, \exists p(x,y) \text{ s.t. } p(x) = p_X(x), p(y) = p_Y(y), p(x=y) = 1-\epsilon \\
&\quad \implies p_X(x) \text{ "agrees" with } p_Y(y) \text{ w.p. } \epsilon \\
&\quad \implies \pi_{\theta'}(a_t|s_t) \text{ takes a different action than } \pi_\theta(a_t|s_t) \text{ w.p. at most } \epsilon
\end{aligned}$$

Thus we have again that

$$\begin{aligned}
|p_{\theta'}(s_t) - p_\theta(s_t)| &= (1-(1-\epsilon)^t)|p_{\text{mistake}}(s_t) - p_\theta(s_t)| \\
&\leq 2(1-(1-\epsilon)^t) \\
&\leq 2\epsilon t
\end{aligned}$$

We have shown that if $|\pi_{\theta'}(a_t|s_t) - \pi_\theta(a_t|s_t)| \leq \epsilon$ for all s_t, $|p_{\theta'}(s_t) - p_\theta(s_t) \leq 2\epsilon t$. Then, for some function $f(\cdot)$,

$$\begin{aligned}
\mathbb{E}_{s_t \sim p_{\theta'}(s_t)}\big[f(s_t)\big] &= \sum_{s_t} p_{\theta'}(s_t) f(s_t) \\
&\geq \sum_{s_t} p_\theta(s_t) f(s_t) - |p_\theta(s_t) - p_{\theta'}(s_t)| \max_{s_t} f(s_t) \\
&\geq \mathbb{E}_{s_t \sim p_\theta(s_t)}\big[f(s_t)\big] - 2\epsilon t \max_{s_t} f(s_t)
\end{aligned}$$

Thus, we see that

$$\sum_t \mathbb{E}_{s_t \sim p_{\theta'}(s_t)}\left[\mathbb{E}_{a_t \sim \pi_\theta(a_t|s_t)}\left[\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)}\gamma^t A^{\pi_\theta}(s_t, a_t)\right]\right] \geq$$
$$\sum_t \mathbb{E}_{s_t \sim p_{\theta}(s_t)}\left[\mathbb{E}_{a_t \sim \pi_\theta(a_t|s_t)}\left[\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)}\gamma^t A^{\pi_\theta}(s_t, a_t)\right]\right] - \sum_t 2\epsilon t C$$

where $C = O(Tr_{max})$ if finite and $C = O(\frac{r_{\max}}{1-\gamma})$ if infinite. This means that maximizing the expectation above with respect to θ will maximize the other expectation above with respect to θ', which we have already shown will maximize the RL objective. To summarize what we have so far,

$$\theta' \leftarrow \arg\max_{\theta'} \sum_t \mathbb{E}_{s_t \sim p_{\theta}(s_t)}\left[\mathbb{E}_{a_t \sim \pi_\theta(a_t|s_t)}\left[\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)}\gamma^t A^{\pi_\theta}(s_t, a_t)\right]\right]$$

such that $|\pi_{\theta'}(a_t|s_t) - \pi_\theta(a_t|s_t)| \leq \epsilon$ for small enough ϵ is guaranteed to improve $J(\theta') - J(\theta)$.

8.3 Policy Gradients with Constraints

Total variation distance is hard to optimize since it involves an absolute value, so we will instead use a more convenient bound involving KL divergence. A useful lemma is that

$$|\pi_{\theta'}(a_t|s_t) - \pi_\theta(a_t|s_t)| \leq \sqrt{\frac{1}{2} D_{KL}(\pi_{\theta'}(a_t|s_t) \| \pi_\theta(a_t|s_t))}$$

The KL divergence, $D_{KL}(p_1(x) \| p_2(x)) = \mathbb{E}_{x \sim p_1(x)}\left[\log \frac{p_1(x)}{p_2(x)}\right]$, can then be used to bound the state marginal difference. It has some very convenient properties that make it much easier to approximate. The new objective then becomes

$$\theta' \leftarrow \arg\max_{\theta'} \sum_t \mathbb{E}_{s_t \sim p_{\theta}(s_t)}\left[\mathbb{E}_{a_t \sim \pi_\theta(a_t|s_t)}\left[\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)}\gamma^t A^{\pi_\theta}(s_t, a_t)\right]\right]$$

such that $D_{KL}(\pi_{\theta'}(a_t|s_t) \| \pi_\theta(a_t|s_t)) \leq \epsilon$ for small enough ϵ is guaranteed to improve $J(\theta') - J(\theta)$. We can enforce this constraint with Lagrange multipliers. The Lagrange of this objective is

$$\begin{aligned}\mathcal{L}(\theta', \lambda) = \sum_t \mathbb{E}_{s_t \sim p_{\theta}(s_t)}&\left[\mathbb{E}_{a_t \sim \pi_\theta(a_t|s_t)}\left[\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)}\gamma^t A^{\pi_\theta}(s_t, a_t)\right]\right] \\ &- \lambda(D_{KL}(\pi_{\theta'}(a_t|s_t) \| \pi_\theta(a_t|s_t)) - \epsilon)\end{aligned}$$

We can then do dual gradient descent (more on this later):

1. Maximize $\mathcal{L}(\theta', \lambda)$ with respect to θ' (done incompletely with a few gradient steps in practice because optimization is expensive)
2. $\lambda \leftarrow \lambda + \alpha(D_{KL}(\pi_{\theta'}(a_t|s_t)\|\pi_\theta(a_t|s_t)) - \epsilon)$

The intuition is that we raise λ if constraint is violated too much (i.e. divergence greater than ϵ) and lower it otherwise (i.e. divergence less than ϵ).

8.4 Natural Gradient

How else can we optimize our objective

$$\theta' \leftarrow \arg\max_{\theta'} \sum_t \mathbb{E}_{s_t \sim p_\theta(s_t)} \left[\mathbb{E}_{a_t \sim \pi_\theta(a_t|s_t)} \left[\frac{\pi_{\theta'}(a_t|s_t)}{\pi_\theta(a_t|s_t)} \gamma^t A^{\pi_\theta}(s_t, a_t) \right] \right] = \arg\max_{\theta'} \bar{A}(\theta')$$

such that $D_{KL}(\pi_{\theta'}(a_t|s_t)\|\pi_\theta(a_t|s_t)) \leq \epsilon$ for small enough ϵ is guaranteed to improve $J(\theta') - J(\theta)$. We could use first order Taylor approximation for the objective since it is constrained:

$$\theta' \leftarrow \arg\max_{\theta'} \nabla_\theta \bar{A}(\theta)^T (\theta' - \theta)$$

such that $D_{KL}(\pi_{\theta'}(a_t|s_t)\|\pi_\theta(a_t|s_t)) \leq \epsilon$. Note that

$$\nabla_\theta \bar{A}(\theta) = \sum_t \mathbb{E}_{s_t \sim p_\theta(s_t)} \left[\mathbb{E}_{a_t \sim \pi_\theta(a_t|s_t)} \left[\gamma^t \nabla_\theta \log \pi_\theta(a_t|s_t) A^{\pi_\theta}(s_t, a_t) \right] \right] = \nabla_\theta J(\theta)$$

which is exactly the normal policy gradient. So our objective is:

$$\theta' \leftarrow \arg\max_{\theta'} \nabla_\theta J(\theta)^T (\theta' - \theta)$$

such that $D_{KL}(\pi_{\theta'}(a_t|s_t)\|\pi_\theta(a_t|s_t)) \leq \epsilon$. Why can't we just use gradient ascent?

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$

The issue with this is that some parameters change probabilities a lot more than others, and thus most likely does not respect the KL divergence constraint. More concretely, gradient ascent does this:

$$\theta' \leftarrow \arg\max_{\theta'} \nabla_\theta J(\theta)^T (\theta' - \theta)$$

such that $\|\theta - \theta'\|^2 \leq \epsilon$, with the solution being

$$\theta' = \theta + \sqrt{\frac{\epsilon}{\|\nabla_\theta J(\theta)\|^2}} \nabla_\theta J(\theta)$$

Note that this constraint is not the same as the KL-divergence. We can approximate KL-divergence with a second order Taylor expansion

$$D_{KL}(\pi_{\theta'}\|\pi_\theta) \approx \frac{1}{2}(\theta' - \theta)^T F (\theta' - \theta)$$

where F is the Fischer-information matrix, which can be estimated with samples:

$$F = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) \nabla_\theta \log \pi_\theta(a|s)^T]$$

Using the Lagrangian, we can show that

$$\theta' = \theta + \alpha F^{-1} \nabla_\theta J(\theta)$$

If we set an ϵ constraint, then the appropriate α would be

$$\alpha = \sqrt{\frac{2\epsilon}{\nabla_\theta J(\theta)^T F \nabla_\theta J(\theta)}}$$

Practically, natural policy gradient is generally a good choice to stabilize policy gradient training. For instance, with a policy with

$$\log \pi_\theta(a_t|s_t) = -\frac{1}{2\sigma^2}(ks_t - a_t)^2 + \text{const}$$

, vanilla policy gradient doesn't converge at the optimal solution because σ affects the log probability when it is small much more than k. As a result, its corresponding gradient component is much larger. With natural policy gradient, the Fischer information matrix makes the gradient much more well conditioned since it does gradient descent in the "probability" space rather than "theta" space. In trust region policy optimization, we see a non-trivial way of computing Fischer-vector products without computing the full matrix. Alternatively, we could use proximal policy optimization through dual gradient descent or a fixed regularization constant as discussed in the above section.

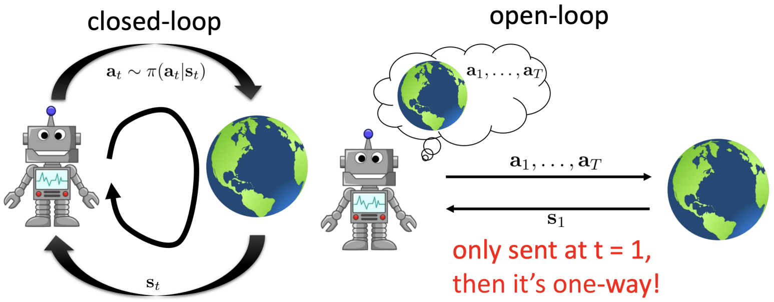
Chapter 9

Model-Based Planning

9.1 Optimal Planning and Control

Recall that in model-free RL (as we have seen so far), we assume the transition probabilities are unknown and don't attempt to learn it. However, what if we knew the transition dynamics? Often, we know the dynamics such as in games, easily modeled systems, and simulated environment. Other times, we can learn the dynamics with system identification or just general-purpose learning. In model-based reinforcement learning, we learn the transition dynamics and then figure out how to choose actions. In this chapter, we will look at how to choose actions under perfect knowledge of system dynamics. Optimal control (general problem of selecting controls to maximize reward or minimize cost), trajectory optimization (selecting a sequence of states/actions to optimize some outcome), and planning (generally the discrete analog of trajectory optimization) fall under this umbrella.

In control, we have open-loop and closed-loop systems.


closed-loop
$\mathbf{a}_t \sim \pi(\mathbf{a}_t|\mathbf{s}_t)$
$\mathbf{s}_t$
open-loop
$\mathbf{a}_1, \ldots, \mathbf{a}_T$
$\mathbf{a}_1, \ldots, \mathbf{a}_T$
$\mathbf{s}_1$
only sent at t = 1,
then it's one-way!

In open-loop systems, the agent is only sent $t = 1$ and must then provide all actions. In closed-loop systems, the agent instead provides a mapping from states to actions. Let's first look at open-loop systems. Since there is no policy anymore, our objective becomes

$$\min_{a_1,...,a_T} \sum_{t=1}^{T} c(s_t, a_t) \text{ s.t. } s_t = f(s_{t-1}, a_{t-1})$$

This is objective is for the deterministic open-loop case, which is pretty straightforward. In the stochastic open-loop case, our objective becomes

$$\arg\max_{a_1,...,a_T} \mathbb{E}\left[\sum_{t=1}^{T} r(s_t, a_t) | a_1, ..., a_T\right]$$

with the expecation taken under

$$p_\theta(s_1, ..., s_T | a_1, ..., a_T) = p(s_1) \prod_{t=1}^{T} p(s_{t+1} | s_t, a_t)$$

However, this is suboptimal because there are many cases where information will be revealed to us in the future that will be useful for taking better actions. Thus, many stochastic cases are better suited with closed-loop control, where (as we have seen before)

$$\pi = \arg\max_{\pi} \mathbb{E}_{\tau \sim p(\tau)}\left[\sum_t r(s_t, a_t)\right]$$

such that $p(\tau) = p(s_1, a_1, ..., s_T, a_T) = p(s_1) \prod_{t=1}^{T} \pi(a_t|s_t) p(s_{t+1}|s_t, a_t)$

9.2 Open-Loop Planning

In stochastic optimization, we can abstract away optimal control and planning from $\arg\max_{a_1,...,a_T} J(a_1, ..., a_T)$ to become $\arg\max_A J(A)$.

9.2.1 Random Shooting Method

The random shooting method is just guess and check where we

Algorithm 16 Random Shooting Method

```
1: pick $A_1, ..., A_N$ from some distribution (e.g. uniform)
2: choose $A_i$ based on $\arg\max_i J(A_i)$
```

9.2.2 Cross-Entropy Method (CEM)

CEM is like random shooting but we refit the distribution we sample from to the best A_i sampled, allowing us to converge on optimal solutions. We could also use CMA-ES, which is like CEM with momentum. The upside to CEM is that it is parallelizable and simple. However, it has a very harsh dimensionality limit and is only for open-loop planning.

Algorithm 17 Cross-Entropy Method (CEM)

```
while some stop condition is not satisfied do
    sample $A_1, ..., A_N$ from $p(A)$
    evaluate $J(A_1), ..., J(A_N)$
    pick the elites $A_{i_1}, ..., A_{i_M}$ with the highest value, where $M < N$
    refit $p(A)$ to the elites $A_{i_1}, ..., A_{i_M}$
end while
```

9.2.3 Monte Carlo Tree Search (MCTS)

MCTS formulates discrete planning as a search problem, where we choose where to search first by running a policy to get a sample estimate of the reward and choosing nodes with the best reward while also preferring rarely visited nodes.

Algorithm 18 Monte Carlo Tree Search (MCTS)

```
for t ← 1 to T do
    while some stop condition is not satisfied do
        find a leaf $s_l$ using TreePolicy($s_t$)
        evaluate the leaf using DefaultPolicy($s_l$)
        update all values in the tree between $s_t$ and $s_l$.
    end while
    take best action from $s_t$
end for
```

Here, the subscript for s indicates the timestep and doesn't distinguish different states. One common TreePolicy is UCT TreePolicy, where if s_t is not fully expanded, choose new a_t else choose child with best Score(s_{t+1}) where

$$\text{Score}(s_t) = \frac{Q(s_t)}{N(s_t)} + 2C\sqrt{\frac{2\ln N(s_{t-1})}{N(s_t)}}$$

where $Q(s_t)$ is the total reward gained through s_t and $N(s_t)$ is the number of times s_t has been traversed through. Intuitively, $\frac{Q(s_t)}{N(s_t)}$ is the average reward gained through traversing s_t. We then add a bonus of $2C\sqrt{\frac{2\ln N(s_{t-1})}{N(s_t)}}$. The denominator makes it so that the value of the bonus decreases as the state gets traversed more and more because we want to prioritize rarely explored nodes. The numerator makes it so that the value of the bonus increases as the parent node is traversed more because then we are more confident in value of the node.

9.3 Trajectory Optimization with Derivatives

In trajectory optimization, we want to

$$\min_{u_1,\ldots,u_T} \sum_{t=1}^{T} c(x_t, u_t) \text{ s.t. } x_t = f(x_{t-1}, u_{t-1})$$

which unrolled is

$$\min_{u_1,\ldots,u_T} c(x_1, u_1) + c(f(x_1, u_1,), u_2) + ... + c(f(f(...)...), u_T)$$

where x_t is analogous to s_t and u_t is analogous to a_t (control notation). In the shooting method, we optimize over actions only:

$$\min_{u_1,\ldots,u_T} c(x_1, u_1) + c(f(x_1, u_1,), u_2) + ... + c(f(f(...)...), u_T)$$

while in the collocation method we optimize over both states and actions, with constraints:

$$\min_{u_1,\ldots,u_T,x_1,\ldots,x_T} \sum_{t=1}^{T} c(x_t, u_t) \text{ s.t. } x_t = f(x_{t-1}, u_{t-1})$$

9.3.1 Linear Quadratic Regulator (LQR)

In LQR, we assume $f(\cdot)$ is linear and $c(\cdot)$ is quadratic, so we have

$$\min_{u_1,\ldots,u_T} c(x_1, u_1) + c(f(x_1, u_1,), u_2) + ... + c(f(f(...)...), u_T)$$

where $f(x_t, u_t) = \mathbf{F}_t \begin{bmatrix} x_t \\ u_t \end{bmatrix} + \mathbf{f}_t$ and $c(x_t, u_t) = \frac{1}{2} \begin{bmatrix} x_t \\ u_t \end{bmatrix}^T \mathbf{C}_t \begin{bmatrix} x_t \\ u_t \end{bmatrix} + \begin{bmatrix} x_t \\ u_t \end{bmatrix}^T \mathbf{c}_t$.

For our base case, let's solve for u_T only. Let $c(f(f(...)...), u_T) = c(x_T, u_T)$ where x_T is currently unknown to us. Then, our Q-value is

$$Q(x_T, u_T) = \text{const} + \frac{1}{2} \begin{bmatrix} x_T \\ u_T \end{bmatrix}^T \mathbf{C}_T \begin{bmatrix} x_T \\ u_T \end{bmatrix} + \begin{bmatrix} x_T \\ u_T \end{bmatrix}^T \mathbf{c}_T$$

Let $\mathbf{C}_T = \begin{bmatrix} \mathbf{C}_{x_T,x_T} & \mathbf{C}_{x_T,u_T} \\ \mathbf{C}_{u_T,x_T} & \mathbf{C}_{u_T,u_T} \end{bmatrix}$ and $\mathbf{c}_T = \begin{bmatrix} \mathbf{c}_{x_T} \\ \mathbf{c}_{u_T} \end{bmatrix}$. We can calculate the optimal u_T by taking a gradient and setting it to zero:

$$\begin{aligned} \nabla_{u_T} Q(x_T, u_T) &= \mathbf{C}_{u_T,x_T} x_T + \mathbf{C}_{u_T,u_T} u_T + \mathbf{c}_{u_T} = 0 \\ u_T &= -\mathbf{C}_{u_T,u_T}^{-1} (\mathbf{C}_{u_T,x_T} x_T + \mathbf{c}_{u_T}) \\ &= \mathbf{K}_T x_T + \mathbf{k}_T \end{aligned}$$

where $\mathbf{K}_T = -\mathbf{C}^{-1}_{u_T,u_T}\mathbf{C}_{u_T,x_T}$ and $\mathbf{k}_T = -\mathbf{C}^{-1}_{u_T,u_T}\mathbf{c}_{u_T}$. With this, we can now calculate the value at timestep T:

$$\begin{aligned}
V(x_T) &= \text{const} + \frac{1}{2}\begin{bmatrix} x_T \\ \mathbf{K}_T x_T + \mathbf{k}_T \end{bmatrix}^T \mathbf{C}_T \begin{bmatrix} x_T \\ \mathbf{K}_T x_T + \mathbf{k}_T \end{bmatrix} + \begin{bmatrix} x_T \\ \mathbf{K}_T x_T + \mathbf{k}_T \end{bmatrix}^T \mathbf{c}_T \\
&= \text{const} + \frac{1}{2} x_T^T \mathbf{V}_T x_T + x_T^T \mathbf{v}_T
\end{aligned}$$

where

$$\begin{aligned}
\mathbf{V}_T &= \mathbf{C}_{x_T,x_T} + \mathbf{C}_{x_T,u_T}\mathbf{K}_T + \mathbf{K}_T^T\mathbf{C}_{u_T,x_T} + \mathbf{K}_T^T\mathbf{C}_{u_T,u_T}\mathbf{K}_T \\
\mathbf{v}_T &= \mathbf{c}_{x_T} + \mathbf{C}_{x_T,u_T}\mathbf{k}_T + \mathbf{K}_T^T\mathbf{C}_{u_T} + \mathbf{K}_T^T\mathbf{C}_{u_T,u_T}\mathbf{k}_T
\end{aligned}$$

Now, let's solve for $\mathbf{u}_{T-1}$ in terms of $\mathbf{x}_{T-1}$.

$$\begin{aligned}
&f(x_{T-1}, u_{T-1}) = x_T = \mathbf{F}_{T-1}\begin{bmatrix} x_{T-1} \\ u_{T-1} \end{bmatrix} + \mathbf{f}_{T-1} \\
&Q(x_{T-1}, u_{T-1}) = \text{const} + \frac{1}{2}\begin{bmatrix} x_{T-1} \\ u_{T-1} \end{bmatrix}^T \mathbf{C}_{T-1}\begin{bmatrix} x_{T-1} \\ u_{T-1} \end{bmatrix} + \begin{bmatrix} x_{T-1} \\ u_{T-1} \end{bmatrix}^T \mathbf{c}_{T-1} + V(f(x_{T-1}, u_{T-1})) \\
&V(x_T) = \text{const} + \frac{1}{2}\begin{bmatrix} x_{T-1} \\ u_{T-1} \end{bmatrix}^T \mathbf{F}_{T-1}^T\mathbf{V}_T\mathbf{F}_{T-1}\begin{bmatrix} x_{T-1} \\ u_{T-1} \end{bmatrix} + \begin{bmatrix} x_{T-1} \\ u_{T-1} \end{bmatrix}^T \mathbf{F}_{T-1}^T\mathbf{V}_T\mathbf{f}_{T-1} + \begin{bmatrix} x_{T-1} \\ u_{T-1} \end{bmatrix}^T \mathbf{F}_{T-1}^T\mathbf{v}_T \\
&Q(x_{T-1}, u_{T-1}) = \text{const} + \frac{1}{2}\begin{bmatrix} x_{T-1} \\ u_{T-1} \end{bmatrix}^T \mathbf{Q}_{T-1}\begin{bmatrix} x_{T-1} \\ u_{T-1} \end{bmatrix} + \begin{bmatrix} x_{T-1} \\ u_{T-1} \end{bmatrix}^T \mathbf{q}_{T-1}
\end{aligned}$$

where

$$\begin{aligned}
\mathbf{Q}_{T-1} &= \mathbf{C}_{T-1} + \mathbf{F}_{T-1}^T\mathbf{V}_T\mathbf{F}_{T-1} \\
\mathbf{q}_{T-1} &= \mathbf{c}_{T-1} + \mathbf{F}_{T-1}^T\mathbf{V}_T\mathbf{f}_{T-1} + \mathbf{F}_{T-1}^T\mathbf{v}_T
\end{aligned}$$

Taking the gradient, we see that

$$\begin{aligned}
\nabla_{u_{T-1}} Q(x_{T-1}, u_{T-1}) &= \mathbf{Q}_{u_{T-1},x_{T-1}} x_{T-1} + \mathbf{Q}_{u_{T-1},u_{T-1}} u_{T-1} + \mathbf{q}_{u_{T-1}} = 0 \\
u_{T-1} &= -\mathbf{Q}^{-1}_{u_{T-1},u_{T-1}}(\mathbf{Q}_{u_{T-1},x_{T-1}} x_{T-1} + \mathbf{q}_{u_{T-1}}) \\
&= \mathbf{K}_{T-1} x_{T-1} + \mathbf{k}_{T-1}
\end{aligned}$$

We can inductively perform this process up until the first timestep, motivating LQR.

Algorithm 19 LQR

```
for $t = T$ to 1 do
    $\mathbf{Q}_t = \mathbf{C}_t + \mathbf{F}_t^T \mathbf{V}_{t+1} \mathbf{F}_t$
    $\mathbf{q}_t = \mathbf{c}_t + \mathbf{F}_t^T \mathbf{V}_{t+1} \mathbf{f}_t + \mathbf{F}_t^T \mathbf{v}_{t+1}$
    $Q(x_t, u_t) = \text{const} + \frac{1}{2} \begin{bmatrix} x_t \\ u_t \end{bmatrix}^T \mathbf{Q}_t \begin{bmatrix} x_t \\ u_t \end{bmatrix} + \begin{bmatrix} x_t \\ u_t \end{bmatrix}^T \mathbf{q}_t$
    $u_t \leftarrow \arg\min_{u_t} Q(x_t, u_t) = \mathbf{K}_t x_t + \mathbf{k}_t$
    $\mathbf{K}_t = -\mathbf{Q}_{u_t,u_t}^{-1} \mathbf{Q}_{u_t,x_t}$
    $\mathbf{k}_t = -\mathbf{Q}_{u_t,u_t}^{-1} \mathbf{q}_{u_t}$
    $\mathbf{V}_t = \mathbf{Q}_{x_t,x_t} + \mathbf{Q}_{x_t,u_t} \mathbf{K}_t + \mathbf{K}_t^T \mathbf{Q}_{u_t,x_t} + \mathbf{K}_t^T \mathbf{Q}_{u_t,u_t} \mathbf{K}_t$
    $\mathbf{v}_t = \mathbf{q}_{x_t} + \mathbf{Q}_{x_t,u_t} \mathbf{k}_t + \mathbf{K}_t^T \mathbf{Q}_{u_t} + \mathbf{K}_t^T \mathbf{Q}_{u_t,u_t} \mathbf{k}_t$
    $V(x_t) = \text{const} + \frac{1}{2} x_t^T \mathbf{V}_t x_t + x_t^T \mathbf{v}_t$
end for
for $t = 1$ to $T$ do
    $u_t = \mathbf{K}_t x_t + \mathbf{k}_t$
    $x_{t+1} = f(x_t, u_t)$
end for
```

9.4 LQR for Stochastic and Nonlinear Systems

9.4.1 Stochastic Dynamics

With stochastic dynamics, we have $x_{t+1} \sim p(x_{t+1}|x_t, u_t)$. In the special case that the dynamics are gaussian, where $p(x_{t+1}|x_t, u_t) = \mathcal{N}(\mathbf{F}_t \begin{bmatrix} x_t \\ u_t \end{bmatrix} + \mathbf{f}_t, \Sigma_t)$, the optimal control law is still $u_t = \mathbf{K}_t x_t + \mathbf{k}_t$. However, instead of getting a single sequence of actions, we get a closed-loop policy.

9.4.2 Nonlinear Dynamics

If $f(x_t, u_t)$ and $c(x_t, u_t)$ are nonlinear, we can approximate it as linear-quadratic:

$$f(x_t, u_t) \approx f(\hat{x}_t, \hat{u}_t) + \nabla_{x_t,u_t} f(\hat{x}_t, \hat{u}_t) \begin{bmatrix} x_t - \hat{x}_t \\ u_t - \hat{u}_t \end{bmatrix}$$

$$c(x_t, u_t) \approx c(\hat{x}_t, \hat{u}_t) + \nabla_{x_t,u_t} c(\hat{x}_t, \hat{u}_t) \begin{bmatrix} x_t - \hat{x}_t \\ u_t - \hat{u}_t \end{bmatrix} + \frac{1}{2} \begin{bmatrix} x_t - \hat{x}_t \\ u_t - \hat{u}_t \end{bmatrix}^T \nabla^2_{x_t,u_t} c(\hat{x}_t, \hat{u}_t) \begin{bmatrix} x_t - \hat{x}_t \\ u_t - \hat{u}_t \end{bmatrix}$$

where $\hat{x}_t$ and $\hat{u}_t$ are the best states/actions we have found so far. Let

$$\bar{f}(\delta x_t, \delta u_t) = \nabla_{x_t,u_t} f(\hat{x}_t, \hat{u}_t) \begin{bmatrix} \delta x_t \\ \delta u_t \end{bmatrix} = \mathbf{F}_t \begin{bmatrix} \delta x_t \\ \delta u_t \end{bmatrix}$$

$$\bar{c}(\delta x_t, \delta u_t) = \frac{1}{2} \begin{bmatrix} \delta x_t \\ \delta u_t \end{bmatrix}^T \nabla^2_{x_t,u_t} c(\hat{x}_t, \hat{u}_t) \begin{bmatrix} \delta x_t \\ \delta u_t \end{bmatrix} + \begin{bmatrix} \delta x_t \\ \delta u_t \end{bmatrix}^T \nabla_{x_t,u_t} c(\hat{x}_t, \hat{u}_t) = \frac{1}{2} \begin{bmatrix} \delta x_t \\ \delta u_t \end{bmatrix}^T \mathbf{C}_t \begin{bmatrix} \delta x_t \\ \delta u_t \end{bmatrix} + \begin{bmatrix} \delta x_t \\ \delta u_t \end{bmatrix}^T \mathbf{c}_t$$

where $\delta x_t = x_t - \hat{x}_t$ and $\delta u_t = u_t - \hat{u}_t$. Then we can run LQR on $\bar{f}$, $\bar{c}$, δx_t, and δu_t:

Algorithm 20 Iterative LQR

```
1: for until convergence do
2:     $\mathbf{F}_t = \nabla_{x_t,u_t} f(\hat{x}_t, \hat{u}_t)$
3:     $\mathbf{c}_t = \nabla_{x_t,u_t} c(\hat{x}_t, \hat{u}_t)$
4:     $\mathbf{C}_t = \nabla^2_{x_t,u_t} c(\hat{x}_t, \hat{u}_t)$
5:     Run LQR backward pass on $\delta x_t$ and $\delta u_t$
6:     Run forward pass: $u_t = \mathbf{K}_t(x_t - \hat{x}_t) + \mathbf{k}_t + \hat{u}_t$
7:     Update $\hat{x}_t$ and $\hat{u}_t$ based on states and actions in forward pass
8: end for
```

Let's compare iLQR to Newton's method for computing $\min_x g(x)$:

Algorithm 21 Newton's Method

```
for until convergence do
    $g = \nabla_x g(\hat{x})$
    $H = \nabla^2_x g(\hat{x})$
    $\hat{x} \leftarrow \arg\min_x \frac{1}{2}(x - \hat{x})^T H (x - \hat{x}) + g^T (x - \hat{x})$
end for
```

iLQR is the same idea: we locally approximate a complex nonlinear function via Taylor expansion. iLQR is actually really just Newton's method for solving

$$\min_{u_1,\ldots,u_T} c(x_1, u_1) + c(f(x_1, u_1,), u_2) + \ldots + c(f(f(\ldots)\ldots), u_T)$$

but with first order dynamics. We could also use second order dynamics, which would then be exactly Newton's method for solving this objective. This is what differential dynamic programming (DDP) does. The connection to Newton's method also allows us to derive an improvement to iLQR. Notice that in Newton's method, $\hat{x} \leftarrow \arg\min_x \frac{1}{2}(x - \hat{x})^T H (x - \hat{x}) + g^T (x - \hat{x})$ may be a bad idea as we could be overshooting the optimal. Thus, we want to first compute our solution and then check if our solution is actually better than what we had before. And if it is not better, we want to move closer to where we were before. We can do this by simply adding an α term to line 6 in the iLQR algorithm below that is a constant between 0 and 1. This constant α allows us to control how much we deviate from our starting point. Notice that if $\alpha = 0$, we will execute the exact same sequence of actions as we did before. In general, as we reduce α, we will get closer and closer to the action sequence before. We can search over α until we get improvement.

Algorithm 22 Iterative LQR (with α)

```
for until convergence do
    $\mathbf{F}_t = \nabla_{x_t,u_t} f(\hat{x}_t, \hat{u}_t)$
    $\mathbf{c}_t = \nabla_{x_t,u_t} c(\hat{x}_t, \hat{u}_t)$
    $\mathbf{C}_t = \nabla^2_{x_t,u_t} c(\hat{x}_t, \hat{u}_t)$
    Run LQR backward pass on $\delta x_t$ and $\delta u_t$
    Run forward pass: $u_t = \mathbf{K}_t(x_t - \hat{x_t}) + \alpha \mathbf{k}_t + \hat{u}_t$
    Update $\hat{x}_t$ and $\hat{u}_t$ based on states and actions in forward pass
end for
```

Chapter 10

Model-Based Reinforcement Learning

10.1 Model-Based RL Basics

If we don't know $f(s_t, a_t)$ (or $p(s_{t+1}|s_t, a_t)$ in the stochastic case), then we can learn it from data and then plan through it. This gives us a naive model-based RL:

Algorithm 23 Model-Based Reinforcement Learning Version 0.5

```
run base policy $\pi_0(a_t|s_t)$ (e.g. random policy) to collect $\mathcal{D} = \{(s, a, s')_i\}$
learn dynamics model $f(s, a)$ to minimize $\sum_i \|f(s_i, a_i) - s'_i\|^2$
plan through $f(s, a)$ to choose actions
```

In step 3, we can use algorithms from last chapter like LQR to do this. This naive algorithms is particularly effective if we can hand-engineer a dynamics representation of our knowledge of physics and then fit a few parameters. However, the distribution mismatch problem (i.e. the base policy data distribution is different from real world data distribution $p_{\pi_f}(s_t) \neq p_{\pi_0}(s_t)$) becomes exacerbated as we use more expressive model classes. We can do better by essentially doing DAgger by collecting data from $p_{\pi_f}(s_t)$. We will call this model-based RL version 1.0:

Algorithm 24 Model-Based Reinforcement Learning Version 1.0

```
run base policy $\pi_0(a_t|s_t)$ (e.g. random policy) to collect $\mathcal{D} = \{(s, a, s')_i\}$
for until some stop condition is satisfied do
    learn dynamics model $f(s, a)$ to minimize $\sum_i \|f(s_i, a_i) - s'_i\|^2$
    plan through $f(s, a)$ to choose actions
    execute those actions and add the resulting data $\{(s, a, s')_j\}$ to $\mathcal{D}$
end for
```

In this case, since we are doing open-loop planning, any small mistake in the dynamics model will compound over time. Replanning helps with model errors, motivating model predictive control (MPC):

Algorithm 25 Model-Based Reinforcement Learning Version 1.5

```
run base policy $\pi_0(a_t|s_t)$ (e.g. random policy) to collect $\mathcal{D} = \{(s, a, s')_i\}$
for until some stop condition is satisfied do
    learn dynamics model $f(s, a)$ to minimize $\sum_i \|f(s_i, a_i) - s'_i\|^2$
    for N steps do
        plan through $f(s, a)$ to choose actions
        execute the first planned action, observe resulting state $s'$ (MPC)
        append $\{(s, a, s')\}$ to dataset $\mathcal{D}$
    end for
end for
```

For the planning step, the more we replan, the less perfect each individual plan needs to be. So, we could use shorter horizons and even random sampling to reduce computational costs for needing to replan every timestep.

10.2 Uncertainty in Model-Based RL

In practice, model-based RL does worse than model-free RL. This is because we need our model to not overfit in the beginning but still have high capacity later on when there is more data. However, high capacity models do pretty poorly in early stages and the agent ends up getting stuck. If a model overfits, the planner exploits these mistakes. In uncertainty estimation, we instead predict a distribution of next states we can reach under a distribution of the uncertainty of the model. We want to take actions that are good in expectation of the distribution of all possible worlds under the dataset. By taking the expected value, note that the expected reward under high-variance prediction is very low even if the mean prediction leads us to a high reward. Thus, in our planning phase of MPC, we will only take actions for which we think we'll get high reward in expectation with respect to uncertain dynamics. This avoids exploiting the model, and the model will then be able to adapt and get better. There a few caveats: our planner needs to explore to get better, and expected value is the same as pessimistic or optimistic value but is often a good start.

10.3 Uncertainty-Aware Neural Net Models

How can we create uncertainty-aware models? One idea is to use output entropy of probability distribution outputted by model. However, this is not enough because the entropy measures aleatoric or statistical uncertainty (i.e. how noisy the dynamics are). What we want is epistemic or model uncertainty (i.e. the uncertainty about the model). In other words, what we want is to figure out when the model is certain about the training data, but we are not certain about the model. Maximum likelihood where we find $\arg\max_\theta \log p(\theta|\mathcal{D})$ cannot do this. Instead, we want to estimate the full $p(\theta|\mathcal{D})$ because the entropy of this tells us the model uncertainty. In theory, we can then predict according to $\int p(s_{t+1}|s_t, a_t, \theta)p(\theta|\mathcal{D})d\theta$ which integrates out all uncertainty. Note this is clearly intractable for large-dimensional parameter spaces, so we have a few ways to approximated this.

10.3.1 Bayesian Neural Networks

In bayesian neural networks, instead of the weights being deterministic, they are each a distribution. We can then estimate the posterior by repeatedly sampling the neural net. Our prediction would then just be the average of the model predictions over all samples of the model. Modeling full joint distribution over the parameters is very difficult since parameters are high-dimensional, so we can approximate the posterior $p(\theta|\mathcal{D}) = \prod_i p(\theta_i|\mathcal{D})$. A common choice is to represent $p(\theta_i|\mathcal{D}) = \mathcal{N}(\mu_i, \sigma_i)$.

10.3.2 Bootstrap Ensembles

Another choice is to use bootstrap ensembles, where we have N ”independent” models trained on the ”independent” datasets and see if they agree. Formally, we estimate the posterior $p(\theta|\mathcal{D}) \approx \frac{1}{N}\sum_i \delta(\theta_i)$ (i.e. each model in the ensemble represents a dirac delta distribution and we average these distributions). Our prediction is then $\int p(s_{t+1}|s_t, a_t, \theta)p(\theta|\mathcal{D})d\theta \approx \frac{1}{N}\sum_i p(s_{t+1}|s_t, a_t, \theta_i)$. Note that we are averaging the distributions and not their realizations (e.g. if the distributions were gaussian, we have a mixture of gaussians). In order for the models to be trained on ”independent” datasets, we can sample a subset of the training data with replacement for each model. Resampling with replacement, in practice, is usually unnecessary because SGD and random initialization usually makes the models ufficiently independent. Bootstrap ensembles is a crude approximation because the number of models is usually small (less than 10).

10.4 Planning With Uncertainty, Examples

Before, we calculated reward as

$$J(a_1, ..., a_H) = \sum_{t=1}^{H} r(s_t, a_t), \text{ where } s_{t+1} = f(s_t, a_t)$$

With uncertainty planning, we now have

$$J(a_1, ..., a_H) = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{H} r(s_{t,i}, a_t), \text{ where } s_{t+1,i} = f_i(s_{t,i}, a_t)$$

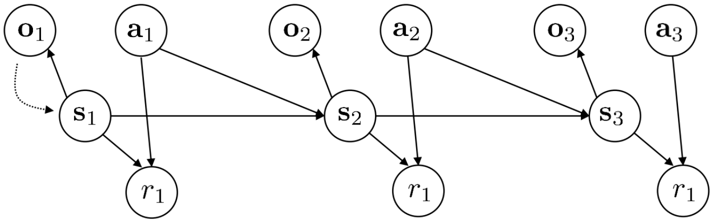
In general, for candidate action sequence $a_1, ..., a_H$, we

1. sample $\theta \sim p(\theta|\mathcal{D})$ (via BNN or bootstrap ensembles)
2. at each time step t, sample $s_{t+1} \sim p(s_{t+1}|s_t, a_t, \theta)$
3. calculate $R = \sum_t r(s_t, a_t)$
4. repeat steps 1 to 3 and accumulate the average reward

Other more advanced options for uncertainty planning include moment matching, more complex posterior estimation using BNNs, etc.

10.5 Model-Based RL with Images

With complex observations such as images, we have a hard time creating a dynamics model since the observations have high dimensionality, redundancy, and are partially observable. To solve this, we can learn a latent representation of the image. In other words, we separately learn $p(o_t|s_t)$ (high-dimensional and not dynamic) and $p(s_{t+1}|s_t, a_t)$ (low-dimension but dynamic). Our latent space model will have the observation model $p(o_t|s_t)$, dynamics model $p(s_{t+1}|s_t, a_t)$, and reward model $p(r_t|s_t, a_t)$:



The latent space model objective is

$$\max_{\phi} \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \mathbb{E}_{(s_t, s_{t+1}) \sim p(s_t, s_{t+1}|o_{1:T}, a_{1:T})} [\log p_\phi(s_{t+1,i}|s_{t,i}, a_{t,i}) + \log p_\phi(o_{t,i}|s_{t,i})]$$

We can learn an approximate posterior $q_\psi(s_t|o_{1:t}, a_{1:t})$, which we call an encoder. There are many choices for approximate posterior, ranging from the most complicated and accurate full smoothing posterior $q_\psi(s_t, s_{t+1}|o_{1:T}, a_{1:T})$ to the simplest and least accurate single-step encoder $q_\psi(s_t|o_t)$. In this section, we will talk about $q_\psi(s_t|o_t)$ as the more complicated posterior required variational inference which we will cover later. We can also make the additional simplification that $q(s_t|o_t)$ is deterministic. We will call this deterministic single-step encoder $g_\psi(o_t)$. The new objective then becomes

$$\max_{\phi, \psi} \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \log p_\phi(g_\psi(o_{t+1,i})|g_\psi(o_{t,i}), a_{t,i}) + \log p_\phi(o_{t,i}|g_\psi(o_{t,i}))$$

Since everything is differentiable, we can train with backprop. With rewards, our objective becomes

$$\max_{\phi,\psi} \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \log p_\phi(g_\psi(o_{t+1,i})|g_\psi(o_{t,i}), a_{t,i}) + \log p_\phi(o_{t,i}|g_\psi(o_{t,i})) + \log p_\phi(r_{t,i}|g_\psi(o_{t,i}))$$

The three expression are the latent space dynamics, image construction, and reward model respectively. With everything, we can now derive model-based RL with latent space models:

Algorithm 26 Model-Based Reinforcement Learning with Latent Space Models

```
run base policy $\pi_0(a_t|s_t)$ (e.g. random policy) to collect $\mathcal{D} = \{(o, a, o')_i\}$
for until some stop condition is satisfied do
    learn $p_\phi(s_{t+1}|s_t, a_t), p_\phi(r_t|s_t), p(o_t|s_t), g_\psi(o_t)$
    for N steps do
        plan through $f(s, a)$ to choose actions
        execute the first planned action, observe resulting $o'$ (MPC)
        append $\{(o, a, o')\}$ to dataset $\mathcal{D}$
    end for
end for
```

As a side note, we could also directly learn $p(o_{t+1}|o_t, a_t)$ instead of using an encoder.