CS 285

Preston Fu

Fall 2023

These are course notes for the Fall 2023 rendition of CS 285, Deep Reinforcement Learning, by Prof. Sergey Levine.

Contents

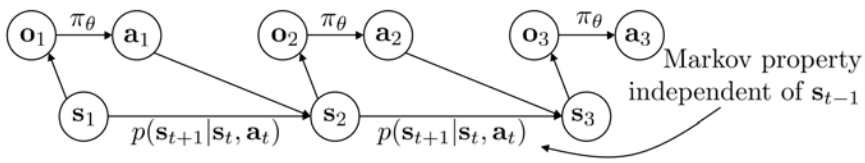


Figure 1: States, observations, and actions in RL.

1 Introduction and Course Overview (08/23)

This lecture was an overview of the course and RL. The notes are here.

2 Imitation Learning (08/28)

The videos begin here.

Notation

- $\mathbf{s}_t$ = state
- $\mathbf{o}_t$ = observation [possible to infer this from state, but sometimes not vice-versa]
- $\mathbf{a}_t$ = action
- $\pi_\theta(\mathbf{a}_t \mid \mathbf{o}_t)$ = policy [a distribution of actions over a particular observation]
- $\pi_\theta(\mathbf{a}_t \mid \mathbf{s}_t)$ = policy (fully observed)

What is imitation learning?

Example 2.1. Consider automatic driving, where you are given some pixels and want to determine whether to go left or right. You can provide some training data from a human and run supervised learning, i.e. behavioral cloning.

In general, this fails: supervised learning can produce small mistakes, and over time, the expected trajectory will continue to deviate farther from the training trajectory (the **distributional shift problem**). Specifically, we train under $p_{\text{data}}(\mathbf{o}_t)$, and our supervised learning approach computes

$$\max_\theta \mathbb{E}_{\mathbf{o}_t \sim p_{\text{data}}(\mathbf{o}_t)}[\log \pi_\theta(\mathbf{a}_t \mid \mathbf{o}_t)].$$

However, we test under $p_{\pi_\theta}(\mathbf{o}_t)$.

It is possible to address this by being smart about how we collect and augment our data (in this example, adding left- and right-facing cameras allowed for the introduction of states the model is familiar with), as well as improving our models. Other methods include using multi-task learning and changing the algorithm entirely.

Behavioral cloning

We use the latter approach by clarifying more precisely what we want. A learned policy is good if it chooses the best action most often. Accordingly, define

$$c(\mathbf{s}_t, \mathbf{a}_t) = \mathbb{1}\{\mathbf{a}_t = \pi^*(\mathbf{s}_t)\}.$$

Our goal is

$$\min_\theta \mathbb{E}_{\mathbf{s}_t \sim p_{\pi_\theta}(\mathbf{s}_t)}[c(\mathbf{s}_t, \mathbf{a}_t)].$$

Proposition 2.2. *Assume that* $\pi_\theta(\mathbf{a} \neq \pi^*(\mathbf{s}) \mid \mathbf{s}) \leq \varepsilon$ *for* $\mathbf{s} \sim p_{train}(\mathbf{s})$. *Then*

$$\max_\theta \sum_{t=1}^{T} \mathbb{E}_{p_\theta(\mathbf{s}_t)}[c_t] \in \Theta(\varepsilon T^2).$$

The equality case occurs for, say a tightrope walker, where any mistake immediately results in failure of the task. In reality, we can often recover from mistakes, so the analysis above may be a bit pessimistic. But it's still not good—we want the worst-case cost to scale linearly.

Addressing the problem in practice

- Collecting and augmenting data [often works well but requires domain-specific expertise]
  - Clever data collection
    - Example: for drone-flying through nature, strap some cameras to a person and have them walk, assuming that their head movements match the "correct" drone trajectory
  - Intentionally add mistakes and corrections
  - Add some fake data that illustrates corrections (e.g. side-facing cameras)
- Using more powerful models
  - Non-Markovian behavior: RL produces $\pi_\theta(\mathbf{a}_t \mid \mathbf{o}_t)$, so receiving the same observation will result in the same policy. Humans make decisions based on $o_1, \ldots, o_t$. However, we can use a sequence model (e.g. transformer, LSTM) to observe the whole history.
    - Can work poorly due to **causal confusion**: for instance, pedestrians may cause you to break. The brake indicator lights up when braking. So the model may think that it is good to brake when the indicator is on, not when there is a pedestrian.
  - Multimodal behavior: you want to navigate around a tree, and can go either left or right. They average out to the middle.
    - Expressive continuous distributions
      - Mixture of Gaussians: take $\pi(\mathbf{a} \mid \mathbf{o}) = \sum_i w_i \mathcal{N}(\mu_i, \Sigma_i)$, and train to compute $w_1, \mu_1, \Sigma_1, \ldots, w_N, \mu_N, \Sigma_N$.
      - Latent variable models: you don't know what N is for mixed Gaussians. Provide a "seed" $\xi \sim \mathcal{N}(0, \mathbf{I})$. The most widely used type of model of this sort is the conditional variational autoencoder.

  - Diffusion models: denote the training image as $p_0(\mathbf{x}_0)$. Let noise $= f(\mathbf{x}_i) = \mathbf{x}_{i-1} - \mathbf{x}_i$. Learn $f(\mathbf{x}_i)$.

    This is generalizable to RL. Let $\mathbf{a}_{t,0}$ be the true action, and let noise $= f(\mathbf{s}_t, \mathbf{a}_{t,i}) := \mathbf{a}_{t,i+1} - \mathbf{a}_{t,i}$, and learn f.
- Discretization: discretize one dimension at a time.
  - Autoregressive discretization: use LSTM or transformer.

Goal-conditioned behavioral cloning

In some cases, it is easier to train for multiple classes than it is for one. The idea of goal-conditioned behavioral cloning is that any trajectory will take you to some state, and we can treat that trajectory as "good" for reaching that state. Naively, training to reach a particular goal can work, but you get much less training data because much fewer trajectories end up at the desired state(s). Formally, suppose you have several demonstrations indexed by i,

$$\left\{\mathbf{s}_1^i, \mathbf{a}_1^i, \ldots, \mathbf{s}_{T-1}^i, \mathbf{a}_{T-1}^i, \mathbf{s}_T^i\right\}.$$

Then the objective is to maximize $\log \pi_\theta(\mathbf{a}_t^i \mid \mathbf{s}_t^i, \mathbf{g} = \mathbf{s}_T^i)$.

Exercise 2.3. There are two sources of distributional shift. The first is the same as before, where the distributions of states are distinct in the training and true trajectories. What is the second?

Example 2.4. Lynch et al. [23] taught a robot how to play with objects at a desk. This is pretty hard because any state-action pair could be pretty reasonable. Goal-conditioned training allowed it to reach specific goals.

Example 2.5. It is possible to go beyond just imitation, e.g. Ghosh et al. [9]. The following procedure iterates supervised learning and benefits from its own experience:

- Start with a random policy
- Collect data with random goals
- Treat this data as "demonstrations" for the goals that were reached
- Improve the policy
- Repeat

Example 2.6. It is possible to use goal-conditioning at scale, i.e. generalizing to new but similar tasks. For instance, Shah et al. [33] enables a variety of robots to navigate due to the use of training data spanning a large range of vehicles.

DAgger

Let's go about algorithmically improving behavioral cloning. Recall that the distributional shift issue is that $p_{\pi_\theta}(\mathbf{o}_t) \neq p_{\text{data}}(\mathbf{o}_t)$. So our idea is to collect training data from $p_{\pi_\theta}(\mathbf{o}_t)$ directly by just running $\pi_\theta(\mathbf{a}_t \mid \mathbf{o}_t)$.

Eventually, this will converge such that the distribution of observations in the dataset will approach the distribution of observations that the policy sees when it runs. The labeled actions might be suboptimal, but eventually the dataset is dominated by samples from the correct p_θ.

Algorithm 1 **D**ataset **Agg(er)**gation

1: **for** $i \leftarrow 1$ to K **do**
2: Train $\pi_\theta(\mathbf{a}_t \mid \mathbf{o}_t)$ from human data $\mathcal{D} = \{\mathbf{o}_1, \mathbf{a}_1, \ldots, \mathbf{o}_N, \mathbf{a}_N\}$
3: Run $\pi_\theta(\mathbf{a}_t \mid \mathbf{o}_t)$ to get dataset $\mathcal{D}_\pi = \{\mathbf{o}_1, \ldots, \mathbf{o}_M\}$
4: Ask human to label $\mathcal{D}_\pi$ with actions $\mathbf{a}_t$
5: Aggregate: $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}_\pi$
6: **end for**

A shortcoming of DAgger is that line 4 isn't what you want an agent do in real life: a car will not make decisions instantaneously at every timestep, it should have reaction time and other more natural characteristics. There exist some improvements on this aspect of DAgger.

Proposition 2.7. *Given the same conditions as Proposition 2.2, the cost scales linearly with T.*

What's wrong with imitation learning?

- Humans need to provide data.
- Humans aren't good at providing some kinds of actions: if you want to make a robotic spider to walk, domain-specific knowledge is required.
- Theoretically, RL should be able to outperform humans. (That's the whole point!)

3 PyTorch Tutorial (08/30)

The notes are here.

Tensor operations

- Indexing (what do `...`, `-1`, and `None` mean)
- `squeeze`, `permute`, `view`

Note. Be careful with `view`! Only do one dimension at a time, and use `assert` before and after.

Utils

```
device = None

def init_gpu(use_gpu=True, gpu_id=0):
    global device
    if torch.cuda.is_available() and use_gpu:
        device = torch.device("cuda:" + str(gpu_id))
        print("Using GPU id {}".format(gpu_id))
    else:
        device = torch.device("cpu")
        print("GPU not detected. Defaulting to CPU.")

def set_device(gpu_id):
    torch.cuda.set_device(gpu_id)

```

```
def from_numpy(*args, **kwargs):
    return torch.from_numpy(*args, **kwargs).float().to(device)

def to_numpy(tensor):
    return tensor.to('cpu').detach().numpy()
```

Custom networks

```
import torch.nn as nn

class SingleLayer(nn.Module):
    def __init__(self, in_dim, out_dim, hidden_dim):
        super().__init__()
        self.net = nn.Sequential(
            nn.Module(in_dim, hidden_dim),
            nn.ReLu(),
            nn.Module(hidden_dim, out_dim)
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor():
        return self.net(x)

batch_size = 256
net = SingleLayer(2, 32, 1).to(device)
output = net(torch.randn(size=(batch_size, 2)).to(device))
```

Training loop

```
net = (...).to(device)
dataset = ...
dataloader = ...
optimizer = ...
loss_fn = ...
for epoch in range(num_epochs):
    net.train()
    for data, target in dataloader:
        data = from_numpy(data)
        target = from_numpy(target)

        pred = net(data)
        loss = loss_fn(pred, target)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    net.eval()
    # do evaluation
```

4 Introduction to Reinforcement Learning (09/04)

Imitation learning doesn't perform well in practice because we don't always have expert data, which is often hard to collect or requires domain-specific expertise. Thus, we want a more generalizable framework.

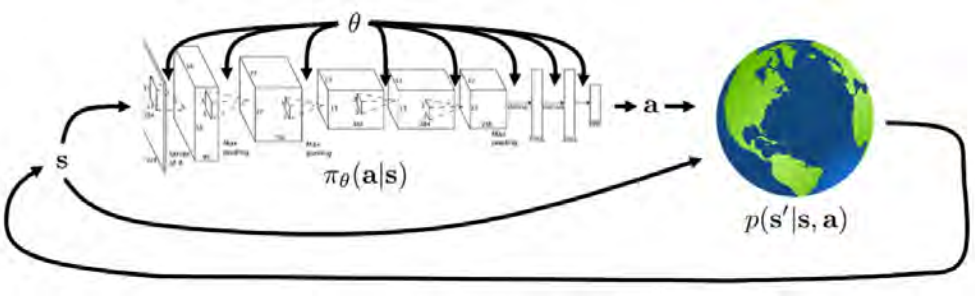


Figure 2: Reinforcement learning

Notation and Terminology

Review last lecture for states, actions, etc. $r(\mathbf{s}, \mathbf{a})$ is a **reward function**.

Definition 4.1. A **Markov chain** $\mathcal{M} = \{\mathcal{S}, \mathcal{T}\}$ is a state space and transition operator. In particular, if we $\mathcal{T}$ as a matrix, $T_{i,j} := p(\mathbf{s}_{t+1} = i \mid \mathbf{s}_t = j)$, and $\boldsymbol{\mu}_t$ is a vector of probabilities, then $\boldsymbol{\mu}_{t+1} = \mathcal{T}\boldsymbol{\mu}_t$.

Definition 4.2. A **partially observed Markov decision process** $\mathcal{M} = \{\mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{T}, \mathcal{E}, r\}$ has:

- $\mathcal{S}$ = state space
- $\mathcal{A}$ = action space
- $\mathcal{O}$ = observation space
- $\mathcal{T}$ = transition operator
- $\mathcal{E}$ = emission probability $p(o_t \mid s_t)$
- r = reward function; $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$

Concept 4.3. The reinforcement learning "pipeline" is shown in Figure 2. Typically, a policy $\pi_\theta(\cdot \mid \mathbf{s})$ is a neural net accepting $\mathbf{s}$ as input with parameters given by θ, and probabilistically produces an action $\mathbf{a}$. Then the next state $\mathbf{s}' \sim p(\cdot \mid \mathbf{s}, \mathbf{a})$.

Thus, for trajectory $\tau = (\mathbf{s}_1, \mathbf{a}_1, \ldots, \mathbf{s}_T, \mathbf{a}_T)$, we have

$$p_\theta(\tau) = p(\mathbf{s}_1) \prod_{t=1}^{T} \pi_\theta(\mathbf{a}_t \mid \mathbf{s}_t) p(\mathbf{s}_{t+1} \mid \mathbf{s}_t, \mathbf{a}_t). \tag{1}$$

Our objective is to compute

$$\theta^* = \arg\max_\theta \mathbb{E}_{\tau \sim p_\theta} \left[\sum_{t=1}^{T} r(\mathbf{s}_t, \mathbf{a}_t) \right]. \tag{2}$$

Observe that in (1), the term in the product can be rewritten as $p((\mathbf{s}_{t+1}, \mathbf{a}_{t+1}) \mid (\mathbf{s}_t, \mathbf{a}_t))$, i.e. model this as a Markov chain with states given by $(\mathbf{s}_t, \mathbf{a}_t)$. In particular, by linearity of expectation, we can rewrite the objective (2)

$$\theta^* = \arg\max_\theta \sum_{t=1}^{T} \mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t) \sim p_\theta} [r(\mathbf{s}_t, \mathbf{a}_t)].$$

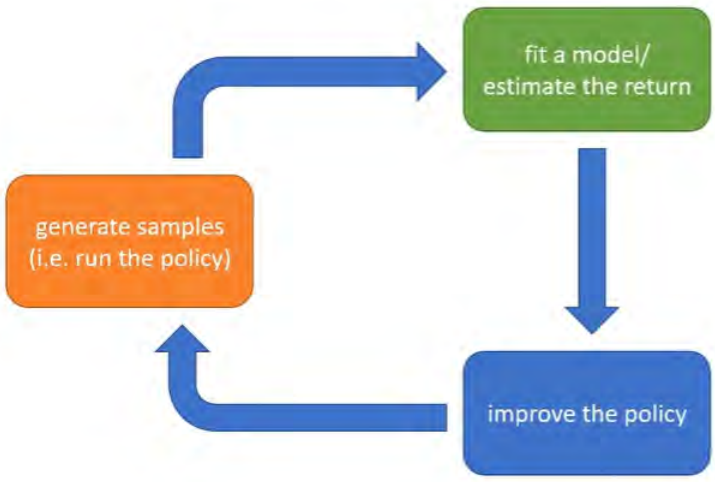


Figure 3: The anatomy of your typical RL algorithm.

For $T = \infty$, the idea is that there exists a stationary distribution $\mu = p_\theta(\mathbf{s}, \mathbf{a})$, so setting

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

suffices.

Example 4.4. The nice thing about this setup is that this works even when r is not smooth. For instance, suppose you are driving a car, and at each timestep get a reward of $+1$ if you stay on the road and -1 if you drive off. Then $\pi_\theta(\mathbf{a} = \text{drive off}) = \theta$ yields $\mathbb{E}_{\pi_\theta}[r]$ smooth in θ, despite r itself not being smooth.

Reinforcement Learning Algorithms

Virtually all RL algorithms will look like Figure 3. Throughout these notes, we'll refer to the "green box" and "blue box" in the figure. One example for the green box is learning f_φ such that $\mathbf{s}_{t+1} \approx f_\varphi(\mathbf{s}_t, \mathbf{a}_t)$ in a neural net, and a corresponding blue box is running backprop through f_φ and r, and training $\pi_\theta(\mathbf{s}_t) = \mathbf{a}_t$.

Computational cost can vary significantly based on the case you're working with:

- If you're using a real robot, generating samples will take much longer than in a MuJoCo simulator.
- Learning a model for $\mathbf{s}_t \approx f_\varphi(\mathbf{s}_t, \mathbf{a}_t)$ is much more expensive than estimating the return as the expected sum of rewards.
- In the last bullet, training π_θ in the former is much more costly than running gradient descent on the latter.

Value Functions

Definition 4.5. $Q^\pi(\mathbf{s}_t, \mathbf{a}_t) = \sum_{t'=t}^{T} \mathbb{E}_{\pi_\theta}[r(\mathbf{s}_{t'}, \mathbf{a}_{t'} \mid \mathbf{s}_t, \mathbf{a}_t]$ is the total reward from taking $\mathbf{a}_t$ at state $\mathbf{s}_t$.

Definition 4.6. $V^\pi(\mathbf{s}_t) = \sum_{t'=t}^{T} \mathbb{E}_{\pi_\theta}[r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) \mid \mathbf{s}_t] = \mathbb{E}_{\mathbf{a}_t \sim \pi(\cdot|\mathbf{s}_t)}[Q^\pi(\mathbf{s}_t, \mathbf{a}_t)]$ is the total reward from $\mathbf{s}_t$.

Idea 4.7. $\mathbb{E}_{\mathbf{s}_1 \sim p}[V^\pi(\mathbf{s}_1)]$ is the RL objective.

Typically, reinforcement learning algorithms based on Q-functions will take one of two forms:

- If you have a policy π and know $Q^\pi(\mathbf{s}, \mathbf{a})$, then we can improve π by setting $\pi'(\mathbf{a}^* \mid \mathbf{s}) = 1$ if $\mathbf{a}^* = \arg\max_{\mathbf{a}} Q^\pi(\mathbf{s}, \mathbf{a})$.
- If $Q^\pi(\mathbf{s}, \mathbf{a}) > V^\pi(\mathbf{s})$, then $\mathbf{a}$ is better than average. So we take increase its probability, $\pi'(\mathbf{a} \mid \mathbf{s}) > \pi(\mathbf{a} \mid \mathbf{s})$.

Types of RL algorithms

- **Policy gradients**: directly differentiate the objective $\pi^* = \arg\max_\theta \mathbb{E}_{\tau \sim p_\theta}\left[\sum_t r(\mathbf{s}_t, \mathbf{a}_t)\right]$
  - Green: evaluate returns $R_\tau = \sum_t r(\mathbf{s}_t, \mathbf{a}_t)$
  - Blue: $\theta \leftarrow \theta + \alpha \nabla_\theta \mathbb{E}_{\tau \sim \pi_\theta}[R_\tau]$
  - *Examples: Q-learning Watkins [39], DQN Mnih et al. [25], temporal difference learning Sutton [34], fitted value iteration*
- **Value-based**: estimate value- or Q-function of the optimal policy
  - Green: fit $V(\mathbf{s})$ or $Q(\mathbf{s}, \mathbf{a})$
  - Blue: set $\pi(\mathbf{s})$ based on a method outlined in the previous section
  - *Examples: REINFORCE Sutton et al. [36], natural policy gradient, TRPO Schulman et al. [31]*
- **Actor-critic**: estimate value- or Q-function of the current policy, use it to improve policy
  - Green: fit $V(\mathbf{s})$ or $Q(\mathbf{s}, \mathbf{a})$
  - Blue: $\theta \leftarrow \theta + \alpha \nabla_\theta \mathbb{E}[Q(\mathbf{s}_t, \mathbf{a}_t)]$
  - *Examples: A3C Mnih et al. [26], SAC Haarnoja et al. [12]*
- **Model-based**: estimate the transition model, then:
  - Use it for planning: use backpropagation to optimize over actions, or discrete planning in discrete action spaces
  - Backpropagate gradients into the policy
  - Use the model to learn a value function (dynamic programming!)
  - Something else
  - *Examples: Dyna Sutton [35], guided policy search Levine and Koltun [21]*

Tradeoffs between RL algorithms

- Tradeoffs
  - **Sample efficiency** is the number of samples we need to get a good policy. The most important question is whether the algorithm is **off-policy** (can improve policy without generating new samples from that policy) or **on-policy** (need to generate new samples each time).

  - Stability and ease of use: stuff isn't guaranteed to converge, especially since much of RL doesn't rely on gradient descent.
- Assumptions: full observability, stochastic vs. deterministic, continuous vs. discrete, episodic vs. infinite horizon
- Easier to represent the policy or the model?

5 Policy Gradients (09/06)

The videos begin here.

REINFORCE

This section is based on Sutton et al. [36].

Recall from last time that the RL objective is as follows: for trajectory $\tau = (\mathbf{s}_1, \mathbf{a}_1, \ldots, \mathbf{s}_T, \mathbf{a}_T)$, we have

$$p_\theta(\tau) = p(\mathbf{s}_1) \prod_{t=1}^{T} \pi_\theta(\mathbf{a}_t \mid \mathbf{s}_t) p(\mathbf{s}_{t+1} \mid \mathbf{s}_t, \mathbf{a}_t).$$

Our objective is to compute

$$\theta^* = \arg\max_\theta \underbrace{\mathbb{E}_{\tau \sim p_\theta} \left[\underbrace{\sum_{t=1}^{T} r(\mathbf{s}_t, \mathbf{a}_t)}_{r(\tau)} \right]}_{J(\theta)} \overset{\text{finite horizon}}{=} \arg\max_\theta \sum_{t=1}^{T} \mathbb{E}_{\mathbf{s}_t, \mathbf{a}_t \sim p_\theta} [r(\mathbf{s}_t, \mathbf{a}_t)]$$

To evaluate the objective, we just rollout N times:

$$J(\theta) \approx \frac{1}{N} \sum_i \sum_t r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}).$$

To improve the objective, we want to differentiate it. However, we don't know the true values of p (initial state distribution and transitions in the real world), so we do some algebra:

$$\begin{aligned}
\nabla_\theta J(\theta) &= \mathbb{E}_{\tau \sim p_\theta}[r(\tau)] \\
&= \int r(\tau) \nabla_\theta p_\theta(\tau) \mathrm{d}\tau \\
&= \int r(\tau) p_\theta(\tau) \nabla_\theta \log p_\theta(\tau) \mathrm{d}\tau \\
&= \mathbb{E}_{\tau \sim p_\theta}[r(\tau) \nabla_\theta \log p_\theta(\tau)].
\end{aligned}$$

Observe that

$$\begin{aligned}
\nabla_\theta \log p_\theta(\tau) &= \nabla_\theta \left[\log p(\mathbf{s}_1) + \sum_{t=1}^{T} \left(\log \pi_\theta(\mathbf{a}_t \mid \mathbf{s}_t) + \log p(\mathbf{s}_{t+1} \mid \mathbf{s}_t, \mathbf{a}_t) \right) \right] \\
&= \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\mathbf{a}_t \mid \mathbf{s}_t),
\end{aligned}$$

so continuing the original chain,

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim p_\theta}\left[\left(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(\mathbf{a}_t \mid \mathbf{s}_t)\right)\left(\sum_{t=1}^{T} r(\mathbf{s}_t, \mathbf{a}_t)\right)\right].$$

To evaluate the policy gradient, we just need to run N of them and take the average, then apply gradient descent.

Intuitively, observe that actions associated with high rewards get their log probabilities increased, and vice-versa. So you can think of it as a weighted maximum likelihood, or trial-and-error.

Remark. We didn't use the Markov property in our derivation, so you can use policy gradient in partially observed MDPs without modification; simply replace **s** with **o**.

Unfortunately, vanilla policy gradient doesn't work too well in practice: if you take small N, then your rewards will have high variance, and changes to the policy can be unpredictable or undesirable.

Reducing variance

Causality

Above, we found

$$\begin{aligned}\nabla_\theta J(\theta) &\approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t} \mid \mathbf{s}_{i,t})\right)\left(\sum_{t=1}^{T} r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t})\right)\\ &= \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t} \mid \mathbf{s}_{i,t})\left(\sum_{t'=1}^{T} r(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'})\right).\end{aligned}$$

Since the policy at time t' cannot affect rewards at times $t < t'$, in expectation, we find

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t} \mid \mathbf{s}_{i,t})\underbrace{\left(\sum_{t'=t}^{T} r(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'})\right)}_{\hat{Q}_{i,t}=\text{“reward to go”}}.$$

I don't real get this

Normalization

If our rewards are always positive, the more positive ones will just be increased more than the less positive ones. Let's say we want to make them zero-mean by subtracting $b = \frac{1}{N}\sum_{i=1}^{N} r(\tau)$ from $r(\tau)$ in the objective. We claim that in fact this is the same objective:

$$\mathbb{E}[\nabla_\theta \log p_\theta(\tau) b] = \int p_\theta(\tau)\nabla_\theta \log p_\theta(\tau) b \mathrm{d}\tau = \int \nabla_\theta p_\theta(\tau) b \mathrm{d}\tau = b\nabla_\theta \int p_\theta(\tau)\mathrm{d}\tau = b\nabla_\theta 1 = 0.$$

Note that any value of b works; is our selected value of b actually good? Let's write out the variance explicitly.

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

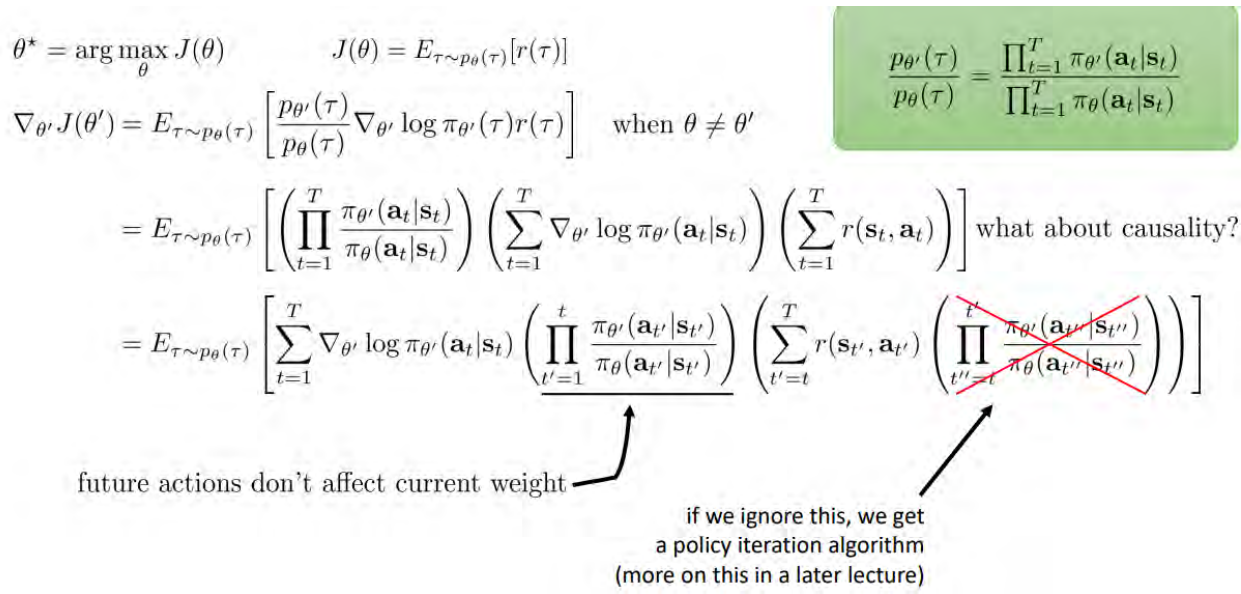


Figure 4: Off-policy Policy Gradient

We showed just now that the second term does not depend on b. So we only care about optimizing the first one.

$$\begin{aligned} 0 &= \frac{\mathrm{d}\,\mathrm{Var}\,\nabla_\theta J(\theta)}{\mathrm{d}b} \\ &= -2\mathbb{E}[(\nabla_\theta \log p_\theta(\tau))^2 r(\tau)] + 2b\mathbb{E}[(\nabla_\theta \log p_\theta(\tau))^2] \\ \implies b^* &= \frac{\mathbb{E}[(\nabla_\theta \log p_\theta(\tau))^2 r(\tau)]}{\mathbb{E}[(\nabla_\theta \log p_\theta(\tau))^2]}. \end{aligned}$$

Off-policy policy gradients

Policy gradient is an on-policy algorithm: recall that we approximate $\mathbb{E}_{\tau\sim p_\theta}$ by sampling trajectories, and this changes each time we update our θ.

It is, however, possible to alter this toward an off-policy setting. Suppose we don't have samples from $p_\theta(\tau)$, but rather $\overline{p}(\tau)$. We proceed by **importance sampling**:

$$J(\theta) = \mathbb{E}_{\tau\sim p_\theta}[r(\tau)] = \int p_\theta(\tau) r(\tau)\mathrm{d}\tau = \int \overline{p}(\tau)\frac{p_\theta(\tau)}{\overline{p}(\tau)} r(\tau)\mathrm{d}\tau = \mathbb{E}_{\tau\sim\overline{p}}\left[\frac{p_\theta(\tau)}{\overline{p}(\tau)} r(\tau)\right].$$

In fact, we can express the ratio in a convenient way:

$$\frac{p_\theta(\tau)}{\overline{p}(\tau)} = \frac{p(\mathbf{s}_1)\prod_{t=1}^T \pi_\theta(\mathbf{a}_t \mid \mathbf{s}_t) p(\mathbf{s}_{t+1} \mid \mathbf{s}_t, \mathbf{a}_t)}{p(\mathbf{s}_1)\prod_{t=1}^T \overline{\pi}(\mathbf{a}_t \mid \mathbf{s}_t) p(\mathbf{s}_{t+1} \mid \mathbf{s}_t, \mathbf{a}_t)} = \frac{\prod_{t=1}^T \pi_\theta(\mathbf{a}_t \mid \mathbf{s}_t)}{\prod_{t=1}^T \overline{\pi}(\mathbf{a}_t \mid \mathbf{s}_t)}.$$

Of course, this works well because we have access to π's.

The gradient calculation is similar; one can follow this computation. See Figure 4.

Implementing policy gradient

In practice, neural nets have many parameters, so we want to set up a graph to run automatic differentiation. We do so with

$$\tilde{J}(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\log \pi_\theta(\mathbf{a}_{i,t} \mid \mathbf{s}_{i,t})\hat{Q}_{i,t}.$$

Warning 5.1.

- Variance is high, so gradients are noisy.
- Use large batches (on the scale of 10^3 or 10^4).
- There are policy gradient–specific learning rates. Adam is only ok.

Advanced policy gradients

Some parameters impact the probabilities much more than others in the gradient descent step, so we want them to be weighed accordingly. It's hard to choose a learning rate that is compatible with every parameter.

We can frame gradient descent as the following optimization problem:

$$\theta \leftarrow \arg\max_{\theta'}(\theta' - \theta)^\top \nabla_\theta J(\theta) \text{ subject to } \|\theta' - \theta\|^2 \le \varepsilon,$$

where ε is a proxy for the learning rate. But we don't really care about parameter space; rather, we want to limit "step size of the policy." Thus, we instead condition

$$\theta \leftarrow \arg\max_{\theta'}(\theta' - \theta)^\top \nabla_\theta J(\theta) \text{ subject to } D(\pi_{\theta'}, \pi_\theta) \le \varepsilon,$$

where D is a parameterization-independent measure of "divergence" between distributions. The usual choice is **Kullback–Leibler divergence**

$$D_{KL}(\pi_{\theta'} \parallel \pi_\theta) = \mathbb{E}_{\pi_{\theta'}}[\log \pi_\theta - \log \pi_{\theta'}].$$

Through Taylor expansion, one can show that

$$D_{KL}(\pi_{\theta'} \parallel \pi_\theta) \approx (\theta' - \theta)^\top \underbrace{\mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(\mathbf{a} \mid \mathbf{s}) \nabla_\theta \log \pi_\theta(\mathbf{a} \mid \mathbf{s})^\top]}_{\mathbf{F}=\text{Fisher information matrix}}(\theta' - \theta).$$

This has a nice interpretation: now we can treat norms wrt $\mathbf{F}$ Schulman et al. [31], or our update looks like $\theta \leftarrow \theta + \alpha \mathbf{F}^{-1}\nabla_\theta J(\theta)$ Kakade [16].

6 Actor-Critic Algorithms (09/11)

Recall that for policy gradient, we found

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t} \mid \mathbf{s}_{i,t}) \underbrace{\sum_{t'=t}^{T} r(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'})}_{\hat{Q}_{i,t}=\text{"reward to go"}}.$$

However, $\hat{Q}_{i,t}$ can be quite a bad estimate, since we only get one trajectory: really we want to approximate the true expected reward to go, $Q(\mathbf{s}_t, \mathbf{a}_t) = \sum_{t'=t}^{T} \mathbb{E}_{\pi_\theta}[r(\mathbf{s}_{t'}, \mathbf{a}_{t'} \mid \mathbf{s}_t, \mathbf{a}_t)]$.

Recall that we want to subtract a baseline to our Q-values to reduce variance. This baseline is $V(\mathbf{s}_t) = \mathbb{E}_{\mathbf{a}_t \sim \pi_\theta(\cdot|\mathbf{s}_t)} Q(\mathbf{s}_t, \mathbf{a}_t)$. So our update looks like

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t} \mid \mathbf{s}_{i,t}) A(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}),$$

where A denotes the **advantage** of taking a given action over the average.

Now, we think back to Figure 3. The green box fits to either Q^π, V^π, or A^π. Which one should we use? Let's do some math:

$$\begin{aligned}
Q^\pi(\mathbf{s}_t, \mathbf{a}_t) &= r(\mathbf{s}_t, \mathbf{a}_t) + \sum_{t'=t+1}^{T} \mathbb{E}_{\pi_\theta}[r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) \mid \mathbf{s}_t, \mathbf{a}_t] \\
&= r(\mathbf{s}_t, \mathbf{a}_t) + \mathbb{E}_{\mathbf{s}_{t+1} \sim \pi_\theta(\cdot|\mathbf{s}_t, \mathbf{a}_t)} V^\pi(\mathbf{s}_{t+1}) \\
&\approx r(\mathbf{s}_t, \mathbf{a}_t) + V^\pi(\mathbf{s}_{t+1}), \\
A^\pi(\mathbf{s}_t, \mathbf{a}_t) &= Q^\pi(\mathbf{s}_t, \mathbf{a}_t) - V^\pi(\mathbf{s}_t) \\
&\approx r(\mathbf{s}_t, \mathbf{a}_t) + V^\pi(\mathbf{s}_{t+1}) - V^\pi(\mathbf{s}_t).
\end{aligned}$$

So it's probably good to learn V^π, since it allows us to estimate the other two and is only dependent on the state.

We can use Monte Carlo evaluation with function approximation, idea being that states and be treated continuously. We are given some training data $\left\{ \left(\mathbf{s}_{i,t}, \underbrace{\sum_{t'=t}^{T} r(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'})}_{y_{i,t}} \right) \right\}$. Our goal is to solve the supervised regression problem with loss $\mathcal{L}(\varphi) = \frac{1}{2} \sum_i \left\| \hat{V}_\varphi^\pi(\mathbf{s}_i) - y_i \right\|^2$. Observe that $y_{i,t} \approx r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) + V^\pi(\mathbf{s}_{i,t+1}) \approx r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) + \hat{V}_\varphi^\pi(\mathbf{s}_{i,t+1})$, a **bootstrapped estimate**. It increases bias but reduces variance.

Remark. If $T = \infty$, then $\hat{V}_\varphi^\pi$ can grow infinitely large. A simple trick is to use a discount factor $\gamma \in [0, 1]$ (usually in $[0.9, 0.999]$), so that $y_{i,t} = r(s_{i,t}, a_{i,t}) + \gamma \hat{V}_\varphi^\pi(s_{i,t+1})$. One can think of this as the addition of a "death state" to the MDP, to which every state has probability $1 - \gamma$ of transitioning. In this case, our update is

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t} \mid \mathbf{s}_{i,t}) \sum_{t'=t}^{T} \gamma^{t'-t} r(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'}).$$

From evaluation to actor-critic

The textbook algorithm is as follows. In practice, we need to make some changes because they don't work well for deep RL.

Design decisions

- **Architecture:** How should we learn $\hat{V}_\varphi^\pi(s)$ and $\pi_\theta(a \mid s)$? We can either use a two network or shared network design.

Algorithm 2 Batch Actor-Critic

1: **for** $t \leftarrow 1$ to T **do**
2: sample trajectory $\tau \sim \pi_\theta$
3: fit $\hat{V}^\pi_\varphi(s)$ to sampled reward sums
4: evaluate $\hat{A}^\pi(s_i, a_i) = r(s_i, a_i) + \gamma \hat{V}^\pi_\varphi(s'_i) - \hat{V}^\pi_\varphi(s_i)$
5: $\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_i \nabla_\theta \log \pi_\theta(a_i \mid s_i) \hat{A}^\pi(s_i, a_i)$
6: $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$
7: **end for**

- **Implementation:** Typically, line 3 in online actor-critic works best with a batch, i.e. parallel workers. So we must ask whether it is best to use synchronous or asynchronous parallelization (do we update θ after all threads are done, or is it fine to trust that updates to θ are small enough that it doesn't make a difference to other threads?). In practice, we usually use the latter approach because performance gains outweigh slight biases.
- **Removing the on-policy assumption:** Add a replay buffer that stores transitions that we saw in previous time steps.

Algorithm 3 Online Actor-Critic

1: **while** loop as long as you want **do**
2: take action $a \sim \pi_\theta(\cdot \mid s)$, get (s, a, s', r), and store in $\mathcal{R}$
3: sample a batch $\{s_i, a_i, r_i, s'_i\}$ from $\mathcal{R}$
4: update $\hat{Q}^\pi_\varphi$ using targets $y_i = r_i + \gamma \hat{Q}^\pi_\varphi(s'_i, a'_i)$
5: $\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_i \nabla_\theta \log \pi_\theta(a^\pi_i \mid s_i) \hat{A}^\pi(s_i, a^\pi_i)$ where $a^\pi_i \sim \pi_\theta(\cdot \mid s_i)$
6: $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$
7: **end while**

In particular, there are several key differences between Algorithms 2 and 3. In line 4, we are learning Q instead of V. The issue is that the value function would take the expectation over actions distributed according to our old policy, not π_θ. So we remedy this by using Q, which works fine for any state-action pair, and sampling $a'_i \sim \pi_\theta(\cdot \mid s'_i)$. The same principle holds for line 5.

Remark. In line 5, it's fine to use $\hat{Q}^\pi$ instead of $\hat{A}^\pi$, despite this having higher variance. Since we don't need to interact with a simulator, we can just sample more actions without generating more states.

Critics as baselines

Actor-critic works well because it has lower variance but can be biased (who knows if $\hat{Q}$ is any good?). Policy gradient is unbiased in expectation but has higher variance (you only collect one sample). So the idea is to use $\hat{V}^\pi_\varphi$ as the baseline, with the intention that the estimator remains unbiased and has only slightly higher variance than actor-critic.

If our critic is good, then we can use the following unbiased estimator, provided that the second

term is computable:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{i,t} \mid s_{i,t})(\hat{Q}_{i,t} - Q^\pi_\varphi(s_{i,t}, a_{i,t})) + \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \mathbb{E}_{a_t\sim\pi_\theta(\cdot|s_{i,t})} Q^\pi_\varphi(s_{i,t}, a_t)$$

In practice, we usually want to trade off bias and variance. In fact, there is a controlled way to do so with ***n*-step returns**. The idea is that the critic

$$\hat{A}^\pi_{\mathrm{C}}(s_t, a_t) = r(s_t, a_t) + \gamma \hat{V}^\pi_\varphi(s_{t+1}) - \hat{V}^\pi_\varphi(s_t)$$

can have high bias but low variance, while the Monte Carlo estimate

$$\hat{A}^\pi_{\mathrm{MC}}(s_t, a_t) = \sum_{t'=t}^{\infty}\gamma^{t'-t} r(s_{t'}, a_{t'}) - \hat{V}^\pi_\varphi(s_t)$$

has no bias and high variance. So we can use Monte Carlo for n steps, cut it off, then use the critic:

$$\hat{A}^\pi_n(s_t, a_t) = \sum_{t'=t}^{t+n}\gamma^{t'-t} r(s_{t'}, a_{t'}) - \hat{V}^\pi_\varphi(s_t) + \gamma^n \hat{V}^\pi_\varphi(s_{t+n}).$$

We can recognize this as a generalization of our previous approach with $n = 1$; in practice, $n > 1$ often works better.

This approach can be generalized further (called **generalized advantage estimation (GAE)**) Schulman et al. [32]. The idea is that we don't have to just choose one n; rather, we can cut everywhere at once. Now we choose weights and take

$$\hat{A}^\pi_{\mathrm{GAE}}(s_t, a_t) := \sum_{n=1}^{\infty} w_n \hat{A}^\pi_n(s_t, a_t).$$

We still prefer cutting earlier, so we want the weights to decrease. In practice, $w_i \propto \lambda^{i-1}$ is a pretty good choice. This essentially contributes to our discount factor; one can show that

$$\hat{A}^\pi_{\mathrm{GAE}}(s_t, a_t) = \sum_{t'=t}^{\infty}(\gamma\lambda)^{t'-t}\left(r(s_t, a_t) + \gamma \hat{V}^\pi_\varphi(s_{t'+1}) - \hat{V}^\pi_\varphi(s_{t'})\right).$$

As such, we can view discounts as variance reduction.

7 Value Function Methods (09/13)

The videos begin here.

Q-iteration

Question 7.1. Can we omit policy gradient? We learn a value function, so maybe we can just use it directly.

Answer. Yes: just take the best action $\arg\max_{a_t} A^\pi(s_t, a_t)$, then follow π. Implicitly, we define a policy

$$\pi'(a_t \mid s_t) = \mathbb{1}\left\{a_t = \arg\max_{a_t} A^\pi(s_t, a_t)\right\}$$

It is at least as good as π, and probably better.

So it remains to evaluate $A^\pi(s_t, a_t)$, which we can do by learning the value function. We use the bootstrapped estimate

$$\begin{aligned} V^\pi(s) &\leftarrow \mathbb{E}_{a\sim\pi(\cdot|s)}[r(s,a) + \gamma \mathbb{E}_{s'\sim p(\cdot|s,a)}[V^\pi(s')]] \\ &= r(s,\pi(s)) + \gamma \mathbb{E}_{s'\sim p(\cdot|s,\pi(s))}[V^\pi(s')], \end{aligned}$$

where the second line follows from π being deterministic.

Alternatively, we can simplify this by avoiding representing the policy altogether: it is equivalent to iterate as

Algorithm 4 Value Iteration (Small Discrete Case)

1: **for** run K times **do**
2: $\quad Q^\pi(s,a) \leftarrow r(s,a) + \gamma \mathbb{E}_{s'\sim p(\cdot|s,a)} V^\pi(s')$
3: $\quad V^\pi(s) = \max_a Q^\pi(s,a)$
4: **end for**

This works great for small examples, say if you're on a 4×4 grid and can only move one of 4 ways, since we're just storing our value function as a table and updating it with a loop. But in practice, this doesn't work; even if you have 200×200 images as states, you'll have $|\mathcal{S}| = 256^{200\times200\times3}$ (more than the number of atoms in the universe). So we'll use a neural net function approximator $V : \mathcal{S} \to \mathbb{R}$. Our goal is to minimize $L(\varphi) = \frac{1}{2}\left\|V_\varphi(s) - \max_a Q^\pi(s,a)\right\|^2$.

Algorithm 5 Fitted Value Iteration

1: **for** run K times **do**
2: $\quad y_i \leftarrow \max_{a_i}\left(r(s_i,a_i) + \gamma \mathbb{E}_{s_i'\sim p(\cdot|s_i,a_i)} V_\varphi(s_i')\right)$
3: $\quad \varphi \leftarrow \arg\min_\varphi \frac{1}{2}\sum_i \left\|V_\varphi(s_i) - y_i\right\|^2$ ▷ gradient descent
4: **end for**

However, this doesn't work in practice: in line 2, we must test multiple actions for each state (which isn't allowed if we can only roll out policies), and we also don't know transition dynamics p. But there is a fairly simple fix, replacing our V function with Q again:

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

Observe that a doesn't appear in the expectation; it is merely conditioned on. As such, we can approximate this by generating $\{(s,a,s')\}$ from the samples we've already collected. (We don't need to simulate new ones! In fact, since we made no assumptions about our sample, this works for off-policy sampling.)

Remark. This isn't just gradient descent, since we can't differentiate through the max.

Algorithm 6 Fitted Q Iteration

1: Collect $\mathcal{D} = \{(s_i, a_i, s_i', r_i)\}$
2: **for** run K times **do**
3: $y_i \leftarrow r(s_i, a_i) + \gamma \mathbb{E}[V_\varphi(s_i')] \approx r(s_i, a_i) + \gamma \max_{a'} Q_\varphi(s_i', a_i')$ ▷ improve the policy
4: $\varphi \leftarrow \arg\min_\varphi \frac{1}{2} \sum_i \left\| Q_\varphi(s_i, a_i) - y_i \right\|^2$ ▷ minimize the error of fit; gradient descent
5: **end for**

Q-learning

In Algorithm 6, we can convert this process to an online process, where instead of sampling from a predetermined dataset, we take some action a_i and observe (s_i, a_i, s_i', r_i), then proceed as before. This is called **online Q-learning**.

Our final policy (ideally) should be $\pi(a_t \mid s_t) = \mathbb{1}\left\{a_t = \arg\max_{a_t} Q_\varphi(s_t, a_t)\right\}$. However, we don't want our initial policy to be deterministic, because it won't explore. A common approach is called **ε-greedy**, i.e.

$$\pi(a_t \mid s_t) = \begin{cases} 1 - \varepsilon & \text{if } a_t = \arg\max_{a_t} Q_\varphi(s_t, a_t) \\ \varepsilon / (|\mathcal{A}| - 1) & \text{else.} \end{cases}$$

It is also common to decrease ε over time: in the beginning, you might want to encourage more exploration, and later on, as the policy converges, you might want behavior to be more stable.

Another approach is **Boltzmann exploration** (softmax), $\pi(a_t \mid s_t) \propto \exp Q_\varphi(s_t, a_t)$. The idea is that two almost equally good actions will be taken almost equally as frequently, while actions you've already discovered as terrible shouldn't be re-explored.

Value function learning theory

We can summarize Algorithm 4 using the **Bellman operator** $\mathcal{B}V := \max_a (r_a + \gamma \mathcal{T}_a V)$, where r_a is a stacked vector of rewards at all states for action a and $\mathcal{T}_a \in \mathbb{R}^{|\mathcal{S}| \times |\mathcal{S}|}$ stores the values of $p(s' \mid s, a)$. The idea is that $V^* = \mathcal{B}V^*$, and moreover:

Claim 7.2. V^* always exists, is unique, and corresponds to the optimal policy.

Proof. First, we claim that $\mathcal{B}$ is a contraction with respect to ℓ_∞:

$$\begin{aligned} \left\| \mathcal{B}^\pi V_1 - \mathcal{B}^\pi V_2 \right\|_\infty &= \left\| R + \gamma P^\pi V_1 - R - \gamma P^\pi V_2 \right\|_\infty \\ &= \gamma \left\| P^\pi (V_1 - V_2) \right\|_\infty \\ &\leq \gamma \left\| V_1 - V_2 \right\|_\infty . \end{aligned}$$

In line 3, since P^π represents a transition matrix, its rows are nonnegative reals summing to 1, and as such its product with any vector must be at most the largest entry in the vector.

Taking $\bar{V} := V^*$ yields the desired result. ■

Remark. An observation that is useless for now but potentially useful for analysis in the future: if for some reason you have small $\mathcal{S}$ and known transition probabilities, it turns out for $\gamma < 1$ that $I - \gamma P^\pi$ is full-rank, and thus $V^\pi = (I - \gamma P^\pi)^{-1} R$.

Now, let us consider the more general case of Algorithm 5. Recall that we update $V' \leftarrow \arg\min_{V' \in \Omega} \frac{1}{2} \sum \|V'(s) - ($ where Ω denotes the space of all value functions represented by our neural net. If we define the orthogonal projection $\Pi V := \arg\min_{V' \in \Omega} \frac{1}{2} \sum \|V'(s) - V(s)\|^2$, then the algorithm is equivalent to looping $V \leftarrow \Pi \mathcal{B} V$. Unfortunately, although Π is a contraction with respect to the ℓ_2 norm, it is not the case that ΠB is a contraction with respect to any norm. As a result, our V' need not converge, and often this happens in practice.

In the case of Algorithm 6, we define $\mathcal{B}$ much in the same way, and following the same steps, find that V does not necessarily converge. The same logic applies to online Q-learning.

8 Deep RL with Q-functions (09/18)

The videos begin here.

Recall that there are some major problems with trying to apply Q-learning directly in deep RL:

- We can't run gradient descent through a max.

  Solution: Only update φ every N steps, so that your target isn't moving most of the time. Each loop can then be regarded as a supervised learning problem.

- s_i' and s_i are correlated, which violates a basic assumption for stochastic gradient methods. Conceptually, the result is that since the target value is always changing, φ will overfit to the most recent states.

  Solution: Instead of online Q iteration, it's fine to use any policy with broad support and to load data (s_i, a_i, s_i', r_i) from a dataset.

A solution to the latter problem is to use a replay buffer: Putting it together, our full pipeline will look like:

Algorithm 7 Q-learning with replay buffer and target network

1: **for** run M times **do**
2: save target network parameters $\varphi' \leftarrow \varphi$
3: **for** run N times **do**
4: collect dataset $\{(s_i, a_i, s_i', r_i)\}$ from some policy, add it to $\mathcal{B}$
5: **for** run K times **do**
6: sample a batch (s_i, a_i, s_i', r_i) from $\mathcal{B}$
7: $\varphi \leftarrow \varphi - \alpha \sum_i \frac{\mathrm{d}Q_\varphi(s_i, a_i)}{\mathrm{d}\varphi}(Q_\varphi(s_i, a_i) - (r(s_i, a_i) + \gamma \max_{a'} Q_{\varphi'}(s_i', a_i')))$
8: **end for**
9: **end for**
10: **end for**

Note that in line 7, we are computing our target using φ'. The classic **deep Q-learning (DQN)** algorithm uses $K = 1$. Usually, N is pretty big, maybe 10^4. It does, however, seem a bit strange that we only update φ' on such long intervals: there is a considerable jump between the maximum "lag" on the Nth step and none on the $(N+1)$th step. A popular alternative, similar to Polyak averaging in convex optimization, is updating $\varphi' \leftarrow \tau\varphi' + (1-\tau)\varphi$, where $\tau = 0.999$ works well.

Remark. All Q-learning methods appear somewhere in Figure 5, with different frequencies for the three processes. For instance, online Q-learning has frequencies $1 = 2 = 3$, DQN $1 = 3 > 2$, and

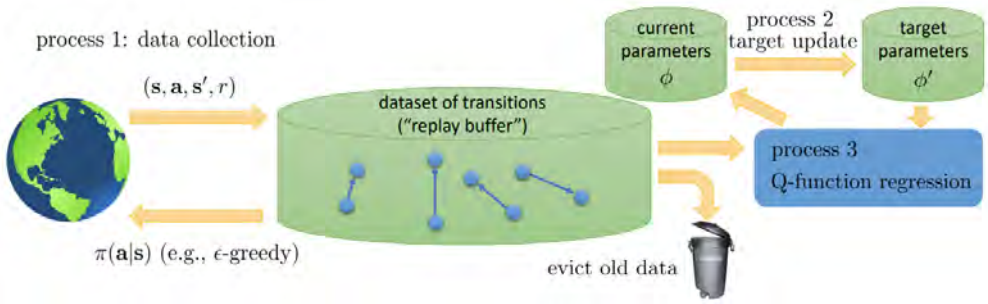


Figure 5: High-level overview of Q-learning methods

fitted Q-iteration $3 > 2 > 1$.

Q-learning in practice

Question 8.1. How accurate are Q-values?

Answer. Not very. Generally they follow the same trend as the true values, but are systematically much higher.

Recall that for random variables X and Y, $\mathbb{E}[\max(X, Y)] \geq \max(\mathbb{E}X, \mathbb{E}Y)$. With this in mind, consider our target value update $y_j = r_j + \max_{a'_j} Q_{\varphi'}(s'_j, a'_j)$. Our $Q_{\varphi'}$ learns some noise, so it'll overestimate the true value even if $Q_{\varphi'}$ is unbiased.

A solution is **double Q-learning**. Recall that our update worked because of the "argmax trick"

$$\max_{a'} Q_{\varphi'}(s', a') = Q_{\varphi'}(s', \arg\max_{a'} Q_{\varphi'}(s', a')).$$

The issue is that both the action and the value come from $Q_{\varphi'}$. If that noise is decorrelated, then the issue should go away (we assume that φ and φ' have sufficiently uncorrelated noise):

$$y \leftarrow r + \gamma Q_{\varphi'}\left(s', \arg\max_{a'} Q_{\varphi}(s', a')\right).$$

Question 8.2. When Q-values are bad, they are primarily guided by rewards. Can we construct multi-step targets to mitigate this?

Answer. Yes:

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

This is good because target values are less biased, and training usually happens more quickly early on. However, it is only correct when learning on-policy (you sample from your buffer instead of the actions your policy would've actually taken).

Below we describe some implementation notes:

- Q-learning takes care to stabilize; its performance on different tasks will vary wildly

- Large replay buffers improve stability
- Gradients can be massive, so you should use clipped gradients or **Huber loss**

$$L(x) = \begin{cases} x^2/2 & \text{if } |x| \leq \delta \\ \delta|x| - \delta^2/2 & \text{else} \end{cases}$$

- Double Q-learning performs way better, with virtually no downsides

Continuous actions

For discrete actions, we evaluated the max or arg max for our update rules. What is the analog?

1. Optimization
   - SGD is pretty slow in the inner loop
   - Pretend it's discrete: sample $a_1, \ldots, a_N$, and approximate $\max_a Q(s,a) \approx \max\{Q(s,a_i)\}$
   - Cross-entropy method (CEM): simple iterative stochastic optimization
   - CMA-ES: a less simple iterative stochastic optimization method
2. Use function class that is easy to optimize:

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

   Then the max is $V_\varphi(s)$, achieved at $a = \mu_\varphi(s)$.
3. Use an approximate maximizer $\mu_\theta(s)$ such that $\mu_\theta(s) \approx \arg\max_a Q_\varphi(s,a)$. You can just use this by pushing gradients, $\frac{\mathrm{d}Q_\varphi}{\mathrm{d}\theta} = \frac{\mathrm{d}Q_\varphi}{\mathrm{d}a}\frac{\mathrm{d}a}{\mathrm{d}\theta}$. In fact, this method exists and is called **DDPG** Lillicrap et al. [22].

Algorithm 8 DDPG

for run K times **do**
  take an action a_i, observe (s_i, a_i, s_i', r_i), and add it to $\mathcal{B}$
  sample a minibatch
  compute target $y_j = r_j + \gamma Q_{\varphi'}(s_j', \mu_\theta(s_j'))$ using target nets $Q_{\varphi'}$ and μ_θ
  $\varphi \leftarrow \varphi - \alpha \sum_j \frac{\mathrm{d}Q_\varphi(s_j,a_j)}{\mathrm{d}\varphi}(Q_\varphi(s_j,a_j) - y_j)$
  $\theta \leftarrow \theta + \beta \sum_j \frac{\mathrm{d}Q_\varphi(s_j,\mu(s_j))}{\mathrm{d}\mu(s_j)}\frac{\mathrm{d}\mu(s_j)}{\mathrm{d}\theta}$
  update φ' and θ' via Polyak averaging
end for

9 Advanced Policy Gradients (09/20)

The videos begin here.

Why does policy gradient work?

Generally, you can think of policy gradient as two steps alternating: (10 estimating $\hat{A}^\pi(s_t, a_t)$ for the current policy π, then (2) using $\hat{A}^\pi(s_t, a_t)$ to improve your policy π'. Structurally, this looks

similar to policy iteration, which alternates (1) evaluate $A^\pi(s,a)$ and (2) set $\pi \leftarrow \pi'$. Analogously, we want to optimize our new policy over the old policy relative to the new policy.

Why do we care about expectation relative to θ rather than θ? (Doesn't it solve our problems w distribution shift?)

Proposition 9.1.

$$J(\theta') - J(\theta) = \mathbb{E}_{\tau\sim\pi_{\theta'}}\left[\sum_t \gamma^t A^{\pi_\theta}(s_t,a_t)\right]$$

Proof.

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

as claimed. Note that the second equality holds for any policy with initial state distribution p. ■

We want to convert this such that the expectation is under π_θ:

$$\begin{aligned}
\mathbb{E}_{\tau\sim\pi_{\theta'}}\left[\sum_t \gamma^t A^{\pi_\theta}(s_t,a_t)\right] &= \sum_t \mathbb{E}_{s_t\sim p_{\theta'}}\left[\mathbb{E}_{a_t\sim\pi_{\theta'}(\cdot|s_t)}\left[\gamma^t A^{\pi_\theta}(s_t,a_t)\right]\right] \\
&= \sum_t \mathbb{E}_{s_t\sim p_{\theta'}}\left[\mathbb{E}_{a_t\sim\pi_{\theta}(\cdot|s_t)}\left[\frac{\pi_{\theta'}(a_t \mid s_t)}{\pi_\theta(a_t \mid s_t)}\gamma^t A^{\pi_\theta}(s_t,a_t)\right]\right] =: \bar{A}(\theta).
\end{aligned}$$

To do so, we want $p_{\theta'} \approx p_\theta$. Specifically,

Claim 9.2. Suppose that for all s_t, $|\pi_{\theta'}(a_t \mid s_t) - \pi_\theta(a_t \mid s_t)| \le \varepsilon$. Then for small enough ε, the update $\theta' \leftarrow \arg\max_{\theta'} \bar{A}(\theta')$ is guaranteed to improve $J(\theta') - J(\theta)$.

First, we state the following lemma without proof:

Lemma 9.3. *If $\sum_x |p_X(x) - p_Y(y)| = \varepsilon$, there exists $p(x,y)$ such that $p(x,\cdot) = p_X(x)$, $p(\cdot,y) = p_Y(y)$, and $p(x=y) = 1-\varepsilon$.*

Proof of Claim 9.2. By the lemma, how?

$$|p_{\theta'}(s_t) - p_\theta(s_t)| = (1-(1-\varepsilon)^t)|p_{\text{mistake}}(s_t) - p_\theta(s_t)| \le 2(1-(1-\varepsilon)^t) \le 2\varepsilon t.$$

Then for any function f,

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

Note that we can upper-bound the quantity inside the expectation over states in $\bar{A}(\theta)$ by $O(Tr_{\max})$, so converting the expectation over $p_{\theta'}$ to one over p_θ is guaranteed to improve $J(\theta') - J(\theta)$ for sufficiently small θ. ■

Remark. The proof still works if our assumption of "closeness" on the policies is true in expectation over s_t.

This is pretty nice, but even in expectation, the bound on the absolute divergence between p_θ and $p_{\theta'}$ is pretty tight. Instead, we'll impose a different constraint, which works much better because the KL-divergence has some convenient properties that make it much easier to approximate:

Definition 9.4. The KL-divergence of distributions p_1 and p_2 is

$$D_{KL}(p_1(x) \parallel p_2(x)) = \mathbb{E}_{x\sim p_1}\left[\log \frac{p_1(x)}{p_2(x)}\right].$$

Claim 9.5 (Almost the same as Claim 9.2)**.** Suppose that $D_{KL}(\pi_{\theta'}(a_t \mid s_t) \parallel \pi_\theta(a_t \mid s_t)) \leq \varepsilon$. Then for small enough ε, the update $\theta' \leftarrow \arg\max_{\theta'} \bar{A}(\theta')$ is guaranteed to improve $J(\theta') - J(\theta)$.

Proof. "Recall" from convex optimization that to enforce the constraint on the KL-divergence, we use the **Lagrangian**

$$\mathcal{L}(\theta', \lambda) := A(\theta') - \lambda(D_{KL}(\pi_{\theta'}(a_t \mid s_t) \parallel \pi_\theta(a_t \mid s_t)) - \varepsilon)$$

(where λ is a Lagrange multiplier) and perform dual gradient descent:

1. maximize $\mathcal{L}(\theta', \lambda)$ with respect to θ' [just run a few gradient steps]
2. $\lambda \leftarrow \lambda + \alpha(D_{KL}(\pi_{\theta'}(a_t \mid s_t) \parallel \pi_\theta(a_t \mid s_t)) - \varepsilon)$.

■

Natural gradient

This is based on Peters and Schaal [28]. In practice, it can work much better than the standard gradient.

We can approximately optimize an objective by using a Taylor approximation and optimizing over a small neighborhood. In the case of linearization, optimization is trivial (you just need to take the lower endpoint). In this case, we take

$$\theta' \leftarrow \arg\max_{\theta'} \nabla_{\theta'} \bar{A}(\theta')^\top (\theta' - \theta) \quad \text{subject to} \quad D_{KL}(\pi_{\theta'}(a_t \mid s_t) \parallel \pi_\theta(a_t \mid s_t)) \leq \varepsilon.$$

This is especially clean because

$$\begin{aligned}\nabla_{\theta'}\bar{A}(\theta') &= \sum_t \mathbb{E}_{s_t\sim p_\theta}\left[\mathbb{E}_{a_t\sim\pi_\theta(\cdot|s_t)}\left[\frac{\pi_{\theta'}(a_t\mid s_t)}{\pi_\theta(a_t\mid s_t)}\nabla_{\theta'}\log\pi_{\theta'}(a_t\mid s_t)\gamma^t A^{\pi_\theta}(s_t,a_t)\right]\right]\\ &= \sum_t \mathbb{E}_{s_t\sim p_\theta}\left[\mathbb{E}_{a_t\sim\pi_\theta(\cdot|s_t)}\left[\nabla_\theta\log\pi_\theta(a_t\mid s_t)\gamma^t A^{\pi_\theta}(s_t,a_t)\right]\right],\end{aligned}$$

the standard policy gradient.

So the constrained optimization problem remains. One can observe that gradient ascent is equivalent to $\arg\max_{\theta'}\nabla_\theta J(\theta)^\top(\theta'-\theta)$ subject to $\|\theta'-\theta\|_2^2\le\varepsilon$. We can approximate our problem (the same thing, but with a different constraint) to apply gradient ascent: in another lecture, we saw that $D_{KL}(\pi_{\theta'}\parallel\pi_\theta)\approx\frac{1}{2}(\theta'-\theta)^\top F(\theta'-\theta)$, where

$$F=\mathbb{E}_{\tau\sim\pi_\theta}[(\nabla_\theta\log\pi_\theta(a\mid s))(\nabla_\theta\log\pi_\theta(a\mid s))^\top]$$

is the **Fisher information matrix** (we collect it via samples). Thus, our **natural gradient** update is

$$\theta'=\theta+\alpha F^{-1}\nabla_\theta J(\theta),\quad \alpha=\sqrt{\frac{2\varepsilon}{\nabla_\theta J(\theta)^\top F\nabla_\theta J(\theta)}}.$$

In practice, computing these product efficiently is nontrivial; see Schulman et al. [31].

10 Optimal Control and Planning (09/25)

The videos begin here.

So far, we've only looked at model-free RL, i.e. we treat $p(s_{t+1}\mid s_t,a_t)$ as unknown and make no attempt to learn it. But in games, easily modeled systems (e.g. navigating a car), simulated environments, we know the dynamics completely. In some cases, we can learn dynamics, either through system identification or learning (in this course, mostly the latter).

The deterministic case $s_{t+1}=f(s_t,a_t)$ is pretty simple:

$$a_1,\dots,a_T=\arg\max_{a_t}\sum_{t=1}^T r(s_t,a_t).$$

The stochastic open-loop[1] case is

$$a_1,\dots,a_T=\arg\max_{a_t}\mathbb{E}\left[\sum_t r(s_t,a_t)\,\middle|\,a_1,\dots,a_T\right],$$

and the stochastic closed-loop case is

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

[1]Closed-loop means that the agent is allowed to run one action at a time, $a_t\sim\pi(\cdot\mid s_t)$. Open-loop means that the agent gets one state and must commit to a sequence of actions $a_1,\dots,a_T$.

Open-loop planning

We abstract away optimal control and planning, regarding the sum of rewards as a function J and $a_1, \dots, a_T$ as a vector $\mathbf{a}$ (we don't care about time dependency), leaving us to compute $\arg\max_{\mathbf{a}} J(\mathbf{a})$. Here are some ideas:

- **Random shooting:** Pick $\mathbf{a}_1, \dots, \mathbf{a}_n$ uniformly at random. Then take $\mathbf{a} = \arg\max_i J(\mathbf{a}_i)$.
  - Is super easy to code and can be implemented with some nice tricks on a GPU.
  - Performance might not be good.
- **Cross-entropy method:** Loop the following: Pick $\mathbf{a}_1, \dots, \mathbf{a}_n$ from some distribution p (e.g. Gaussian). Then refit p on the "elites" $\mathbf{a}_{i_1}, \dots, \mathbf{a}_{i_M}$ with the largest $J(\cdot)$.
  - Easy to implement and fast when parallelized.
  - Harsh dimensionality limit, and only works with open-loop planning.

Discrete case: Monte Carlo tree search

Consider a tree, where each node represents a unique sequence of actions; you want to approximate J without the full tree. So you'll take a few actions, then run a baseline policy π at each node. Intuitively, we want to choose nodes with the best rewards, but also prefer rarely visited nodes.

Algorithm 9 MCTS Sketch

procedure UCT TREEPOLICY(s_1) ▷ not actually a policy
  if s_t not fully expanded, choose next a_t
  else choose child state with best $\text{Score}(s_{t+1}) = \frac{Q(s_{t+1})}{N(s_{t+1})} + 2C\sqrt{\frac{2\ln N(s_t)}{N(s_t)}}$ ▷ $Q(\cdot)$ is the reward of a node, and $N(\cdot)$ is the number of times we've visited it
end procedure
for repeat K times **do**
  find a leaf s_ℓ with TREEPOLICY(s_1)
  evaluate the leaf using DEFAULTPOLICY(s_ℓ)
  update all values in tree between s_1 and s_ℓ
end for
take best action from s_1

Continuous case: trajectory optimization with derivatives

Control theory usually uses $(\mathbf{x}_t, \mathbf{u}_t)$ as states and actions, and minimizes a cost c, so that the objective is

$$\min_{\mathbf{u}_t} \sum_{t=1}^{T} c(\mathbf{x}_t, \mathbf{u}_t) \quad \text{s.t.} \quad x_t = f(\mathbf{x}_{t-1}, \mathbf{u}_{t-1}),$$

or equivalently

$$\min_{\mathbf{u}_t} c(\mathbf{x}_1, \mathbf{u}_1) + c(f(\mathbf{x}_1, \mathbf{u}_1), \mathbf{u}_2) + \cdots + c(f(f(\cdots)\cdots), \mathbf{u}_T).$$

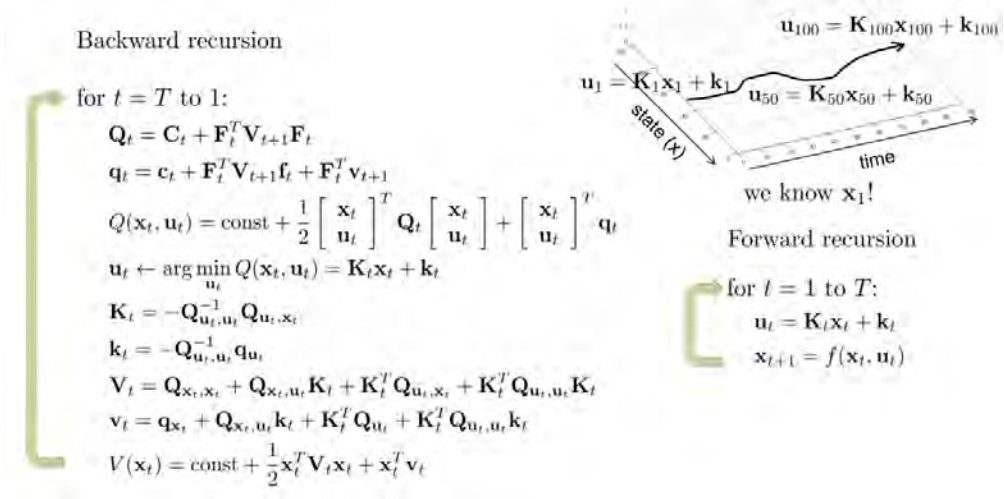


Figure 6: LQR

Unfortunately, your typical gradient methods won't work well, since applying f many times means multiplying many Jacobians together and requires singular values close to 1 (else exploding/vanishing problem). In practice, it helps to use second-order methods (and there exist some that don't require computing massive Hessians!).

First, we'll consider the "linear case"

$$f(\mathbf{x}_t, \mathbf{u}_t) = F_t \begin{bmatrix} \mathbf{x}_t \\ \mathbf{u}_t \end{bmatrix} + \mathbf{f}_t$$

$$c(\mathbf{x}_t, \mathbf{u}_t) = \frac{1}{2} \begin{bmatrix} \mathbf{x}_t \\ \mathbf{u}_t \end{bmatrix}^\top C_t \begin{bmatrix} \mathbf{x}_t \\ \mathbf{u}_t \end{bmatrix} + \begin{bmatrix} \mathbf{x}_t \\ \mathbf{u}_t \end{bmatrix}^\top \mathbf{c}_t$$

We can optimize this one entry at a time; in the objective, note that $\mathbf{u}_T$ only appears once, in the final term. The math is pretty annoying, but conceptually it's very straightforward to differentiate with respect to $\mathbf{u}_T$ (giving a linear expression in $\mathbf{x}_T$) and substitute to optimize both variables. Next, we optimize $(\mathbf{x}_{T-1}, \mathbf{u}_{T-1})$; we must be careful because they influence $\mathbf{x}_T$, but again we get a quadratic, and it's not so bad. See Figure 6 for the full algorithm.

Remark. It turns out that for stochastic dynamics given by $x_{t+1} \sim \mathcal{N}\left(F_t \begin{bmatrix} \mathbf{x}_t \\ \mathbf{u}_t \end{bmatrix} + \mathbf{f}_t, \Sigma_t\right)$, we choose actions using exactly the same update we saw in the linear case. To prove this, you use the fact that Gaussians are symmetric and that expectation of quadratics under Gaussians are analytic.

In the case of nonlinear systems, we can simply use a Taylor approximation; see Figure 7. One may observe that it is almost the same as Newton's method, excecpt that we still have f linear. If we make our approximation to f quadratic, then the resulting algorithm, **differential dynamic programming (DDP)**, is really just Newton's method.

11 Model-Based RL (09/27)

The videos begin here.

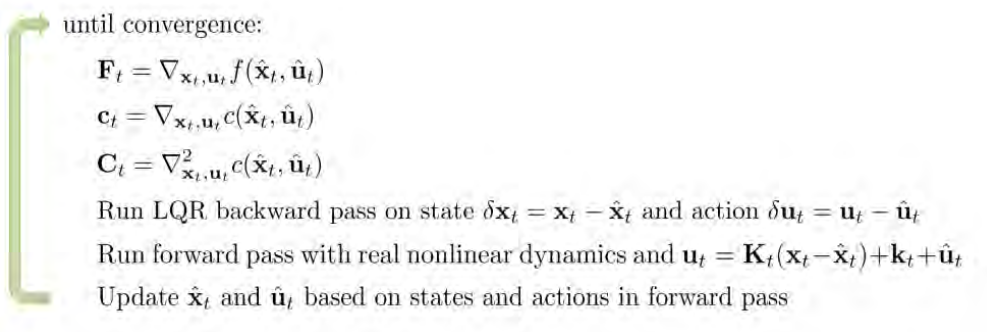


Figure 7: Iterative LQR

Suppose (WLOG) that our world is deterministic, and we want to learn $f(s_t, a_t)$ from data. The basic strategy is as follows:

- run base policy $\pi_0(a_t \mid s_t)$ 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

This strategy works well in some cases, e.g. system identification in robotics. Typically, in this case, we fit parameters to a specific model (e.g. physics), and there are not very many of them. In deep neural nets, however, we experience the problem of distributional shift, since π_0 won't collect as much data on states with low marginals. As we use more expressive model classes, the fit will be tighter, and the problem becomes worse.

We apply two fixes:

- In DAgger, we found that this problem was pretty hard to resolve; we required a human expert to tell us the best actions at each state. With model-based RL, this is easy: just execute the actions and see what happens.
- We don't need to relearn the dynamics model every time, and furthermore executing several actions consecutively loses information. So instead, we'll just execute one action at a time and append to our dataset to adjust for errors as quickly as possible.

Algorithm 10 Model Predictive Control Mayne et al. [24]

run base policy $\pi_0(a_t \mid s_t)$ to collect $\mathcal{D} = \{(s, a, s')_i\}$
for run K times **do**
  learn dynamics model $f(s, a)$ to minimize $\sum_i \|f(s_i, a_i) - s_i'\|^2$
  for run N times **do**
    plan through $f(s, a)$ to choose actions
    execute the first action and append (s, a, s') to $\mathcal{D}$
  end for
end for

The computational bottleneck is planning. In practice, since we are doing so much planning, each individual plan can afford to be worse. Things like shortening horizons or even random shooting can work well.

Uncertainty

By default, model-based RL may have low performance because it overfits to its collected data. If there are weird spikes in the learned reward function, it's tempting for the planner to exploit mistakes in the model, particularly in the early stages of training where uncertainty is high.

Given a particular state-action pair, instead of predicting a particular state s', we predict a distribution over states under your uncertainty about the model. Importantly, predicting the expected reward is sufficient here, since the planner will naturally avoid large negative rewards.

As a clarification, there are two types of uncertainty:

- aleatoric or **statistical uncertainty** (the data is noisy),
- epistemic or **model uncertainty** (the true function isn't noisy, but we don't know what the "right function" is).

Outputting the MLE or the distribution over the next state doesn't resolve the latter problem, and this is what we're interested in solving.

In the case of neural nets, suppose that the model parameters θ yield a distribution $p_\theta(s_{t+1} \mid s_t, a_t)$. We are uncertain about θ; the entropy of $p(\theta \mid \mathcal{D})$ represents our model uncertainty, and we'll predict according to $\mathbb{E}_{\theta \sim p(\cdot \mid \mathcal{D})}[p_\theta(s_{t+1} \mid s_t, a_t)]$. Computing this expectation is intractable, so we want a good way to estimate $p(\theta \mid \mathcal{D})$.

Note. A brief aside: traditionally, neural nets have weights assigned to each edge. In Bayesian neural nets, we assign distributions, i.e. $p(\theta_i \mid \mathcal{D}) = \mathcal{N}(\mu_i, \sigma_i^2)$. (There are two weights per θ_i.)

We'll use bootstrap ensembles: we generate "independent" datasets to train "independent" models. If we have N models (usually < 10 due to computational requirements), then we will write

$$p(\theta \mid \mathcal{D}) \approx \frac{1}{N} \sum_i \delta(\theta_i) \implies \int p(s_{t+1} \mid s_t, a_t) p(\theta \mid \mathcal{D}) \mathrm{d}\theta \approx \frac{1}{N} \sum_i p(s_{t+1} \mid s_t, a_t).$$

In theory, the idea of bootstrapping is to train θ_i on $\mathcal{D}_i$, which is sampled with replacement from $\mathcal{D}$. In practice, it's usually unnecessary to do this, since SGD and random initialization are enough to make them roughly independent.

Finally, suppose we've trained our N uncertainty models and want to plan using them. If we have some candidate action sequence $a_1, \ldots, a_T$, we will:

1. sample $\theta \sim p_i(\cdot \mid \mathcal{D})$
2. sample $s_{t+1,i} \sim p_\theta(\cdot \mid s_{t,i}, a_{t,i})$
3. compute $R_i = \sum_{t=1}^T r(s_{t,i}, a_{t,i})$
4. compute $\frac{1}{N} \sum_{i=1}^N R_i$

In practice, this works quite well; in Chua et al. [3], model-based methods exceed model-free performance in around 10 minutes of training.

Images

Complex observations, e.g. images, are hard to deal with using $f(s_t, a_t) = s_{t+1}$, since they have high dimensionality, lots of redundancy, and are working with a POMDP (we don't know states!). Instead, we can separately learn the observation, dynamics, and reward models, as in Figure 8.

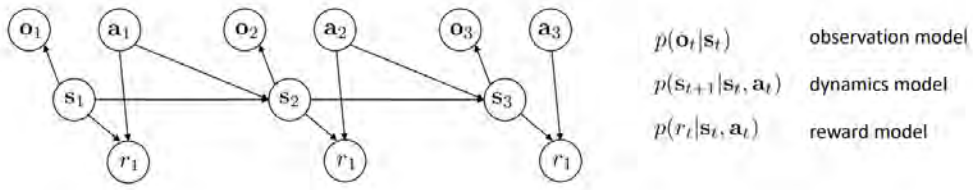


Figure 8: State space models

With this in mind, the objective is as follows:

$$\max_{\phi} \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} E\left[\log p_\phi(\mathbf{s}_{t+1,i}|\mathbf{s}_{t,i}, \mathbf{a}_{t,i}) + \log p_\phi(\mathbf{o}_{t,i}|\mathbf{s}_{t,i})\right]$$

$$\text{expectation w.r.t. } (\mathbf{s}_t, \mathbf{s}_{t+1}) \sim p(\mathbf{s}_t, \mathbf{s}_{t+1}|\mathbf{o}_{1:T}, \mathbf{a}_{1:T})$$

Thus, we want to learn an approximate posterior $q_\psi(s_t \mid o_{1:t}, a_{1:t})$, which we call the "encoder." In the very simple case where $q(s_t \mid o_t)$ is deterministic, i.e. $s_t = g_\psi(o_t)$ (the stochastic case requires variational inference), the problem is equivalent to

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

latent space dynamics image reconstruction reward model

(The previous objective omitted reward, but we've added it back here.) This is good, because we can optimize ϕ and ψ jointly with backprop.

The final algorithm looks very similar to Algorithm 10. In line 3, we instead learn $p_\phi(s_{t+1} \mid s_t, a_t)$, $p_\phi(r_t \mid s_t)$, $p(o_t \mid s_t)$, and $g_\psi(o_t)$. In line 6, we instead append (o, a, o').

12 Model-based RL with policies (10/02)

The videos begin here.

Review of last week:

- Open-loop control is suboptimal.

  Example 12.1. Consider a math test where you add two one-digit numbers. There are two timesteps. On the first, you can either accept the test or go home. On the second, you get to look at the test and produce an answer. If you get the answer right, you get \$2; wrong, -\$1; stay home, \$0.

  In open-loop planning, you have to commit to both actions, which is necessarily suboptimal.

  MPC doesn't solve this, because even though it plans at every timestep, each plan is open-loop.

  The closed-loop case solves this problem.

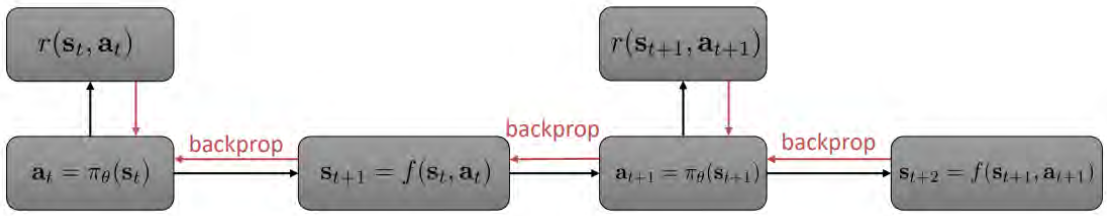


Figure 9: Backprop into policy computation graph

- We have to decide the form of π. Common choices are a neural net or a time-varying linear map $K_t s_t + k_t$.

Question 12.2. What's the problem with backprop directly into the policy? See Figure 9, where we optimize the negative sum of rewards.

Answer.

- With shooting methods, we had LQR. There is no analog here, because policy parameters couple all the timesteps and we cannot use dynamic programming (we no longer benefit from the temporal structure). Moreover, second-order methods are usually pretty flaky for neural nets.
- This is similar to training long RNNs with vanilla backprop, and without fancy architectures will produce vanishing or exploding gradients. Also, we cannot just choose dynamics; LSTMs will produce Jacobians close to the identity matrix, but we can't just choose the dynamics (the real thing might have large or small eigenvalues, which our patch won't fix).

Model-free learning with a model

The idea is that you don't need to actually run the policy all that often; you can run your model very quickly and use it as a proxy for the real world.

However, as-is, this would result in distributional shift, and is exacerbated by the fact that the policy is updated periodically (we aren't sampling from the same policy from which our data was collected), and as in imitation learning, it's possible to prove that error accumulates in $O(\varepsilon T^2)$.

The issue with just taking T small is that you'll never see later timesteps. One workaround is that you can run some long trajectories to produce $\mathcal{D}$, and generate short trajectories (on the order of ≤ 10 steps) starting at different states in $\mathcal{D}$. However, this means that:

- You're using a different policy to roll into the state than to roll out of it.
- As a result, your state distribution is complicated: it's a combination of the two policies.

In practice, people use a method based on Dyna Sutton [35]. See Algorithm 12.

For example, see Algorithm 13, which, along with MBA Gu et al. [10] and MVE Feinberg et al. [5] apply some specific design decisions to the preceding general structure.

- These methods tend to be sample-efficient. We add data from running the policy to our buffer, which is fed back in as input.
- If our model $\hat{p}$ is bad, then our model-based rollout won't provide good data we add to $\mathcal{B}$.

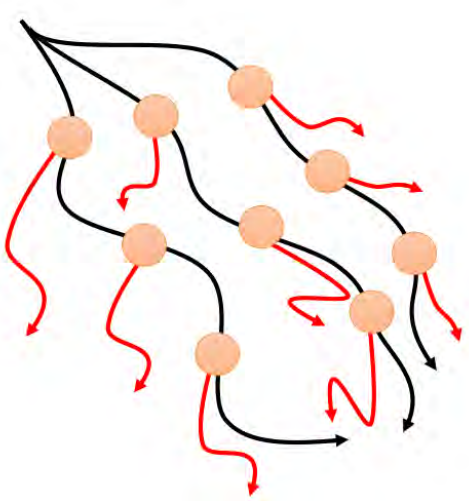


Figure 10: Black represents true rollouts from the data, and red represents running the policy from different points along the true trajectories.

Algorithm 11 Model-based RL with short rollouts

1: run base policy $\pi_0(a_t \mid s_t)$ to collect $\mathcal{D} = \{(s, a, s')_i\}$
2: learn dynamics model $f(s, a)$ to minimize $\sum_i \|f(s_i, a_i) - s_i'\|^2$
3: **for** loop N times **do**
4: **for** loop K times **do**
5: pick states s_i from $\mathcal{D}$, use $f(s, a)$ to generate short rollouts
6: use both real and model data to improve $\pi_\theta(a \mid s)$ with off-policy RL
7: **end for**
8: run $\pi_\theta(a_t \mid s_t)$, appending visited tuples (s, a, s') to $\mathcal{D}$
9: **end for**

Algorithm 12 Generalization of Dyna

collect some data $\mathcal{B}$ $\{s, a, s', r)\}$
learn model $\hat{p}(s' \mid s, a)$ and $\hat{r}(s' \mid s, a)$
for loop K times **do**
 sample $s \in \mathcal{B}$, choose action a (from $\mathcal{B}$, π, or random)
 simulate $s' \sim \hat{p}(s' \mid s, a)$ and $r = \hat{r}(s, a)$
 train on (s, a, s', r) with model-free RL
 take N more model-based steps
end for

Algorithm 13 Model-Based Policy Optimization

1: take some action a_i and observe (s_i, a_i, s_i', r_i), add it to $\mathcal{B}$
2: **for** loop K times **do**
3: sample mini-batch $\left\{s_j, a_j, s_j', r_j\right\}$ from $\mathcal{B}$ uniformly
4: use $\left\{s_j, a_j, s_j'\right\}$ to update model $\hat{p}(s' \mid s, a)$
5: sample $\left\{s_j\right\}$ from $\mathcal{B}$
6: for each s_j, perform model-based rollout with $a = \pi(s)$
7: use all transitions (s, a, s', r) along rollout to update Q-function
8: **end for**

Multi-step models and successor representations

Recall that our objective in evaluation is

$$J(\pi) = \mathbb{E}_{s\sim p(s_1)}[V^\pi(s_1)]$$
$$V^\pi(s_t) = \sum_{t=t'}^{\infty} \gamma^{t'-t} \mathbb{E}_{s_{t'}\sim p(\cdot|s_t)} \mathbb{E}_{a_{t'}\sim\pi(\cdot|s_{t'})}[r(s_{t'}, a_{t'})] = \sum_{t=t'}^{\infty} \gamma^{t'-t} \mathbb{E}_{s_{t'}\sim p(\cdot|s_t)}[r(s_{t'})]$$

For simplicity of notation, we'll condition on just states, and add actions back in at the end (they were just sampled from π).

$$= \sum_{t=t'}^{\infty} \gamma^{t'-t} \mathbb{E}_{s_{t'}\sim p(\cdot|s_t)}[r(s_{t'})]$$
$$= \sum_s r(s) \sum_{t=t'}^{\infty} \gamma^{t'-t} p(s_{t'} = s \mid s_t).$$

Define $p_\pi(s_{\text{future}} = s \mid s_t) := (1-\gamma)\sum_{t=t'}^{\infty} \gamma^{t'-t} p(s_{t'} = s \mid s_t)$, where the $1-\gamma$ is to ensure that summing over s_t yields 1, so that

$$= \frac{1}{1-\gamma} \sum_s p_\pi(s_{\text{future}} = s \mid s_t) r(s).$$

Equivalently, one can take $\mu_i^\pi(s_t) := p_\pi(s_{\text{future}} = i \mid s_t)$ (the **successor representation**), so that

$$= \frac{1}{1-\gamma} \boldsymbol{\mu}^\pi(s_t)^\top \mathbf{r}.$$

Like a model, the successor representation predicts future states, but like a value function, it is a discounted average.

One can observe by definition that the successor representation satisfies a Bellman equation

$$\mu_i^\pi(s_t) = (1-\gamma)\mathbb{1}\{s_t = i\} + \gamma \mathbb{E}_{a_t\sim\pi(\cdot|s_t), s_{t+1}\sim p(\cdot|s_t,a_t)}[\mu_i^\pi(s_{t+1})].$$

Let $\varphi_j(s)$ represent some features, and define

$$\psi_j^\pi(s_t) = \sum_s \mu_s^\pi(s_t)\varphi_j(s).$$

If we can express $r(s)$ as a linear combination $\mathbf{w}^\top \boldsymbol{\varphi}(s)$, then it is easy to check that $V^\pi(s_t) = \mathbf{w}^\top \psi^\pi(s_t)$. The benefit is that we can make the number of features much smaller than the number of states, which makes this computationally tractable.

A result of this is that we can use successor features to recover Q-functions very quickly. Suppose we manually define our features ahead of time. Then This is equivalent to one step of policy

Algorithm 14 Recovering Q-function

1: train $\psi^\pi(s_t, a_t)$ via Bellman updates
2: get some sample rewards $\{(s, r)_i\}$
3: $\mathbf{w} \leftarrow \arg\min_{\mathbf{w}} \sum_i \left\| \boldsymbol{\varphi}(s_i)^\top \mathbf{w} - r_i \right\|^2$
4: recover $Q^\pi(s_t, a_t) \approx \boldsymbol{\psi}^\pi(s_t, a_t)^\top \mathbf{w}$
5: $\pi'(s) = \arg\max_a \boldsymbol{\psi}^\pi(s, a)^\top \mathbf{w}$

iteration, i.e. it does not produce the optimal Q-function. One idea is to recover many Q-functions:

Algorithm 15 Recover many Q-functions

1: train $\psi^{\pi_k}(s_t, a_t)$ for many policies via Bellman updates
2: get some reward samples $\{(s, r)_i\}$
3: $\mathbf{w} \leftarrow \arg\min_{\mathbf{w}} \sum_i \left\| \boldsymbol{\varphi}(s_i)^\top \mathbf{w} - r_i \right\|^2$
4: recover $Q^\pi(s_t, a_t) \approx \psi^{\pi_k}(s, a)^\top \mathbf{w}$ for every π_k
5: $\pi'(s) = \arg\max_a \max_k \psi^{\pi_k}(s, a)^\top \mathbf{w}$

In the continuous case, we can frame successor representations as classification: take $F = \mathbb{1}\{s_{\text{future}} \text{ is a future state}$ Then

$$p^\pi(s_{\text{future}} \mid s_t, a_t) = p^\pi(s_{\text{future}}) \frac{p^\pi(F = 1 \mid s_t, a_t, s_{\text{future}})}{p^\pi(F = 0 \mid s_t, a_t, s_{\text{future}})}.$$

$p^\pi(s_{\text{future}}$ is a constant independent of s_t, a_t, so really we care most about the classifier. We can train it simply by sampling from p^π and applying a cross-entropy loss.

13 Exploration (10/04)

The videos begin here.

Example 13.1. Montezuma's revenge is pretty hard for RL. It understands that getting a key is a reward and opening a door is a reward, but getting killed by a skull doesn't have any positives or negatives (you start over). Finishing the game only weakly correlates with rewarding events. We only know what to do because we understand what keys and skulls mean symbolically.

This becomes more difficult as you extend the task or know less about the rules.

Definition 13.2. Exploitation is doing what you know will yield the highest reward. **Exploration** is doing things you haven't done before, in the hopes of getting even higher rewards.

Question 13.3. Can we derive an optimal exploration strategy? What does optimal mean?

In general, it may not be possible to determine optimality (consider, for instance, an infinite MDP or continuous state space). We'll adapt theory from smaller cases, e.g. multi-armed bandits (1-step stateless RL problems) to intractable ones.

You can think of a one-armed bandit as a slot machine and a multi-armed bandit problem as N slot machines. In the one-armed case, you decide whether to play (there's only one action), and in the multi-armed case, you care about which machine to use.

Define $r(a_i) \sim p_{\theta_i}$ (for example, Bern(θ_i)). Suppose we have some prior $\theta_i \sim p(\theta)$. Each time you pull an arm, you can update your belief state $\hat{p}(\theta_1, \dots, \theta_n)$. But solving the POMDP is very overkill: p might be super complicated with nonzero correlations between θ_i's and so on. It turns out that we can do pretty well with much simpler strategies, and say that a strategy is optimal if its reward is $\Theta(\text{POMDP optimal})$.

Formally, our objective is to minimize

$$\text{regret}(T) = T\mathbb{E}[r(a^*)] - \sum_{t=1}^{T} r(a_t).$$

Optimistic exploration

We can take $a = \arg\max(\hat{\mu}_a + C\sigma_a)$, where $\hat{\mu}_a$ is the average reward so far and σ_a is a variance estimate. It turns out that using $a = \arg\max\left(\hat{\mu}_a + \underbrace{\sqrt{\frac{2\ln T}{N(a)}}}_{\text{exploration bonus}}\right)$, where $N(a)$ is the number of times we have picked a, yields regret $O(\log T)$.

You can use the same idea with any MDP. Suppose you maintain an exploration counter $N(s)$ or $N(s,a)$. Then $r^+(s,a) := r(s,a) + \mathcal{B}(N(s))$, where $\mathcal{B}$ is a bonus that decreases with $N(s)$, can be used as a replacement for r in any model-free algorithm.

This doesn't work in the case of large or continuous environments, where it will be very unlikely to see the same state more than once. A workaround is to fit a density model $p_\theta(s)$ (how likely is it to see s?). We fit a model to "pseudo-counts"; see Figure 11.

There are several possible bonuses $\mathcal{B}$ we can use, like the UCB bonus $\sqrt{2\ln T/N(a)}$ above or $\sqrt{1/N(a)}$. There are probably reasons for choosing specific ones, but they all work.

$p_\theta(\cdot)$ is designed to produce output densities, but it doesn't necessarily need to produce good samples (i.e., the opposite considerations as a GAN or VAE).

There are several ideas in the literature providing alternative implementations of optimism:

- Compress s into a k-bit code via $\varphi(s)$ (a lossy compression; you'll get collisions), then count $N(\varphi(s))$. Tang et al. [37] You learn a compression such that similar states get the same hash (and need to mess with code lengths so that you get an appropriate number of collisions).
- A state is novel if it is easy to distinguish from all previous seen states by a classifier. So we can fit a classifier $p_\theta(s) = \frac{1-D_s(s)}{D_s(s)}$ using an amortized model Fu et al. [8].

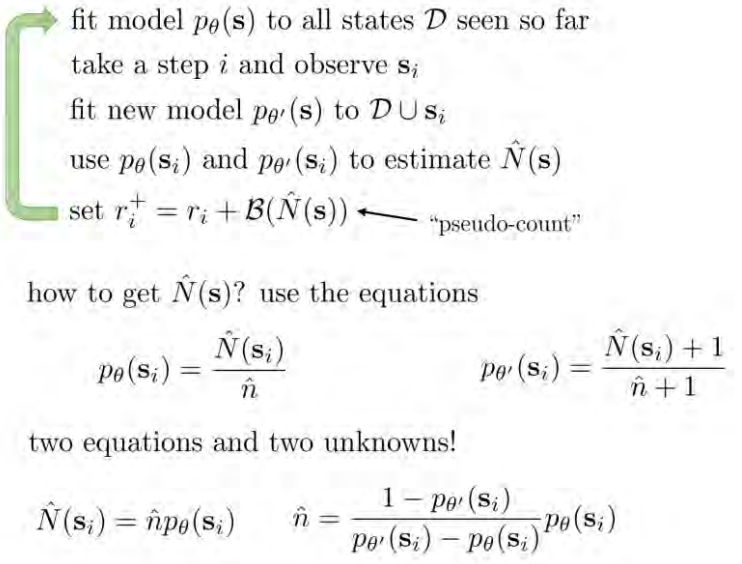


Figure 11: Exploring with pseudo-counts

Probability matching/posterior sampling

The idea is to sample $\theta_1, \ldots, \theta_n \sim \hat{p}(\theta_1, \ldots, \theta_n)$, take the optimal action according to the current belief, and update the model. This is hard to analyze theoretically, but has strong empirical performance.

In the bandit setting, we always act greedily, so $\hat{p}(\theta_1, \ldots, \theta_n)$ is a distribution over rewards. The MDP analog is the Q-function: we can sample a Q-function Q from $p(Q)$, act according to Q for one episode, then update it. To sample a function, we can use a bootstrap: given a dataset $\mathcal{D}$, resample with replacement to get $\mathcal{D}_1, \ldots, \mathcal{D}_N$. Train each model f_{θ_i} on $\mathcal{D}_i$, and use f_{θ_i} with i selected uniformly at random. You can simulate this by making N "heads" (use one network, plus N copies of the last layer), so you don't have to train N networks separately.

This is a convenient trick because you don't have to change the original reward function. However, setting a good bonus typically performs better.

Information gain

Suppose we want to determine some latent variable z, e.g. the optimal action. Let $\mathcal{H}(\hat{p}(z))$ be the current entropy of our z estimate, and we take some action y and observe y. We want to maximize the drop in entropy, or equivalently the information gain.

$$\text{IG}(z, y \mid a) = \mathbb{E}_y[\mathcal{H}(\hat{p}(z)) - \mathcal{H}(\hat{p}(z) \mid y) \mid a]$$

Example 13.4 (Russo and Van Roy [29])**.** Take $y = r_a$ and $z = \theta_a$. Let $g(a) = \text{IG}(\theta_a, r_a \mid a)$ and $\Delta(a) = \mathbb{E}[r(a^*) - r(a)]$. Choose a according to $\arg\min_a \frac{\Delta(a)^2}{g(a)}$.

In general, it's intractable to use information gain exactly, regardless of what is being estimated, so we'll need to approximate it:

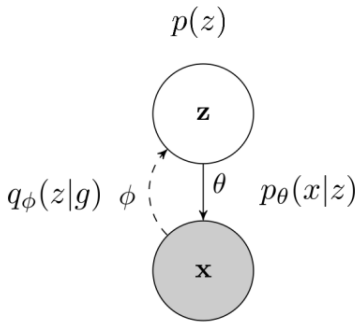


Figure 12: Variational Autoencoder

- Prediction gain: if the density changes a lot, then the last state was novel: $\log_{p_{\theta'}}(s) - \log p_\theta(s)$
- Variational inference: learn about the transitions $p_\theta(s_{t+1} \mid s_t, a_t)$, and write the information gain as $D_{KL}(p(\theta \mid h, s_t, a_t, s_{t+1}) \parallel p(\theta \mid h))$, where h denotes the history.

There's some more literature on this, but I was too tired to listen to the rest of the lecture.

14 Exploration (10/09)

The videos begin here.

Question 14.1. What if we want to recover diverse behavior without any reward function? This is good for learning skills without supervision and using them to accomplish goals, learning subskills for hierarchical RL, and exploring the space of possible behaviors.

For example, you might have a general purpose robot which you put in your home, and want it to be able to prepare for an unknown future goal (e.g. wash the dishes).

First, some math review:

- **Entropy** $\mathcal{H}(p(x)) = -\mathbb{E}_{x\sim p} \log p(x)$, intuitively, measures how "broad" $p(x)$ is
- **Mutual information**

$$\mathcal{I}(x;y) = D_{KL}(p(x,y) \parallel p(x)p(y)) = \mathbb{E}_{(x,y)\sim p} \log \frac{p(x,y)}{p(x)p(y)} = \mathcal{H}(p(y)) - \mathcal{H}(p(y \mid x))$$

  Intuitively, if x and y are independent, then the joint distribution and product of marginals are the same, and the KL divergence is 0. As they grow more dependent, the joint distribution gets farther away, and the KL divergence changes accordingly.

  For example, **empowerment** is $\mathcal{I}(s_{t+1}; a_t) := \mathcal{H}(s_{t+1}) - \mathcal{H}(s_{t+1} \mid a_t)$.

We'll use $\pi(s) = p_{\pi_\theta}(s)$ to denote the state marginal.

In the example above, maybe we'll pass in some images x and x_g representing the current and goal states (your dishes are in a pile, or they are cleaned at neatly organized). The RL agent will have some latent representations of these images z and z_g, as given by a VAE (see Figure 12).

The idea is that we'll provide it some input x, it'll construct a latent representation z, and through unsupervised learning generate its own goals z_{g_i}. This will give something similar to Algorithm 16. We take $w(\overline{x}) = p_\theta(\overline{x})^\alpha$ for some $\alpha \in [-1, 0)$ (reusing our idea of counts in the past).

The idea for weighting is that it will help us skew our samples toward less-used latent representations. (Otherwise, our model will do well at picking up a cup, for example, generate more images of picking up a cup, and not learn how to do anything else.) The key result is that $\alpha \in [-1, 0)$ will result in an increase in the entropy $\mathcal{H}(p_\theta(x))$.

Algorithm 16 Sketch: Learning without rewards

1: **for** repeat K times **do**
2: Propose goal: $z_g \sim p(z)$, $x_g \sim p_\theta(\cdot \mid z_g)$
3: Run a goal-conditioned policy $\pi(a \mid x, x_g)$, which ultimately reaches $\overline{x}$
4: Use data to update π
5: $\theta, \varphi \leftarrow \arg\max_{\theta,\varphi} \mathbb{E}[w(\overline{x}) \log p(\mathbf{x})]$
6: **end for**

Skew-fit (sort of) maximizes $\mathcal{H}(p(G))$, as we want good state coverage. RL trains $\pi(a \mid S, G)$ to reach G, i.e. $p(G \mid S)$ becomes more deterministic, or effective goal reaching. Intuitively, this means that we are maximizing $\mathcal{H}(p(G)) - \mathcal{H}(p(G \mid S)) = \mathcal{I}(S; G)$.

Distribution matching

In the typical formulation, we want to incentivize our policy $\pi(a \mid s)$ to explore diverse states, so we can add a bonus $\hat{r}(s) = r(s) - \log p_\pi(s)$.

Another problem worth considering is matching state marginals: we want to learn $\pi(a \mid s)$ to minimize $D_{KL}(p_\pi(s) || p^*(s))$; this seems fairly similar conceptually, so maybe it's possible to apply the same ideas from intrinsic motivation.

Naively, We can set $\hat{r}(s) = \log p^*(s) - \log p_\pi(s)$. One can notice that the RL objective is exactly the KL-divergence, since we are taking the expectation over p_π. So this doesn't perform marginal matching for the reason that p_π is dependent on π. We can apply a simple fix to produce Algorithm 17. One can view it as a Nash equilibrium to a two-player game between π^k and p_{π^k} (supposedly the mixture of π^k's produces an equilibrium by a well-known result in game theory).

He says so thing about why π jum but I don't quite get it.

what does mean?

Algorithm 17 State Marginal Matching

1: learn $\pi^k(a \mid s)$ to maximize $\mathbb{E}_\pi[\hat{r}^k(s)]$
2: update $p_{\pi^k}(s)$ to fit all states seen so far
3: return $\pi^*(a \mid s) = \sum_k \pi^k(a \mid s)$

Question 14.2. Why is state coverage a good objective?

Answer. Suppose you have an adversary that will choose the worst possible goal G. It is possible to show that the best possible goal distribution to use during training is $p(G) = \arg\max_p \mathcal{H}(p(G))$, i.e. uniform. In a similar vein, we should try to visit all states uniformly.

Learning diverse skills

Suppose you have several policies $\pi(a \mid s, z)$ indexed by the task index z. (Note that tasks are a strict superset of goals: not all behaviors can be captured by goal-reaching, e.g. eating a candy

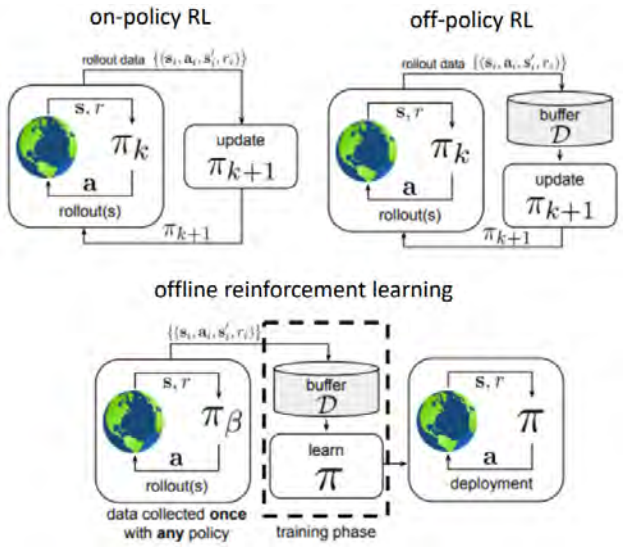


Figure 13: Overview of RL methods

while avoiding lava.) Intuitively, then, different skills should visit different state space regions, not just individual states.

You can implement this with

$$\pi(a \mid s, z) = \arg\max_{\pi} \sum_{z} \mathbb{E}_{s \sim \pi(\cdot|z)}[r(s, z)],$$

where $r(s, z)$ is high at z but low for $z' \neq z$. In particular, taking $r(s, z)$ as a classifier $\log p(z \mid s)$ suffices. The idea is that reward is maximized when classification is easy, i.e. the decision boundary between skills pushes them apart. In an information-theoretic context, we can treat the objective as $I(z, s) = H(z) - H(z \mid s)$: the former is maximized with a uniform prior, and the latter is minimized by maximizing $\log p(z \mid s)$

15 Offline RL (10/16)

The videos begin here.

There is a large gap between the RL tasks we've looked at so far and common supervised learning tasks. For example, if we want to run an image generation task, it is impractical to have an online process (ImageNet is just too massive). Modern ML works well because of massive data and massive models, which is hard for RL as we've seen so far. The question is whether we can develop similarly data-driven RL methods.

Formally, we will have a dataset $\mathcal{D} = \{(s_i, a_i, s_i', r_i)\}$. You don't know how this data is acquired, but there is some underlying π_β with $a \sim \pi_\beta(\cdot \mid s)$. The state marginal $s \sim d^{\pi_\beta}$, and as before $s' \sim p(s' \mid s, a)$, and r is a function of s and a.

Offline RL problems take the same forms that their online varieties do. Given $\mathcal{D}$, we might want to estimate $J(\pi_\beta) = \mathbb{E}_{\tau \sim \pi_\beta}\left[\sum_{t=1}^{T} r(s_t, a_t)\right]$ (off-policy evaluation), or we might want to learn θ to produce the "best possible policy π_θ given $\mathcal{D}$."

Offline RL is distinct from imitation learning (in fact, it is provably better than imitation learning, under some assumptions Kumar et al. [19]). There, we had several trajectories that ended up at a goal and learned a corresponding policy. Here, we can "get order from chaos": we might have several trajectories that don't seem to go anywhere, but by stitching them together/generalizing better, we can do better than the best data example.

The fundamental problem is counterfactual queries. Suppose you have a dataset of driving data. You'll get some errors, e.g. running a red light, but probably nothing like driving off a cliff; we wouldn't know if such an action is good or bad unless we see it in the data. Online algorithms don't have to handle this because they can just try it, while offline methods must account for OOD actions safely. Errors with function approximation in standard RL are much more severe in online RL: in the online setting, we can just tesst things, but generalizability cannot be easily corrected without data.

Recall that the deep learning problem boils down to empirical risk minimization,

$$\theta \leftarrow \arg\min_{\theta} \mathbb{E}_{x\sim p(\cdot), y\sim p(\cdot|x)}[\mathcal{L}(y, f_\theta(x))].$$

"Distribution shift" means that this fails if we instead sample x from $\bar{p} \neq p$. Even if we sample x^* from p, the loss at x^* might still be high: just because it is low in expectation, it might not be low for a particular point. Usually we aren't worried about this because neural nets generalize pretty well, but for example an adversary might choose $x^* \leftarrow \arg\max_x f_\theta(x)$. Similarly, in Q-learning, we take $y(s,a) \leftarrow r(s,a) + \mathbb{E}_{a'\sim\pi_{\text{new}}}[Q(s',a')]$, and the objective is $\min_Q \mathbb{E}_{s,a\sim\pi_\beta}[(Q(s,a) - y(s,a))^2]$. This setup results in:

- We get good accuracy when $\pi_\beta(a \mid s) = \pi_{\text{new}}(a \mid s)$. This isn't great given that we want to improve upon the data rather than imitate it.
- We take $\pi_{\text{new}} = \arg\max_\pi \mathbb{E}_{a\sim\pi(\cdot|s)}[Q(s,a)]$, which is roughly the same problem described above. π_{new} is selected adversarially to produce large Q-values, which results in massive overestimation.

Importance Sampling

Next, we will pivot to an early development in offline RL. Recall that in importance sampling, we can produce an unbiased estimator for $\nabla_\theta J(\theta)$ by using importance sampling with weight

$$\frac{\pi_\theta(\tau)}{\pi_\beta(\tau)} = \frac{p(s_1)}{p(s_1)} \prod_t \frac{p(s_{t+1}|s_t,a_t)\pi_\theta(a_t|s_t)}{p(s_{t+1}|s_t,a_t)\pi_\theta(a_t|s_t)} = \prod_t \frac{\pi_\theta(a_t|s_t)}{\pi_\beta(a_t|s_t)}.$$

However, this is exponential in T, and weights are likely to be degenerate as T becomes large (one weight will blow up, and the rest will vanish). That is, the expectation is always the same, but the variance grows with T.

In the advanced policy lecture, we noted that this product can be split into

$$\prod_{t'=0}^{t-1} \frac{\pi_\theta(a_t|s_t)}{\pi_\beta(a_t|s_t)} \cdot \prod_{t'=t}^{T} \frac{\pi_\theta(a_t|s_t)}{\pi_\beta(a_t|s_t)}.$$

The first term accounts for the difference in $d^{\pi_\beta}(s_t)$ and $d^{\pi_\theta}(s_t)$, and the second term accounts for having the incorrect $\hat{Q}$. Previously, we dropped the first term because we assumed that π_θ and

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{T}\nabla_\theta \gamma^t \log \pi_\theta(\mathbf{a}_{t,i}|\mathbf{s}_{t,i})\left(\prod_{t'=t}^{T}\frac{\pi_\theta(\mathbf{a}_{t',i}|\mathbf{s}_{t',i})}{\pi_\beta(\mathbf{a}_{t',i}|\mathbf{s}_{t',i})}\right)\hat{Q}(\mathbf{s}_{t,i},\mathbf{a}_{t,i}) \leftarrow E_{\pi_\theta}\left[\sum_{t'=t}^{T}\gamma^{t'-t}r_{t'}\right] \approx \sum_{t'=t}^{T}\gamma^{t'-t}r_{t',i}$$

$$\sum_{t'=t}^{T}\left(\prod_{t''=t}^{t'}\frac{\pi_\theta(\mathbf{a}_{t'',i}|\mathbf{s}_{t'',i})}{\pi_\beta(\mathbf{a}_{t'',i}|\mathbf{s}_{t'',i})}\right)\gamma^{t'-t}r_{t',i}$$

Figure 14: Further simplifying the policy gradient. We can change T to t' as the upper limit because actions in the future don't affect rewards in the past.

$$d^{\pi_\beta}(\mathbf{s}',\mathbf{a}')w(\mathbf{s}',\mathbf{a}') = (1-\gamma)\underbrace{p_0(\mathbf{s}')\pi_\theta(\mathbf{a}'|\mathbf{s}')}_{\text{probability of starting in } (\mathbf{s}',\mathbf{a}')} + \gamma\sum_{\mathbf{s},\mathbf{a}}\underbrace{\pi_\theta(\mathbf{a}'|\mathbf{s}')p(\mathbf{s}'|\mathbf{s},\mathbf{a})d^{\pi_\beta}(\mathbf{s},\mathbf{a})}_{\text{probability of transitioning into } (\mathbf{s}',\mathbf{a}')}w(\mathbf{s},\mathbf{a})$$

Figure 15: Zhang et al. [42]

π_β were similar enough. We can't do that anymore in the offline RL setting, but for the sake of explanation we'll do it anyway. See Figure 14. This is still exponential but does a little better.

The idea of the **doubly robust estimator** is to just use the same estimator we just found,

$$V^{\pi_\theta}(s_0) \approx \sum_{t=0}^{T}\left(\prod_{t'=0}^{t}\frac{\pi_\theta(a_{t'}|s_{t'})}{\pi_\beta(a_{t'}|s_{t'})}\right)\gamma^t r_t =: \sum_{t=0}^{T}\left(\prod_{t'=0}^{t}\rho_{t'}\right)\gamma^t r_t = \rho_0(r_0+\gamma(\rho_1(r_1+\cdots))) =: \bar{V}^T,$$

where we've shifted indices for ease of notation. One can observe that $\bar{V}^{T+1-t} = \rho_t(r_t + \gamma\bar{V}^{T-t})$. **Doubly robust estimation** takes $V_{DR}(s) := \hat{V}(s) + \rho(s,a)(r(s,a) - \hat{Q}(s,a))$, so that

$$\bar{V}_{DR}^{T+1-t} = \hat{V}(s_t) + \rho_t(r_t + \gamma\bar{V}_{DR}T - t - \hat{Q}(s_t,a_t)).$$

In **marginalized importance sampling**, you can estimate $J(\theta) \approx \frac{1}{N}\sum_i w(s_i,a_i)r_i$. You can determine $w(s,a)$ by solving some consistency condition, i.e. write down a relationship and corresponding error between w's at current and future states, and estimate the error using samples from the dataset. For example, see Figure 15.

Linear Fitted Value Functions

Let Φ be a feature matrix with dimensions $|S| \times K$, where K is the number of features. With a linear model, we will have:

- The reward model $\Phi w_r \approx r$ where $w_r = (\Phi^\top\Phi)^{-1}\Phi^\top r \in \mathbb{R}^K$.
- The transition model $\Phi P_\Phi \approx P^\pi\Phi$, where $P^\pi \in \mathbb{R}^{|S|\times|S|}$ is the real transition matrix and $P_\Phi = (\Phi^\top\Phi)^{-1}\Phi P^\pi\Phi \in \mathbb{R}^{K\times K}$.
- The value function $V^\pi \approx V_\Phi^\pi = \Phi w_V$.

Then the Bellman backup $V^\pi = r + \gamma P^\pi V^\pi$ implies $V^\pi = (I - \gamma P^\pi)^{-1}r$, or equivalently $w_V = (I - \gamma P_\Phi)^{-1}w_r$ in feature space. Substituting, we get

$$w_v = (\Phi^\top\Phi - \gamma\Phi^\top P^\pi\Phi)^{-1}\Phi^\top r,$$

called **least squares temporal difference (LSTD)**. On its own, this runs into some problems.

- We can't use this directly in practice unless we know P^π. But we can estimate this with samples; suppose we have $\mathcal{D} = \{(s_i, a_i, r_i, s_i')\}$. Then everything works exactly the same way (now with some sampling error), where instead Φ' has dimensions $|\mathcal{D}| \times K$ with $\Phi_i' = \varphi(s_i')$.
- We can't use this method for policies other than the one that collected the data, since P^π is dependent on π.

We can resolve both of these with **least-squares policy iteration (LSPI)** by estimating a Q-function instead of a value function. Now we have $\Phi \in \mathbb{R}^{|S||A| \times K'}$, where $\Phi_i' = \varphi(s_i', \pi(s_i'))$. Then computing $w_Q = (\Phi^\top \Phi - \gamma \Phi^\top \Phi')^{-1} \Phi^\top r$ suffices, where we update $\pi_{k+1}(s) = \arg\max_a \phi(s,a) w_Q(\pi_k)$ and set $\Phi_i' = \varphi(s_i', \pi_{k+1}(s_i'))$.

In practice, this doesn't work well because in the argmax step, we experience distributional shift.

16 Offline RL (10/18)

The videos begin here.

Recall that with our update step

$$\pi_{\text{new}} = \arg\max_\pi \mathbb{E}_{a \sim \pi(\cdot|s)}[Q(s,a)],$$

our algorithm will make adversarial updates to fool our Q-function to take actions with erroneously high Q-values. To make practical RL algorithms, we have to resolve this distributional shift problem.

Implicit Policy Constraints

One very old way of dealing with this is to constrain your policy, $D_{\text{KL}}(\pi \| \pi_\beta) \leq \varepsilon$ in the argmax. However, this has several issues:

- We don't know the behavior policy $\pi_\beta(a|s)$, so we have to be careful in our estimation of the KL divergence term.
- It may not be pessimistic enough. Suppose that you have something true in expectation; that doesn't mean that you can expect the same for any individual data point. Likewise, we are stuck trying to strike a balance between moving π close enough to β while also far enough that it reaches convergence.
- It may be too pessimistic. Suppose π_β is uniformly random, i.e. any action is equally likely. Then the KL condition means that we should keep π roughly uniform, even though it probably isn't a good choice.

Another idea to explicitly constrain our policy is to apply a support constraint, e.g. $\pi(a|s) \geq 0$ only if $\pi_\beta(a|s) \geq \varepsilon$. As in Figure 16, this is a bit closer to what we want, but hard to implement.

So we'll try to elaborate some more on the first method. One can prove through duality that

$$\pi^*(a|s) = \frac{1}{Z(s)} \pi_\beta(a|s) \exp\left(\frac{1}{\lambda} A^\pi(s,a)\right),$$

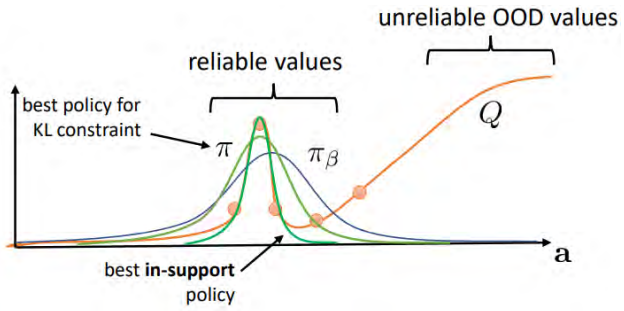


Figure 16: Explicit policy constraint methods

and as such we can approximate

$$\pi_{\text{new}}(a|s) = \arg\max_{\pi} \mathbb{E}_{(s,a)\sim\pi_\beta}\left[\log \pi(a|s) \frac{1}{Z(s)} \exp\left(\frac{1}{\lambda} A^{\pi_{\text{old}}}(s,a)\right)\right].$$

We can compute the expectation through sampling from the dataset, and the critic can be used to compute the advantage. Peng et al. [27]

At a high level, this is essentially behavioral cloning, where we choose the good actions more often than the bad ones.

However, this runs into two sources of distributional shifts while querying advantage values:

- In the target value while computing the expectation over π_θ in our update $Q(s,a) = r(s,a) + \mathbb{E}_{a'\sim\pi_{\text{new}}}[Q(s,a)]$ via the critic gradient step with

$$\mathcal{L}_C(\varphi) = \mathbb{E}_{(s,a,s')\sim\mathcal{D}}\left[(Q_\varphi(s,a) - (r(s,a) + \gamma \mathbb{E}_{a'\sim\pi_\theta(a'|s')}[Q_\varphi(s',a')]))^2\right].$$

  It will obey "the constraint" by the end of training with a good choice of λ, but not necessarily during training.

  whatever th is?

- When estimating the advantage, we query for actions from π_θ.

Question 16.1. Is it possible to avoid all OOD actions in the Q update?

Answer. It is insufficient to update

$$V \leftarrow \arg\min_V \frac{1}{N} \sum_{i=1}^{N} \ell(V(s_i), Q(s_i,a_i)),$$

because $a_i \sim \mathcal{D}$ comes from π_β, not π_{new}.

However, notice that in the case of very large $|S|$ or continuous states, really we have a distribution over $V(s)$ when we consider the possibility of similar state-action pairs. In the case of MSE loss, we are asking for the expectation over actions. We can avoid problems by instead considering the value of the best policy supported by the data via the **expectile loss**

$$\ell_\tau(x) = \begin{cases} (1-\tau)x^2 & x > 0 \\ \tau x^2 & \text{else.} \end{cases}$$

The idea is that for $\tau > \frac{1}{2}$, the loss penalizes negative errors more than positive ones. It is possible to show that for a sufficiently large τ, the update is simply

$$V(s) \leftarrow \max_{a \in \Omega(s)} Q(s,a),$$

where $\Omega(s) := \{a : \pi_\beta(a|s) \geq \varepsilon\}$ is the support over actions. The algorithm, called IQL Kostrikov et al. [17], produces the implicit policy $\pi_{\text{new}}(a|s) = \mathbb{1}\left\{a = \arg\max_{a \in \Omega(s)} Q(s,a)\right\}$.

Note. It is also possible to constrain the policy explicitly. We can modify the actor objective as

$$\theta \leftarrow \arg\max_\theta \mathbb{E}_{s\sim\mathcal{D}}[\mathbb{E}_{a\sim\pi_\theta(\cdot|s)}[Q(s,a) + \lambda\pi_\beta(a|s)] + \lambda\mathcal{H}(\pi(a|s))],$$

where λ is a Lagrange multiplier. This setup produces $D_{\text{KL}}(\pi\|\pi_\beta) = -\mathbb{E}_\pi[\log\pi_\beta(a|s)] - \mathcal{H}(\pi)$.
Or you can modify the reward function directly, $\bar{r}(s,a) = r(s,a) - D_{\text{KL}}(\pi\|\pi_\beta)$ (or another divergence metric of your choice).
In practice, most of these methods don't work very well.

Conservative Q-Learning

Another way to mitigate adversarial updates in the Bellman backup is to directly fix the backup. An idea to directly counteract this behavior is

$$\hat{Q}^\pi = \arg\min_Q \left(\underbrace{\max_\mu \alpha\mathbb{E}_{s\sim D, a\sim\mu(\cdot|s)}[Q(s,a)]}_{\text{always pushes down Q-values}} + \underbrace{\mathbb{E}_{(s,a,s')\sim\mathcal{D}}\left[(Q(s,a) - (r(s,a) + \mathbb{E}_\pi[Q(s',a')]))^2\right]}_{\text{regular objective}} \right)$$

It is possible to show that $\hat{Q}^\pi \leq Q^\pi$ for large enough α, so it is too pessimistic. So we will resolve this by pushing back up on samples in the data, Kumar et al. [18]

$$\hat{Q}^\pi = \arg\min_Q \Bigg(\max_\mu \alpha\mathbb{E}_{s\sim D, a\sim\mu(\cdot|s)}[Q(s,a)] - \alpha\mathbb{E}_{(s,a)\sim\mathcal{D}}[Q(s,a)] + \underbrace{\mathbb{E}_{(s,a,s')\sim\mathcal{D}}\left[(Q(s,a) - (r(s,a) + \mathbb{E}_\pi[Q(s',a')]))^2\right]}_{\mathcal{L}_{CQL}(\hat{Q}^\pi)} \Bigg)$$

The idea is that if the actions are from the dataset, the two terms should balance out. If they are out-of-distribution, though, they will be penalized, and the more actions will be pushed into the distribution.
In this setting, it is possible to prove that $\mathbb{E}_{a\sim\pi(\cdot|s)}[\hat{Q}^\pi(s,a)] \leq \mathbb{E}_{a\sim\pi(\cdot|s)}[Q^\pi(s,a)]$ for all $s \in \mathcal{D}$. In the algorithm, you repeatedly update $\hat{Q}^\pi$ wrt $\mathcal{L}_{CQL}$ using $\mathcal{D}$, then update π. If the actions are discrete, then you can just use the greedy policy

$$\pi(a|s) = \mathbb{1}\left\{a = \arg\max_a \hat{Q}(s,a)\right\}.$$

In the continuous case, you can use an actor-critic method,

$$\theta \leftarrow \theta + \alpha \nabla_\theta \sum_i \mathbb{E}_{a \sim \pi_\theta(a|s_i)}[\hat{Q}(s_i, a)].$$

One might want to subtract a regularizer $R(\mu)$; a common choice is $R = \mathbb{E}_{s \sim \mathcal{D}}[\mathcal{H}(\mu(\cdot|s))]$. In this case, we can compute the optimal choice as $\mu(a|s) \propto \exp Q(s,a)$, or $\mathbb{E}_{a \sim \mu(\cdot|s)}[Q(s,a)] = \log \sum_a \exp Q(s,a)$. In the discrete case, these can be computed directly; in the continuous case, you can use importance sampling.

Model-Based Offline RL

An idea to punish the policy for exploring is to impose a penalty $\tilde{r}(s,a) = r(s,a) - \lambda u(s,a)$ Yu et al. [40]. In fact, it is possible to prove some theoretical guarantees, where the true return of the policy trained under the model improves over the behavior policy, and moreover performs as well as the optimal policy minus a constant.

Another idea is to apply CQL, where instead of the $\max_\mu$ term that handles adversarial inputs, we can just take the expectation over state-action tuples from our model Yu et al. [41].

It is possible to plan while handling ood actions without explicitly constructing a policy. One idea is the trajectory transformer, where we train a joint state-action model that provides probabilities $p_\beta(\tau) = p_\beta(s_1, a_1, \ldots, s_T, a_T)$. We will optimize for a plan that has high probability under p_β. For an offline model, we can afford to use a more expressive model class, since we don't have to update our model between trials. The most powerful sequence density estimator today is a transformer. The idea is to use a sequence model over dimensions of states and actions, i.e. at each time step it predicts the corresponding dimensions. Janner et al. [14]

Summary

17 RL Theory (10/23)

The videos begin here.

Suppose we're using an algorithm with N samples and k iterations. Can we provide some guarantee $\left\|\hat{Q}_k - Q^*\right\| \leq \varepsilon$ with probability at least $1 - \delta$ for sufficiently large N? Alternatively, can we bound $\|Q^{\pi_k} - Q^*\|$? (This is different, since $\hat{Q}_k$ might be an erroneous estimator for it.) Or can we give a bound on regret over time?

The purpose of theoretical analysis is not to provide any provable guarantees that work well in practice; typically strong, unrealistic assumptions are required. At best, it is a rough guide to what might happen. This provides some qualitative pros and cons of certain algorithms — sometimes, they are very useful in guiding our choices of parameters, and so on.

First, we state some useful results we will use throughout the lecture:

Theorem 17.1 (Hoeffding's inequality). *Let $X_1, \ldots, X_n$ be iid with mean μ, and suppose that $X_i \in$*

If you want to *only* train offline...

Conservative Q-learning	+ just one hyperparameter	+ well understood and widely tested
Implicit Q-learning	+ more flexible (offline + online)	- more hyperparameters

If you want to *only* train offline and finetune online

Advantage-weighted actor-critic (AWAC)	+ widely used and well tested
Implicit Q-learning	+ seems to perform much better!

If you have a good way to train models in your domain

COMBO	+ similar properties as CQL, but benefits from models
	- not always easy to train a good model in your domain!
Trajectory transformer	+ very powerful and effective models
	- extremely computationally expensive to train and evaluate

Figure 17: Which offline RL algorithm should I use?

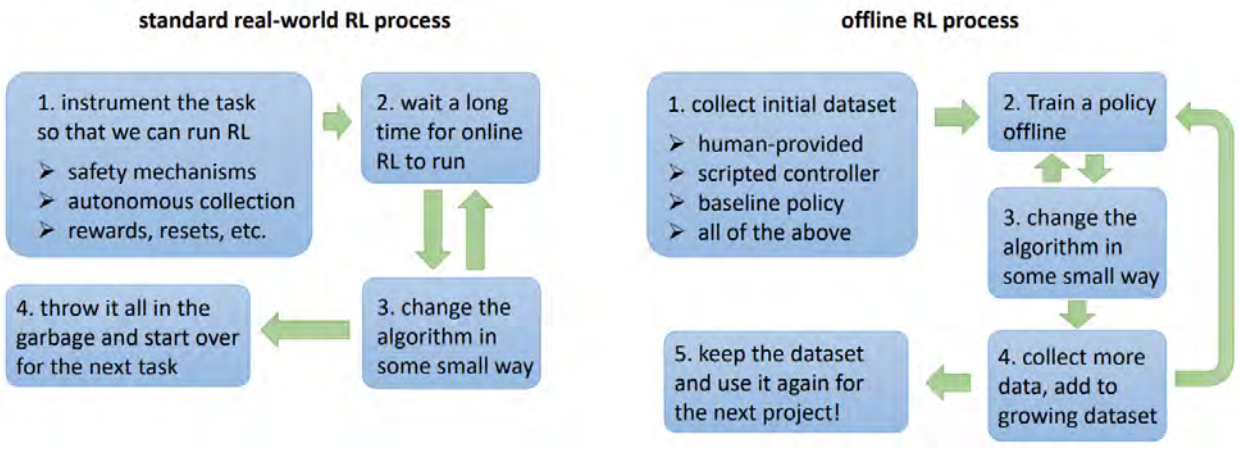


Figure 18: Why offline RL works well

$[a,b]$ w.p. 1. Let $\bar{X}_n = \frac{1}{n}\sum_{i=1}^n X_i$. Then

$$\mathbb{P}(\bar{X}_n \geq \mu + \varepsilon) \leq \exp\left(-\frac{2n\varepsilon^2}{(b-a)^2}\right).$$

Equivalently, if we want this probability to be δ, we can take $\varepsilon \leq \frac{b-a}{\sqrt{2n}}\sqrt{\log\frac{2}{\delta}}$ or $n \leq \frac{(b-a)^2}{2\varepsilon^2}\log\frac{2}{\delta}$.

Theorem 17.2 (Concentration for discrete distributions). *Let $Z \sim \text{Categorical}(\mathbf{q})$ take values in $[d]$. That is, $q_i = \mathbb{P}(Z = i)$. Let our empirical estimate from N samples be $\hat{q}_i = \frac{1}{N}\sum_{j=1}^N \mathbb{1}\{z_j = i\}$. Then*

$$\mathbb{P}\left(\|\hat{\mathbf{q}} - \mathbf{q}\|_2 \geq \frac{1}{\sqrt{N}} + \varepsilon\right) \leq \exp(-N\varepsilon^2)$$
$$\mathbb{P}\left(\|\hat{\mathbf{q}} - \mathbf{q}\|_1 \geq \sqrt{d}\left(\frac{1}{\sqrt{N}} + \varepsilon\right)\right) \leq \exp(-N\varepsilon^2).$$

This time, this is equivalent to $\varepsilon \leq \frac{1}{\sqrt{N}}\sqrt{\log\frac{1}{\delta}}$ or $N \leq \frac{1}{\varepsilon^2}\log\frac{1}{\delta}$.

Model-based RL

Suppose we have N samples $s' \sim p(\cdot|s,a)$ for all s,a, and we define $\hat{p}(s'|s,a) = \frac{\#(s,a,s')}{N}$, and we'll estimate $\hat{Q}^\pi$ via $\hat{p}$. We might be interested in the following questions: if you take sufficiently large N, can we achieve the following worst-case bounds with probability at least $1-\delta$?

- How bad is our estimate, i.e. can we bound $\left\|Q^\pi(s,a) - \hat{Q}^\pi(s,a)\right\|_\infty := \max_{s,a}\left|Q^\pi(s,a) - \hat{Q}^\pi(s,a)\right| \leq \varepsilon$?
- How good is our optimal Q-function learned under $\hat{p}$, i.e. can we bound $\left\|Q^*(s,a) - \hat{Q}(s,a)\right\|_\infty \leq \varepsilon$?
- How good is the resulting policy, i.e. can we bound $\left\|Q^*(s,a) - Q^{\hat{\pi}}(s,a)\right\|_\infty \leq \varepsilon$?

It turns out that the first question applies very easily to the other ones, so we'll focus mostly on it. Recall that we can write updates as $Q^\pi = r + \gamma P V^\pi$, where $Q^\pi, r \in |S||A|$, $P \in |S||A| \times |S|$, and $V^\pi \in |S|$. We can also write $V^\pi = \Pi Q^\pi$ for some $\Pi \in |S| \times |S||A|$. Equivalently, if we let $P^\pi = P\Pi$, then we can rewrite this as $Q^\pi = (I - \gamma P^\pi)^{-1} r$.

Lemma 17.3 (Simulation lemma). $Q^\pi - \hat{Q}^\pi = \gamma(I - \gamma\hat{P}^\pi)^{-1}(P - \hat{P})V^\pi$.

Proof.

$$\begin{aligned}
Q^\pi - \hat{Q}^\pi &= Q^\pi - (I - \gamma\hat{P}^\pi)^{-1} r \\
&= (I - \gamma\hat{P}^\pi)^{-1}(I - \gamma\hat{P}^\pi)Q^\pi - (I - \gamma P^\pi)^{-1}(I - \gamma P^\pi)Q^\pi \\
&= \gamma(I - \gamma\hat{P}^\pi)^{-1}(P^\pi - \hat{P}^\pi)Q^\pi \\
&= \gamma(I - \gamma\hat{P}^\pi)^{-1}(P - \hat{P})V^\pi.
\end{aligned}$$

■

Lemma 17.4. *For any $v \in |S||A|$, $\left\|(I-\gamma P^\pi)^{-1}v\right\|_\infty \leq \frac{\|v\|_\infty}{1-\gamma}$.*

Proof. Let $w := (I-\gamma P^\pi)^{-1}v$, so that

$$\|v\|_\infty = \|(I-\gamma P^\pi)w\|_\infty \geq \|w\|_\infty - \gamma \|P^\pi w\|_\infty \geq (1-\gamma)\|w\|_\infty.$$

Rearranging completes the proof. ■

Proposition 17.5. *There exists a constant c for which*

$$\left\|Q^\pi - \hat{Q}^\pi\right\|_\infty \leq \frac{\gamma}{(1-\gamma)^2} c\sqrt{\frac{|S|\log 1/\delta}{N}}.$$

Proof. Combining the two lemmas,

$$\begin{aligned}\left\|Q^\pi - \hat{Q}^\pi\right\|_\infty &= \left\|\gamma(I-\gamma\hat{P}^\pi)^{-1}(P-\hat{P})V^\pi\right\|_\infty \\ &\leq \frac{\gamma}{1-\gamma}\left\|(P-\hat{P})V^\pi\right\|_\infty \\ &\leq \frac{\gamma}{1-\gamma}\left(\max_{s,a}\left\|P(\cdot|s,a) - \hat{P}(\cdot|s,a)_1\right\|_1\right)\|V^\pi\|_\infty.\end{aligned}$$

Now, we bound each term: there exists a constant c satisfying

$$\left\|\hat{P}(s'|s,a) - P(s'|s,a)\right\|_1 \leq c\sqrt{\frac{|S|\log 1/\delta}{N}},$$

and if $R_{\max} = 1$, then

$$\|V^\pi\|_\infty \leq \sum_{t=0}^{\infty}\gamma^t R_{\max} = \frac{1}{1-\gamma}.$$

Combining, the desired result follows. ■

In particular, the $(1-\gamma)^{-2}$ component means that the error accumulates quadratically in the horizon. Some corollaries:
If we want to bound the error on our estimated Q-value under the optimal policy in each case,

$$\left\|Q^* - \hat{Q}^*\right\|_\infty = \left\|\sup_\pi Q^\pi - \sup_\pi \hat{Q}^\pi\right\| \leq \sup_\pi \left\|Q^\pi - \hat{Q}^\pi\right\|_\infty \leq \varepsilon.$$

If we want to bound the error on the true Q-values,

$$\left\|Q^* - Q^{\hat{\pi}^*}\right\|_\infty = \left\|Q^* - \hat{Q}^{\pi^*}\right\|_\infty + \left\|Q^{\hat{\pi}^*} - \hat{Q}^{\pi^*}\right\|_\infty \leq \varepsilon + \varepsilon = 2\varepsilon$$

from our two preceding results.

$$|\hat{T}Q(s,a) - TQ(s,a)| = |\hat{r}(s,a) - r(s,a) + \gamma(E_{\hat{P}(s'|s,a)}[\max_{a'} Q(s',a')] - E_{P(s'|s,a)}[\max_{a'} Q(s',a')])|$$
$$\leq \underbrace{|\hat{r}(s,a) - r(s,a)|}_{\text{estimation error of continuous random variable}} + \gamma \underbrace{|(E_{\hat{P}(s'|s,a)}[\max_{a'} Q(s',a')] - E_{P(s'|s,a)}[\max_{a'} Q(s',a')])|}_{\text{just use Hoeffding's inequality directly!}}$$

$$|\hat{r}(s,a) - r(s,a)| \leq 2R_{\max}\sqrt{\frac{\log 1/\delta}{2N}}$$

$$\sum_{s'}(\hat{P}(s'|s,a) - P(s'|s,a)) \max_{a'} Q(s',a')$$
$$\leq \sum_{s'} |\hat{P}(s'|s,a) - P(s'|s,a)| \max_{s',a'} Q(s',a')$$
$$= ||\hat{P}(\cdot|s,a) - P(\cdot|s,a)||_1 ||Q||_\infty$$
$$\leq c||Q||_\infty \sqrt{\frac{\log 1/\delta}{N}}$$

Figure 19: Sampling error

Model-free RL

Let $TQ = r + \gamma P \max_a Q(\cdot, a)$ denote the Bellman operator, so that $Q_{k+1} = TQ_k$. Q-iteration will look like $\hat{Q}_{k+1} \leftarrow \arg\min_{\hat{Q}} \left\|\hat{Q} - \hat{T}\hat{Q}_k\right\|$. There are two sources of error, namely in sampling error in $\hat{T}$ (which is built on approximates $\hat{r} = \frac{1}{N(s,a)}\sum_i \mathbb{1}\{(s_i,a_i) = (s,a)\}r_i$ and $\hat{P} = \frac{N(s,a,s')}{N(s,a)}$) and the approximation error in the minimization $\hat{Q}_{k+1} \neq \hat{T}\hat{Q}_k$.

The analysis of sampling error is in Figure 19. The conclusion is that there exist constants c_1, c_2 for which

$$\left\|\hat{T}Q - TQ\right\|_\infty \leq 2R_{\max}c_1\sqrt{\frac{\log|S||A|/\delta}{2N}} + c_2\left\|Q\right\|_\infty\sqrt{\frac{\log|S|/\delta}{N}}.$$

We will assume that the approximation error is bounded, $\left\|\hat{Q}_{k+1} - T\hat{Q}_k\right\|_\infty \leq \varepsilon_k$. Then

$$\begin{aligned}\left\|\hat{Q}_k - Q^*\right\|_\infty &\leq \left\|\hat{Q}_k - T\hat{Q}_{k-1}\right\|_\infty + \left\|T\hat{Q}_{k-1} - Q^*\right\|_\infty \\ &\leq \varepsilon_{k-1} + \left\|T\hat{Q}_{k-1} - TQ^*\right\|_\infty \\ &\leq \varepsilon_{k-1} + \gamma\left\|\hat{Q}_{k-1} - Q^*\right\|_\infty.\end{aligned}$$

In the second line, we note that $TQ^* = Q^*$, and in the third line, T is a γ-contraction. By induction,

$$\left\|\hat{Q}^k - Q^*\right\|_\infty \leq \sum_{i=0}^{k-1}\gamma^i\varepsilon_{k-i-1} + \gamma^k\left\|\hat{Q}_0 - Q^*\right\|_\infty.$$

In the limit,

$$\lim_{k\to\infty}\left\|\hat{Q}_k - Q^*\right\|_\infty \leq \sum_{i=0}^{\infty}\gamma^i\left\|\varepsilon\right\|_\infty = \frac{\left\|\varepsilon\right\|_\infty}{1-\gamma}.$$

18 Variational Inference and Generative Models (10/25)

The videos begin here.

VAE

This is a bit of a break from RL, but many of the same ideas are applicable to latent variable models in model-based RL.

A classic example of a latent variable model is a mixture model; for example, suppose that we collect data x, and it very clearly comes in some clusters. We can let z be a categorical variable representing the clusters, and note that

$$p(x) = \sum_z p(x|z)p(z).$$

For conditional models,

$$p(y|x) = \sum_z p(y|x,z)p(z).$$

In general, latent variable models take the following form: $p(x)$ is complicated (it is hard to find, for example, a Gaussian that fits x well). We'll take some $p(z)$ as an easy distribution to deal with, e.g. a Gaussian, and $p(x|z)$ is another Gaussian with mean $\mu_{\text{nn}}(z)$ and variance $\sigma^2_{\text{nn}}(z)$. (The parameters to the latter Gaussian are probably complicated, but the model itself is quite simple.) Then

$$p(x) = \int p(x|z)p(z)\mathrm{d}z.$$

This works well in RL in settings such as multimodal policies and for model-based RL.

Of course, training this directly is hard to do: if we have some data $\mathcal{D} = \{x_i\}$, then our objective is $\arg\max_\theta \frac{1}{N}\sum_i \log p_\theta(x_i) = \arg\max_\theta \frac{1}{N}\sum_i \log\left(\int p_\theta(x_i|z)p(z)\mathrm{d}z\right)$, and computing the integral each time is intractable. An alternative is to use the **expected log-likelihood**

$$\theta \leftarrow \arg\max_\theta \frac{1}{N}\sum_i \mathbb{E}_{z\sim p(\cdot|x_i)} \log p_\theta(x_i, z).$$

To do so, we must first calculate $p(z|x_i)$, which we do via the variational approximation. Suppose we take $q_i(z) \approx \mathcal{N}(\mu_i, \sigma_i)$. Then

$$\begin{aligned}
\log p(x_i) &= \log \int p(x_i|z)p(z)\mathrm{d}z \\
&= \log \int p(x_i|z)p(z)\frac{q_i(z)}{q_i(z)}\mathrm{d}z \\
&= \log \mathbb{E}_{z\sim q_i}\left[\frac{p(x_i|z)p(z)}{q_i(z)}\right] \\
&\geq \mathbb{E}_{z\sim q_i}\left[\log \frac{p(x_i|z)p(z)}{q_i(z)}\right] \\
&= \mathbb{E}_{z\sim q_i}[\log p(x_i|z) + \log p(z)] + \mathcal{H}(q_i) \\
&=: \mathcal{L}_i(p, q_i).
\end{aligned}$$

The benefit is that every term is tractable. We claim that it is a good idea to maximize this lower bound, which "pushes up" on $\log p(x_i)$, but without additional analysis is insufficient to deduce that it is a good proxy.

Remark. The objective given by $\mathcal{L}_i$ can be thought of as the sum of two terms: maximizing the first term strives to make $\mathcal{H}(q_i)$ small (we want z to be very close to the optimum), while the second term regularizes it.

Some math verifies that

$$\log p(x_i) = D_{\text{KL}}(q_i(x_i) \| p(z|x_i)) + \mathcal{L}_i(p, q_i).$$

This is another method of proving the lower bound we saw above, but more importantly, shows that minimizing the KL divergence is a good way of tightening this bound. Equivalently, maximizing $\mathcal{L}_i(p, q_i)$ w.r.t. q_i minimizes the KL divergence.

An issue with optimizing q_i in this way is that making a μ_i, σ_i for every x_i is that the number of parameters scales linearly with the amount of data. So we can instead learn a network such that $q_i(z) = q(z|x_i) \approx p(z|x_i)$.

To recap, our **amortized variational inference** will have two networks: the generative model we are trying to learn $z \mapsto p_\theta(x|z)$ and the inference network $x \mapsto q_\varphi(z|x) = \mathcal{N}(\mu_\varphi(x), \sigma_\varphi(x))$.

Algorithm 18 Variational Inference

1: **for** $x_i \in \mathcal{D}$ **do**
2: sample $z \sim q_\varphi(z|x_i)$
3: $\nabla_\theta \mathcal{L} \approx \nabla_\theta \log p_\theta(x_i|z)$
4: $\theta \leftarrow \theta + \alpha \nabla_\theta \mathcal{L}$
5: $\varphi \leftarrow \varphi + \alpha \nabla_\varphi \mathcal{L}$
6: **end for**

Computing gradients through θ is easy, but it is not quite so obvious for φ. Here we outline two methods:

- **Policy gradient:** We can write

$$\mathcal{L}_i = \underbrace{\mathbb{E}_{z \sim q_\varphi(z|x_i)}[r(x_i, z)]}_{J(\varphi)} + \mathcal{H}(q_\varphi(z|x_i)),$$

  where $r(x_i, z) = \log p_\theta(x_i|z) + \log p(z)$ does not depend on φ. In particular, we can draw many samples $\{z_j\}$ and use the policy gradient,

$$\nabla J(\varphi) \approx \frac{1}{M} \sum_j \nabla_\varphi \log q_\varphi(z_j|x_i) r(x_i, z_j).$$

  This works fine, as drawing samples is very efficient. The issue in practice is that policy gradient has high variance, so you need a lot of samples to get a good estimate.

- **Reparameterization trick:** In RL, we used the policy gradient because we couldn't differentiate through unknown dynamics; that doesn't exist here, and in fact there is a simple way to improve this. Note that $q_\varphi(z|x) = \mathcal{N}(\mu_\varphi(x), \sigma_\varphi(x))$ implies $z = \mu_\varphi(x) + \varepsilon \sigma_\varphi(x)$, where $\varepsilon \sim \mathcal{N}(0, 1)$ is independent on φ. Now, we can differentiate as usual,

$$\nabla_\varphi J(\varphi) \approx \nabla_\varphi r(x_i, \mu_\varphi(x_i) + \varepsilon_j \sigma_\varphi(x_i)).$$

(We couldn't do this in RL because r wasn't required to be differentiable! But we have a differentiable closed form in this case.) With some math, we compute

$$\mathcal{L}_i \approx \log p_\theta(x_i \mid \mu_\varphi(x_i) + \varepsilon\sigma_\varphi(x_i)) - D_{\mathrm{KL}}(q_\varphi(z|x_i)\|p(z)).$$

In practice, we often only need $M = 1$ sample, and the implementation is very straightforward. However, it only works with continuous latent variables.

The form above is exactly the objective used for the **variational autoencoder**, $\max_{\theta,\varphi} \frac{1}{N}\sum_i \mathcal{L}_i$.

Example 18.1. We can regard z as a representation of x. Then in general, z is more useful in representing the state than the individual pixels of s.

For example, in a video game, your character will consist of many pixels that move together, so they will be highly correlated. The latent representation, which consists of independent Gaussians, is better at capturing good features.

A sample algorithm might look like the following:

Algorithm 19 Sample VAE for RL Algorithm

1: **for** as long as you feel like **do**
2: Collect transition (s, a, s', r), add it to $\mathcal{R}$
3: Update $p_\theta(s|z)$ and $q_\varphi(z|s)$ with batch from $\mathcal{R}$ ▷ pretrain VAE with Atari dataset
4: Update $Q(z, a)$ with batch from $\mathcal{R}$
5: **end for**

Note. Conditional models work in exactly the same way, i.e. the decoder now accepts x_i as an input and produces outputs $p_\theta(y_i|x_i, z)$. At test time, we sample $z \sim p(\cdot|x_i)$ and $y \sim p(\cdot|x_i, z)$.

State space models

Our POMDP now has states replaced with their latent representations. That is, we will learn several things, where we are using a VAE with data $\mathbf{o}_{1:T} = (o_1, \ldots, o_T)$ and latent representation $\mathbf{z}_{1:T} = (z_1, \ldots, z_T)$. For example, the following setup might work well, where we make z_t's conditionally independent with respect to $\mathbf{o}_{1:t}$:

- Transition dynamics $p(z_{t+1}|z_t, a_t)$, or $p(\mathbf{z}) = p(z_1)\prod_t p(z_{t+1}|z_t, a_t)$ (we take $z_1 \sim \mathcal{N}(0, I)$).
- Encoder $p_\theta(\mathbf{o}|\mathbf{z}) = \prod_t p(o_t|z_t)$.
- Decoder $q_\varphi(\mathbf{z}|\mathbf{o}) = \prod_t q_\varphi(z_t|\mathbf{o}_{1:t})$.

19 Control as Inference (10/30)

The videos begin here. The lecture is based on Levine [20].

Now we combine our ideas from the last few lectures. The idea is that one can use probabilistic inference to derive RL and optimal control, and we can design RL algorithms based on these ideas.

We will avoid delving too deeply into philosophy and define a rational decision-maker as one whose behaviors can be expressed with well-defined utilities. We will use this framework, combined with optimal control, as a model for human behavior.

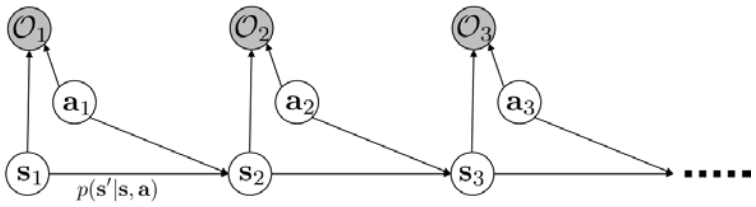


Figure 20: Probabilistic graphical model for near-optimal control

One challenge is that RL doesn't know what non-optimality is. For example, if you have a start and end point, a human might not walk in a perfectly straight line from the start to the end. To the human, it doesn't really matter how they get to the goal — taking one wrong turn might not matter that much, and their behavior is stochastic. Nonetheless, generally "good" behavior is still the most likely.

Control as Exact Inference

We introduce a probabilistic graphical model of decision making that produces near-optimal behavior. The solution to near-optimal control will look similar to that for RL, but not exactly the same.

We will construct independent optimality random variables $\mathcal{O}_t \sim \text{Bern}(\exp r(s_t, a_t))$, where all rewards are negative. (With finite rewards, this does not lose any generality; just subtract the maximum from everything.) Then

$$p(\tau|\mathcal{O}_{1:T}) = \frac{p(\tau, \mathcal{O}_{1:T})}{p(\mathcal{O}_{1:T})} \propto p(\tau) \exp\left(\sum_t r(s_t, a_t)\right).$$

The model is shown in Figure 20. The modeling choice is a bit odd, but we will see later that it produces an elegant mathematical formulation.

Our method for inference looks something like the following:

1. Compute backward messages $\beta_t(s_t, a_t) = p(\mathcal{O}_{t:T}|s_t, a_t)$
2. Compute policy $p(a_t|s_t, \mathcal{O}_{1:T})$
3. Compute forward messages $\alpha_t(s_t) = p(s_t|O_{1:t-1})$

Compute backward messages The setup is quite similar to a Hidden Markov Model or Kalman Filter. Let's do some math.

$$\begin{aligned}
\beta_t(s_t, a_t) &= p(\mathcal{O}_{t:T}|s_t, a_t) \\
&= \int p(\mathcal{O}_{t:T}, s_{t+1}|s_t, a_t) \mathrm{d}s_{t+1} \\
&= \int p(\mathcal{O}_{t+1:T}|s_{t+1}) p(s_{t+1}|s_t, a_t) p(\mathcal{O}_t|s_t, a_t) \mathrm{d}s_{t+1} \\
&= p(\mathcal{O}_t|s_t, a_t) \mathbb{E}_{s_{t+1} \sim p(\cdot|s_t, a_t)}[p(\beta_{t+1}(s_{t+1}))]
\end{aligned}$$

where at each step we employ the Markov property. Next, observe that

$$\begin{aligned} p(\mathcal{O}_{t+1:T}|s_{t+1}) &= \int p(\mathcal{O}_{t+1:T}|s_{t+1}, a_{t+1}) p(a_{t+1}|s_{t+1}) \mathrm{d}a_{t+1} \\ &= \mathbb{E}_{a_t \sim p(\cdot|s_t)}[\beta_t(s_t, a_t)], \end{aligned}$$

where in the last line we have assumed that the action prior $p(a_{t+1}|s_{t+1})$ is uniform. Then:

$$V_t(s_t) := \log \beta_t(s_t) = \log \int \exp(Q_t(s_t, a_t)) \mathrm{d}a_t.$$

Then as $Q_t(s_t, a_t)$ gets larger, $V_t(s_t) \to \max_{a_t} Q_t(s_t, a_t)$ (a sort of softmax).

$$Q_t(s_t, a_t) := \log \beta_t(s_t, a_t) = r(s_t, a_t) + \log \mathbb{E}_{s_{t+1} \sim p(\cdot|s_t, a_t)}[\exp(V_{t+1}(s_{t+1}))].$$

So the update is similar to a Bellman backup; in the deterministic case, we can get rid of the expectation to obtain the original backup $Q_t(s_t, a_t) = r(s_t, a_t) + V_{t+1}(s_{t+1})$. However, in the stochastic case, it is too optimistic because we saw previously that the backup is a softmax.[2]

In the non-uniform case,

$$V(s_t) = \log \int \exp(Q(s_t, a_t) + \log p(a_t|s_t)),$$

so we can just set the Bellman backup as

$$\tilde{Q}(s_t, a_t) = r(s_t, a_t) + \log p(a_t|s_t) + \log \mathbb{E}[\exp(V(s_{t+1}))],$$

and the same setup works. Equivalently, it suffices to define $\tilde{r}(s_t, a_t) = r(s_t, a_t) + \log p(a_t|s_t)$, i.e. we can ignore the action prior.

Compute policy By definition,

$$\pi(a_t|s_t) = p(a_t|s_t, \mathcal{O}_{1:T}) = p(a_t|s_t, \mathcal{O}_{t:T}) = \frac{p(\mathcal{O}_{t:T}|s_t, a_t)}{p(\mathcal{O}_{t:T}|s_t)} p(a_t|s_t) \propto \frac{\beta_t(s_t, a_t)}{\beta_t(s_t)},$$

where in the last equality we have assumed a uniform action prior.

In particular, $\pi(a_t|s_t) = \exp(\log \beta_t(s_t, a_t) - \log \beta_t(s_t)) = \exp(Q_t(s_t, a_t) - V_t(s_t, a_t)) = \exp(A_t(s_t, a_t))$. This is analogous to Boltzmann exploration.

Forward messages $\alpha_1(s_1) = p(s_1)$ is known. For $t \geq 2$,

$$\begin{aligned} \alpha_t(s_t) &= p(s_t \mid \mathcal{O}_{1:t-1}) \\ &= \int p(s_t, s_{t-1}, a_{t-1} \mid \mathcal{O}_{1:t-1}) \mathrm{d}s_{t-1} \mathrm{d}a_{t-1} \\ &= \int p(s_t \mid s_{t-1}, a_{t-1}, \mathcal{O}_{1:t-1}) p(a_{t-1} \mid s_{t-1}, \mathcal{O}_{1:t-1}) p(s_{t-1} \mid \mathcal{O}_{1:t-1}) \mathrm{d}s_{t-1} \mathrm{d}a_{t-1} \\ &= \int p(s_t \mid s_{t-1}, a_{t-1}) p(a_{t-1} \mid s_{t-1}, \mathcal{O}_{t-1}) p(s_{t-1} \mid \mathcal{O}_{1:t-1}) \mathrm{d}s_{t-1} \mathrm{d}a_{t-1} \\ &= \int p(s_t \mid s_{t-1}, a_{t-1}) \frac{p(\mathcal{O}_{t-1} \mid s_{t-1}, a_{t-1}) p(a_{t-1} \mid s_{t-1})}{\cancel{p(\mathcal{O}_{t-1} \mid s_{t-1})}} \frac{\cancel{p(\mathcal{O}_{t-1} \mid s_{t-1})} p(s_{t-1} \mid \mathcal{O}_{1:t-2})}{p(\mathcal{O}_{t-1} \mid \mathcal{O}_{1:t-2})} \mathrm{d}s_{t-1} \mathrm{d}a_{t-1}. \end{aligned}$$

[2] For example, suppose you are buying a lottery ticket. Your expected value isn't good, but the log expected value of the exponential is good. More generally, such an update would be over-reliant on lucky scenarios.

Each term is tractable here because we know the dynamics, and $p(s_{t-1} \mid O_{1:t-2}) = \alpha_{t-1}(s_{t-1})$.

This setup enables us to compute

$$p(s_t \mid \mathcal{O}_{1:T}) = \frac{p(\mathcal{O}_{t:T} \mid s_t)p(s_t, \mathcal{O}_{1:t-1})}{p(\mathcal{O}_{1:T})} \propto p(\mathcal{O}_{t:T} \mid s_t)p(s_t \mid \mathcal{O}_{1:t-1}) = \beta_t(s_t)\alpha_t(s_t).$$

Intuitively, we can regard β as the proportion of states with a high probability of reaching the goal and α as the proportion of states with a high probability of being reached from the initial state. Thus, the state marginals are the product of the two.

Control as Variational Inference

In complex state spaces or the dynamics are not known, we need to do approximate inference.

First, let us rationalize the issue with the optimism problem. The inference problem from earlier (when marginalizing and conditioning) is $p(a_t|s_t, \mathcal{O}_{1:T})$ (the policy). Refer again to the lottery example. This goes something like: "if you won a million dollars, what is the chance you won the lottery?" Of course, this isn't what we actually want, since the posterior and prior are distinct (the evidence that you won a million dollars drastically changes your lottery chances). So the question that we want to ask is more like: "given that you obtained high reward and that your transition probabilities remain unchanged, what was your action probability?"

Formally, we want a distribution $q(s_{1:T}, a_{1:T})$ that is close to $p(s_{1:T}, a_{1:T}|\mathcal{O}_{1:T})$ but has dynamics $p(s_{t+1}|s_t, a_t)$. To do so, we proceed via variational inference: we only learn an action distribution $q(a_t|s_t)$ and take

$$q(s_{1:T}, a_{1:T}) = p(s_1)\prod_t p(s_{t+1}|s_t, a_t)q(a_t|s_t).$$

Letting $\mathbf{x} = \mathcal{O}_{1:T}$ and $\mathbf{z} = (s_{1:T}, a_{1:T})$, substituting into the variational lower bound

$$\log p(\mathbf{x}) \geq \mathbb{E}_{\mathbf{z}\sim q}[\log p(\mathbf{x}, \mathbf{z}) - \log q(\mathbf{z})]$$

yields

$$\begin{aligned}
\log p(\mathcal{O}_{1:T}) &\geq \mathbb{E}_{s_{1:T}, a_{1:T}\sim q}\left[\left(\log p(s_1) + \sum_{t=1}^{T}\log p(s_{t+1}|s_t, a_t) + \sum_{t=1}^{T}\log p(\mathcal{O}_t|s_t, a_t)\right)\right. \\
&\qquad \left. - \left(\log p(s_1) + \sum_{t=1}^{T}\log p(s_{t+1}|s_t, a_t) + \sum_{t=1}^{T}\log q(s_t|a_t)\right)\right] \\
&= \mathbb{E}_{s_{1:T}, a_{1:T}\sim q}\left[\sum_t r(s_t, a_t) - \log q(s_t|a_t)\right] \\
&= \sum_t \mathbb{E}_{s_{1:T}, a_{1:T}\sim q}[r(s_t, a_t) + \mathcal{H}(q(a_t|s_t))].
\end{aligned}$$

We can optimize this lower bound using the same strategies as in exact inference. For the base case,

$$q(s_T, a_T) = \arg\max_q \mathbb{E}_{s_T\sim q}\left[\mathbb{E}_{a_T\sim q(\cdot|s_T)}[r(s_T, a_T) - \log q(a_T|s_T)]\right]$$

is optimized when we take $q(a_T|s_T) \propto \exp(r(s_T, a_T))$, or equivalently

$$q(a_T|s_T) = \exp(Q(s_T, a_T) - V(s_T)).$$

In the same way, where this time *we use the regular Bellman backup* $Q(s_t, a_t) = r(s_t, a_t) + \mathbb{E}_{s_{t+1} \sim p(\cdot|s_t, a_t)} V(s_{t+1})$, we find the same optimizer at t.

RL Algorithms

Q-Learning is exactly the same as before, except that the value function iteration is performed as $V(s') = \log \int \exp(Q_\varphi(s', a'))\mathrm{d}a'$.

Policy gradient with soft optimality imposes an entropy regularizer, i.e. the objective is

$$\sum_t \mathbb{E}_{s_t, a_t}[r(s_t, a_t)] + \mathbb{E}_{s_t}[\mathcal{H}(\pi(a_t|s_t))].$$

This works better than the base policy gradient, which often results in entropy collapse in early stages of training.

Haarnoja et al. [11] shows that soft Q-learning works well for pre-training. The authors use the example of pre-training with the reward function being the speed of the ant. Soft Q-learning has an entropy regularizer and will therefore send ants in different directions. Thus, if you have a new task asking ants to go in a particular direction, it just needs to forget all the other directions. In contrast, another agent trained to run in a different direction would need to unlearn the first direction and relearn the second.

Soft actor-critic applies the same entropy regularizer to Q-function update,

$$Q(s, a) \leftarrow r(s, a) + \mathbb{E}_{s' \sim p, a' \sim \pi(\cdot|s)}[Q(s', a') - \log \pi(a'|s')].$$

20 Inverse RL (11/01)

The videos begin here.

In the past, we have explored problems where the reward function is provided to us. However, in many cases, a reward function is difficult to specify, but perhaps we have lots of data of experts performing the task successfully. We are interested in learning the reward function from experts and using RL on this learned reward function.

In robotic imitation learning, we aim to copy the actions performed by the expert rather without reasoning about the outcomes of those actions. Humans act differently, because we act according the intent of the expert rather than imitating them. (Suppose that a human dropped an object on the ground. Then the natural action would be to pick up that object, not to drop something on the ground yourself.)

However, this problem is very underspecified; given the data, there are infinitely many possible reward functions explaining the data (you want the data to satisfy some constraint, but figuring out how strict the constraint should be is pretty difficult).

In both forward and backward RL, you are given states $\mathcal{S}$ and actions $\mathcal{A}$, and sometimes transition probabilities $p(s'|s, a)$. In forward RL, you are given a reward function $r(s, a)$ and try to learn $\pi^*(a|s)$. In inverse RL, you are given trajectories $\{\tau_i\}$ sampled from $\pi^*(\tau)$, and you try to learn $r_\psi(s, a)$ and $\pi^*(a|s)$.

Learning the reward function

In the previous lecture, we saw that the reward function is given by the optimality variables. That is, $p(\mathcal{O}_t|s_t,a_t;\psi) = \exp\big(r_\psi(s_t,a_t)\big)$, so by Bayes' rule $\log p(\tau|\mathcal{O}_t;\psi) = \log p(\tau) + \sum_t r_\psi(s_t,a_t)$. We will learn ψ by maximum likelihood,

$$\arg\max_\psi \frac{1}{N}\sum_{i=1}^N \log p(\tau_i|\mathcal{O}_{1:T};\psi) = \arg\max_\psi \frac{1}{N}\sum_{i=1}^N r_\psi(\tau_i).$$

The issue is that you could choose ψ fairly arbitrarily given that the rewards happen to be high. But the rewards might be even higher elsewhere, so in practice we apply a regularizer known as the **partition function**:

$$\arg\max_\psi \frac{1}{N}\sum_{i=1}^N r_\psi(\tau_i) - \log Z \qquad\qquad Z := \int p(\tau)\exp\big(r_\psi(\tau)\big)\mathrm{d}\tau.$$

Then

$$\nabla_\psi \mathcal{L} = \frac{1}{N}\sum_{i=1}^N \nabla_\psi r_\psi(\tau_i) - \frac{1}{Z}\int p(\tau)\exp\big(r_\psi(\tau)\big)\nabla_\psi r_\psi(\tau)\mathrm{d}\tau.$$

Observe that $\frac{1}{Z}\int p(\tau)\exp\big(r_\psi(\tau)\big)$ amounts to a probability distribution over $\tau \sim p(\cdot|\mathcal{O}_{1:T};\psi)$, i.e.

$$\nabla_\psi \mathcal{L} = \mathbb{E}_{\tau\sim\pi^*}[\nabla_\psi r_\psi(\tau_i)] - \mathbb{E}_{\tau\sim p(\cdot|\mathcal{O}_{1:T};\psi)}[\nabla_\psi r_\psi(\tau_i)].$$

The first term is obtained by sampling the expert, and the second by running the soft-optimal policy under the current reward we discussed last time:

$$\begin{aligned}\mathbb{E}_{\tau\sim p(\cdot|\mathcal{O}_{1:T};\psi)}[\nabla_\psi r_\psi(\tau_i)] &= \sum_{t=1}^T \mathbb{E}_{(s_t,a_t)\sim p(\cdot|\mathcal{O}_{1:T};\psi)}[\nabla_\psi r_\psi(s_t,a_t)] \\ &= \sum_{t=1}^T \iint p(a_t|s_t,\mathcal{O}_{1:T};\psi)p(s_t|\mathcal{O}_{1:T};\psi)\nabla_\psi r_\psi(s_t,a_t)\mathrm{d}a_t\mathrm{d}s_t.\end{aligned}$$

The inside expression can be defined as

$$\mu_t(s_t,a_t) \propto \frac{\beta(s_t,a_t)}{\beta(s_t)}\cdot\alpha(s_t)\beta(s_t) = \beta(s_t,a_t)\alpha(s_t).$$

Substituting,

$$\mathbb{E}_{\tau\sim p(\cdot|\mathcal{O}_{1:T};\psi)}[\nabla_\psi r_\psi(\tau_i)] = \sum_{t=1}^T \boldsymbol{\mu}_t^\top \nabla_\psi \mathbf{r}_\psi,$$

where $\boldsymbol{\mu}$ and $\mathbf{r}$ are length $|\mathcal{S}||\mathcal{A}|$ vectors storing the values for each (s_t,a_t).

Note. Since we must compute the forward and backward messages, this method works for exact inference, i.e. we must know the dynamics. Furthermore, it only works when $|\mathcal{S}|$ and $|\mathcal{A}|$ are small.

In fact, this is known as the maximum entropy algorithm Ziebart et al. [43]. In the case where $r_\psi(s_t,a_t) = \psi^\top f(s_t,a_t)$, we can show that the optimization problem is equivalent to

$$\max_\psi \mathcal{H}(\pi^{r_\psi}) \quad \text{s.t.} \quad \mathbb{E}_{\pi^{r_\psi}}[f] = \mathbb{E}_{\pi^*}[f].$$

Approximations in High Dimensions

In the previous section, we found that the gradient of the objective is the difference of two terms: the first by sampling $\{\tau_i\}$ from the expert, and the second by running the soft-optimal policy. The first term is easy to do. For the second term, we can learn $p(a_t|s_t, \mathcal{O}_{1:T}; \psi)$ using any max-entropy RL algorithm, then run the resulting policy to generate samples $\{\tau_j\}$. That is,

$$\nabla_\psi \mathcal{L} \approx \frac{1}{N}\sum_{i=1}^{N} \nabla_\psi r_\psi(\tau_i) - \frac{1}{M}\sum_{j=1}^{M} \nabla_\psi r_\psi(\tau_j).$$

This is viable but computationally intractable, since we would need to run the policy to convergence for each gradient step on r_ψ. An idea to improve the runtime is to just improve $p(a_t|s_t, \mathcal{O}_{1:T}; \psi)$ a little bit rather than learn it to convergence (i.e. lazy policy optimization). This means that our estimate is biased due to sampling from the wrong policy, but we can simply use importance sampling:

$$\nabla_\psi \mathcal{L} \approx \frac{1}{N}\sum_{i=1}^{N} \nabla_\psi r_\psi(\tau_i) - \frac{1}{\sum_j w_j}\sum_{j=1}^{M} w_j \nabla_\psi r_\psi(\tau_j),$$

where

$$w_j = \frac{p(\tau)\exp\big(r_\psi(\tau_j)\big)}{\pi(\tau_j)} = \frac{p(s_1)\prod_t p(s_{t+1}|s_t,a_t)\exp\big(r_\psi(s_t,a_t)\big)}{p(s_1)\prod_t p(s_{t+1}|s_t,a_t)\pi(a_t|s_t)} = \frac{\exp\left(\sum_t r_\psi(s_t,a_t)\right)}{\prod_t \pi(a_t|s_t)}.$$

This algorithm is introduced in Finn et al. [6].

GANs

One might recognize that this setup is a bit like a game: in one step, the expert demonstrations are made more likely and the current policy samples less likely. Then the policy is updated, making it harder to distinguish them from the demos (one can see this because at convergence, the importance sampling weights are all ones).

Recall the GAN setup: the generator accepts some random noise z and produce some distribution $p_\theta(x|z)$, which should resemble the true data distribution. Then some samples are collected from the generator outputs $p_\theta(x)$ and samples from the true data distribution, $p^*(x)$. Then the discriminator $D_\psi(x)$ emits a probability of x coming from p^* versus p_θ (a binary classification problem). Then we alternately optimize the discriminator and generator,

$$\psi = \arg\max_\psi \frac{1}{N}\sum_{x_i \overset{\text{iid}}{\sim} p^*} \log D_\psi(x) + \frac{1}{M}\sum_{x_j \overset{\text{iid}}{\sim} p_\theta} \log\big(1 - D_\psi(x)\big)$$

$$\theta = \arg\max_\theta \mathbb{E}_{x\sim p_\theta} \log D_\psi(x).$$

In CS 182, we showed that

$$D^*(x) = \frac{p^*(x)}{p_\theta(x) + p^*(x)}.$$

With this setup, we can frame IRL as a GAN, where we use samples $\{\tau_i\} \sim \pi_\theta$ as the generator. If we use expert demonstrations according to $p^*(\tau)$, we can let

$$D_\psi(\tau) = \frac{p(\tau)\frac{1}{Z}\exp(r(\tau))}{p_\theta(\tau) + p(\tau)\frac{1}{Z}\exp(r(\tau))} = \frac{\frac{1}{Z}\exp(r(\tau))}{\prod_t \pi_\theta(a_t|s_t) + \frac{1}{Z}\exp(r(\tau))},$$

and optimize

$$\psi \leftarrow \arg\max_{\psi} \mathbb{E}_{\tau\sim p^*}[\log D_\psi(\tau)] + \mathbb{E}_{\tau\sim\pi_\theta}[\log(1 - D_\psi(\tau))].$$

Through some math, one can find that this in fact optimizes Z, and we don't need the importance weights anymore.

Remark. One can use a typical binary classification neural net, which may be a bit easier to set up. However, the discriminator "knows nothing" at convergence, and the reward is lost.

20.1 RLHF Algorithms and Applications (11/06)

This is a guest lecture by Eric Mitchell (Stanford), with video here.

RLHF

A language model (LM) is an autoregressive model over tokens, i.e. LM π maps an input sequence $X = (x_1, \ldots, x_n)$ to a distribution over the vocabulary V, $x_{n+1} \sim \pi(\cdot|X)$. Accordingly, states are token sequences, and actions are tokens (or sequences of tokens).

GPT-4 is the product of four steps:

Step 0: Unsupervised pre-training This is GPT-3. Just feed in a ton of scraped text ($>$ 1T tokens) with this autoregressive task. We will call the resulting policy π_{θ_0}.

Step 1: Supervised fine-tuning Provide some human demonstrations: pass in prompt and some good human responses. We will call the resulting polciy $\pi_{\theta_{\text{SFT}}}$.

Step 2: Fit a reward function The motivation for steps 2 and 3 (why bother with RL?) is that as per usual, we seek to exceed human performance, not merely imitate it.

We want a reward function that assigns high reward to things humans like and low rewards to things humans don't like. Scoring things on a 1-10 scale is pretty hard (maybe you have a recipe; two people will give different responses for their opinion on whether it's well-written). It is easy, however, to put two responses in front of people and ask which one is preferable. This process enables us to collect a dataset $\mathcal{D} = \{x^i, y_w^i, y_\ell^i\}$, where x's are prompts, y_w's are "winning" responses, and y_ℓ's are "losing" responses.

The **Bradley-Terry Model** connects scores to preferences, asserting that $\mathbb{P}(a \succ b) = \sigma(s(a) - s(b))$ for some unobserved s. Thus, we simply learn $r_\varphi := s$ such that it aligns with our preferences, running gradient ascent on

$$\mathcal{L}(\varphi) = \mathbb{E}_{(x,y_w,y_\ell)\sim\mathcal{D}} \log \sigma(r_\varphi(x, y_w) - r_\varphi(x, y_\ell)).$$

Remark. The whole setup is a bit silly, since the quality of a response is more intricate than a binary decision. It's somewhat surprising that it works so well.

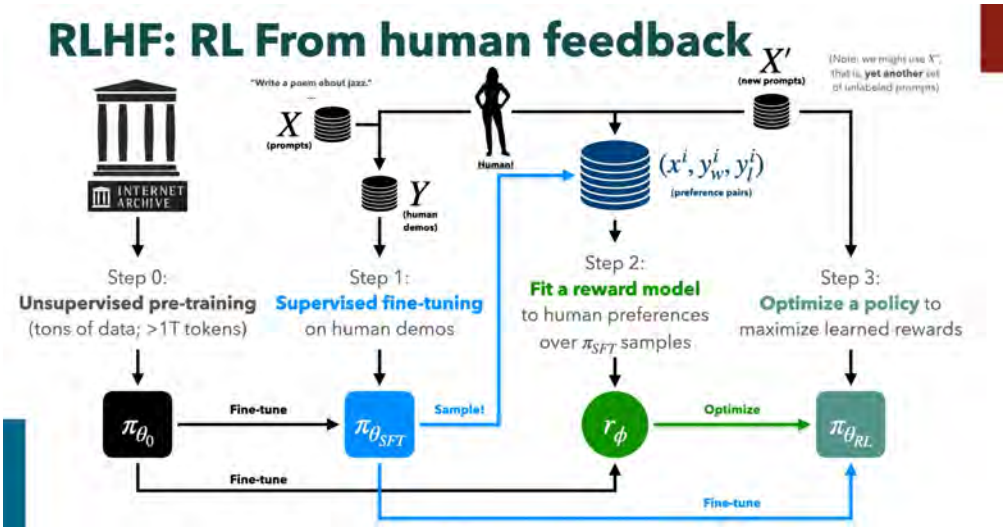


Figure 21: RLHF

Step 3: Learn a policy We will optimize

$$\max_{\theta} \mathbb{E}_{x\sim\mathcal{D},y\sim\pi_\theta(\cdot|x)} r_\varphi(x,y) - \beta D_{\mathrm{KL}}(\pi_\theta(y|x)\|\pi_{\mathrm{SFT}}(y|x)). \tag{3}$$

The reason for the regularizer is that our reward model might be bad, and RL will push this to progressively higher rewards when in reality the results are garbage because they stray too far from the SFT policy. Typically, the policy is optimized with PPO, and we call the final policy $\pi_{\theta_{\mathrm{RL}}}$.

And this recipe really does work! Humans prefer a 1.3B RLHF model to a 175B SFT model.

There are many variations of RLHF, e.g. implementing "latent cells" given by a judge of human similarity improves diversity in CLIP, and Anthropic's LLM Claude is based on RL with AI feedback.

Direct Preference Optimization

RLHF suffers from a few problems:

- Hard to implement
- Resource requirements (lots of different models!)
- Since the loss is based on the differences between rewards, there is an extra degree of freedom. In particular, you can't even compare rewards across different inputs, which prevents writing value functions or other desirable things.

DPO simplifies this process by removing the RL entirely: if we parameterize the reward function correctly, then we can extract the optimal policy with no required learning.

Specifically, the solution to (3) is

$$\pi^*(y|x) = \frac{1}{Z(x)}\pi_{\mathrm{SFT}}(y|x)\exp\left(\frac{1}{\beta}r(x,y)\right),$$

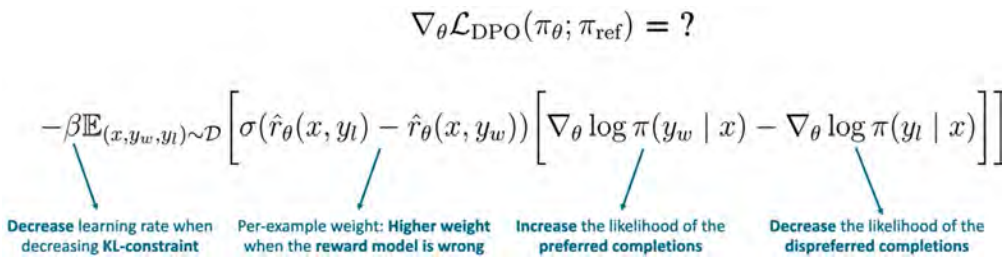


Figure 22: Gradient of $\mathcal{L}_{\text{DPO}}$

where $Z(x)$ is a normalizer so that the expression sums to 1. Note that it is intractable because the sum is over all possible sequences of tokens y. But since we are only interested in differences in r, this turns out not to matter:

$$r(x,y) = \beta \log \frac{\pi^*(y|x)}{\pi_{\text{SFT}}(y|x)} + \beta \log Z(x)$$

implies that we can run gradient ascent on

$$\mathcal{L}_{\text{DPO}}(\pi_\theta, \pi_{\text{SFT}}) = \mathbb{E}_{(x,y_w,y_\ell)\sim\mathcal{D}}\left[\log \sigma\left(\beta \frac{\pi_\theta(y_w|x)}{\pi_{\text{SFT}}(y_w|x)} - \beta \frac{\pi_\theta(y_\ell|x)}{\pi_{\text{SFT}}(y_\ell|x)}\right)\right].$$

In doing so, we've gotten rid of the extra degree of freedom (the precise value of $Z(x)$ doesn't matter) while maintaining expressiveness.

Question 20.1.1. How do we make LLMs more factual?

Answer. See Tian et al. [38]. The idea is that you can sample "atomic claims," e.g. "Yo-Yo Ma was born in 1951." So you can either scrape Wikipedia for this information or sample responses from GPT-3.5 to extract the most likely response.

21 RL with Sequence Models and Language Models (11/13)

The videos begin here.

Previously, we saw that POMDPs act quite different from typical MDPs. Observations do not satisfy the Markov property because they encode state information, and as such past observations contribute information about the current state.

Moreover, they may not be as well-behaved in general:

- POMDPs can benefit from "information-gathering," whereas in the fully observed setting you don't need this because you have access to the full state.
- It is possible to only have stochastic optimal policies, whereas in the fully observed setting there always exists some optimal deterministic policy.

We consider what methods handle partial observability, where "handle" means that it can find the optimal memoryless policy, i.e. $\pi_\theta(a|o)$ where o is the current observation. Effectively, we are asking whether the method invokes the Markov property, in which case we must take more care.

Policy gradients The estimator

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

does not work because the advantage is a function of the state, not the observation. (The point of a value function is that we expect to get the same value from a state regardless of what came before.)

The estimator

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

works. So does

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

since it only relies on the future not influencing the past, not the Markov property.

Value-based methods As mentioned previously, it is not ok to make a Q-function with observations rather than states.

Model-based methods Directly swapping states for observations doesn't work. Consider an example where there are two doors. Exactly one of them is unlocked, each w.p. 0.5. Then $\mathbb{P}(o' = \text{pass} \mid o = \text{left}, a = \text{open}) = 0.5$ according to the model due to memorylessness, when in reality $\mathbb{P}(o' = \text{pass} \mid o = \text{left}, a = \text{open}) = 0$ if you've tried the door already and it hasn't opened. Markovian models are unable to capture these dynamics.

Approach 0. On the other hand, it is possible to learn a Markovian state space. Do variational inference with $\mathbf{x} = (o_1, \ldots, o_T)$ and $\mathbf{z} = (z_1, \ldots, z_T)$. Specifically, we enforce a prior $z_1 \sim \mathcal{N}(0, I)$, so that $p_\psi(\mathbf{z}) = p(z_1)\prod_t p_\psi(z_{t+1}|z_t, a_t)$. Then we learn $p_\theta(\mathbf{o}|\mathbf{z}) := \prod_t p_\theta(o_t|z_t)$ and $q_\varphi(\mathbf{z}|\mathbf{o}) := \prod_t q_\varphi(z_t|o_t)$.

This method is overkill; it is not necessary to predict all of the observations, as in general, this is probably harder to do than the actual RL problem.

Approach 1. A much simpler approach is to just use the observation history $s_t := (o_1, \ldots, o_T)$ as states. Clearly this satisfies the Markov property, because conditioning on s_t tells you everything about s_{t-1}.

To make this work with arbitrary-length histories, you need to use a sequence model, e.g. a RNN/LSTM/Transformer.

Computationally, it is expensive to sample trajectories

$$(s_{1,i}, a_{1,i}, \ldots, s_{T,i}, a_{T,i}) = (o_{1:1,i}, a_{1,i}, \ldots, o_{1:T,i}, a_{T,i})$$

because the length of the vector scales with T^2. The key idea is that you can use the hidden states in the replay buffer. As of Sergey Levine's knowledge, no one has figured out the analog for transformers.

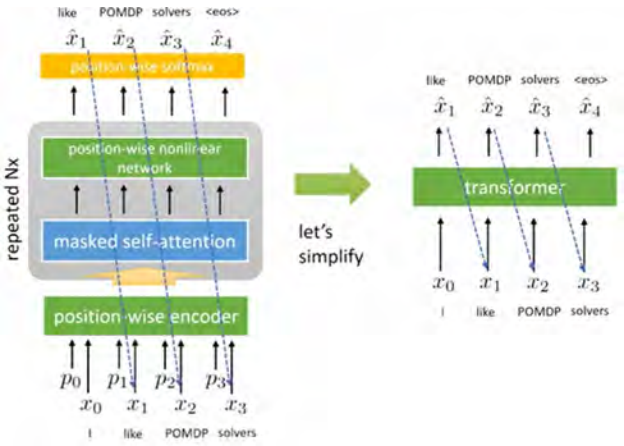


Figure 23: Transformer-based LM

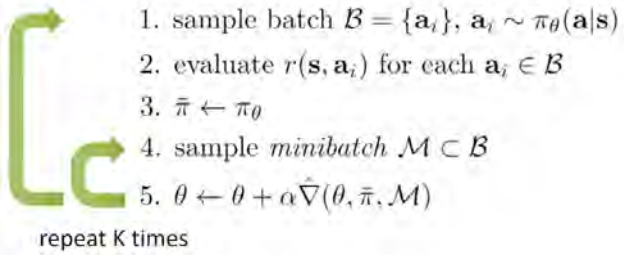


Figure 24: LM-RL setup

RLHF

A typical LM looks something like Figure 23. As in Section 20.1, we can regard the policy $\pi_\theta(a|s)$ as the process of autoregressive generation. For example, if $x_{1:4}$ is the prompt and we proceed to generate x_5 and x_6, this is equivalent to

$$\pi_\theta(a|s) = p(x_5|x_{1:4})p(x_6|x_{1:4}, x_5).$$

(So one timestep in the RL world can correspond to one or more timesteps in the LM world.)

With a policy gradient approach,

$$\begin{aligned}
\nabla_\theta \mathbb{E}_{\pi_\theta(a|s)}[r(s,a)] &= \mathbb{E}_{\pi_\theta(a|s)}[\nabla_\theta \log \pi_\theta(a|s) r(s,a)] \\
&\approx \frac{1}{N}\sum_{i=1}^{N} \nabla_\theta \log \pi_\theta(a_i|s) r(s,a_i) && \text{(sample from } \pi_\theta\text{)} \\
&\approx \frac{1}{N}\sum_{i=1}^{N} \frac{\pi_\theta(a_i|s)}{\bar{\pi}(a_i|s)} \nabla_\theta \log \pi_\theta(a_i|s) r(s,a_i) && \text{(sample from } \bar{\pi}\text{)}
\end{aligned}$$

The latter approach is generally preferred because collecting samples from an LM is expensive, so using some $\bar{\pi}$ saves us from running the model on every gradient step. A typical simple setup is shown in Figure 24.

As mentioned in the previous section, the idea is to learn a reward function with a transformer model. Since human preferences are expensive, most of the data comes from the initial super-

vised model, and each RL update only adds a small subset of preferences. We also apply a KL-divergence penalty to avoid over-optimization.

Multi-step RL with LMs

I got bored of this and skipped this video.

22 Transfer Learning and Meta-Learning (11/15)

The videos begin here.

Transfer Learning

Transfer learning is using experience from one set of tasks for faster learning and better performance on a new task.

"Shot" means the number of attempts in the target domain, e.g. zero-shot means that you are running a policy only trained in the source domain. One-shot, few-shot, and many-shot are also common terms.

There are lots of different ideas; training something on video games to play another game and training robots on videos to grasp objects will have very different strategies. There are, however, several broadly applicable techniques used across domains, which we will discuss here.

Pre-training and fine-tuning is the most popular method in supervised learning. The most common issues we might face when applying it to RL are as follows:

Domain shift Representations learned in the source domain may not be useful in the target domain.

In computer vision, one strategy is as follows: suppose you train some algorithm that learns the correct action given an image. (This can be either supervised or RL.) Since it's hard to train an RL algorithm in the real world, you train it using simulation data, where your images are a little different from the real world: maybe cars look a little different, or the world has rain when the simulation does not.

The assumption you can make (in this case, understandably because your two domains are quite similar), is the invariance assumption: everything that is different between the two domains is irrelevant. That is, if x denotes your input and y the output, then $p_{\text{sim}}(x) \neq p_{\text{real}}(x)$, but there exists some representation $z = f(x)$ that captures everything you need to know about x: $p(y|z) = p(y|x)$ and $p_{\text{sim}}(z) = p_{\text{real}}(z)$.

We implement this using a GAN-like approach: you take some middle layers in each network (say, after a convolution), which we call z. Then you learn a discriminator $D_\varphi(z)$, which is a binary classifier for which domain you think it came from.

Difference in the MDP Some things are possible to do in the source domain but not target.

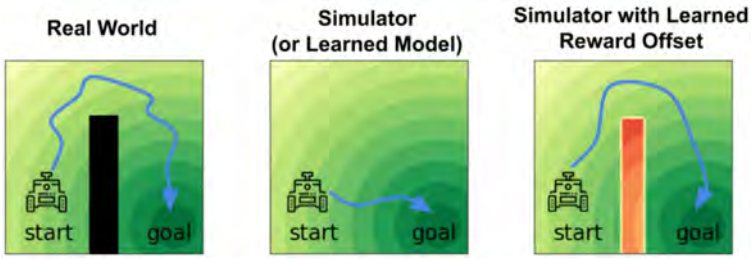


Figure 25: RL dynamics transfer

For example, suppose your setup is in Figure 25, i.e. your goal is to get from start to goal, but in the real world there is a wall between them. One strategy, provided that you have done some exploration in both environments, is to learn the dynamics and apply a reward bonus

$$\Delta r(s_t, a_t, s_{t+1}) = \log p_{\text{target}}(s_{t+1}|s_t, a_t) - \log p_{\text{source}}(s_{t+1}|s_t, a_t).$$

A better strategy is to learn two discriminators, where the equality is actually a $\propto$ by Bayes' rule:

$$\begin{aligned}\Delta r(s_t, a_t, s_{t+1}) = &\log p(\text{target} \mid s_t, a_t, s_{t+1}) - \log p(\text{target} \mid s_t, a_t) \\ &- \log p(\text{source} \mid s_t, a_t, s_{t+1}) + \log p(\text{source} \mid s_t, a_t).\end{aligned}$$

Fine-tuning Now we consider the other direction: some things are possible to do in the target domain but not the source domain. (In the previous subsection, we intuitivelly saw a way of transferring the intersection of the two domains, rather than to the second domain.)

Thus, we need to figure out how to fine-tune. Some problems:

- In contrast to CV/NLP, downstream RL tasks are generally much less diverse. Useful features are usually task-specific, and policies and value functions may be overly specialized.
- Optimal policies in fully-observed MDPs are deterministic. Thus, at convergence, there is a loss of exploration, i.e. we want our original policy to be more diverse than what we would normally get through typical RL pre-training.

One idea is that you can train with randomized physical parameters; the more varied the training domain is, the more likely we are to generalize in zero-shot to a slightly different domain. Randomization is widely used in simulation to real-world transfer.

Multi-task learning

The idea is that if you have several similar tasks that might require similar representations (maybe all of them are trying to grasp objects, but different object shapes/purposes), it may be beneficial to learn all of them simultaneously rather than independently. Moreover, if you are presented with another task at test time, it seems more likely that the representations you have learned will generalize given multiple rather than individual tasks.

A straightforward but important idea is that multi-task RL is just single-task RL in a joint MDP. In particular, if you are using the same robot for all environments, then you can learn a joint policy π, which is trained exactly the same way, but you first sample an MDP from the first state $p(s_0)$.

Equivalently, you can view this as a contextual policy $\pi(a|s,\omega)$, where you simply define an augmented state space $\tilde{s} = [s,\omega]$, $\tilde{S} = S \times \Omega$.

A particularly common class of contextual policies are goal-conditioned, i.e. $\pi(a|s,g)$, where g is another state. We define $r(s,a,g) = \mathbb{1}\{d(s,g) < \varepsilon\}$. This setup seems exceptionally simple, and theoretically it transfers well in zero-shot to a new task if it's another goal. However, in practice it is quite hard to train (for example, maybe only a few of the states are actually goals). Some literature covers clever ways of selecting goals Kaelbling [15], representing value functions Schaul et al. [30], or formulating rewards Andrychowicz et al. [1] and loss functions Eysenbach et al. [4].

Supervised meta-learning

Suppose you've learned 100 tasks already. Can you figure out how to learn other tasks more efficiently?

Remark. This question is particularly relevant to deep RL: model-free learning requires a ton of samples, so if we can meta-learn a faster reinforcement learner, we can learn new tasks efficiently.

As an introduction, we consider meta-learning in the supervised case; see Figure 26. Suppose we are considering an image classification problem. You have several train/test data pairs, where each pair consists of images from the same set of classes (for example, in the first train/test data pair in the meta-training set, the dog and piano in the test set are also in the training data). Those classes may differ across different pairs. Your meta-training set refers to some subset of these datasets, and the meta-testing set is the rest. Through meta-training, the idea is that you can pass in some new training data you've never seen before and produce good outputs on the test set.

Formally, in supervised learning, we accept some image x and learn an output (e.g. label) y. In supervised meta-learning, we accept a training dataset $\mathcal{D}^{\text{train}}$ and test image x and learn a label y. To read in the training set, you can use an RNN or transformer.

Framing this as an optimization problem, in "generic learning," we minimize a loss on the training set,

$$\theta^* = \arg\min_\theta \mathcal{L}(\mathcal{D}^{\text{train}};\theta).$$

In "generic meta-learning," we learn f_θ that accepts a training set and produces some parameters φ that are everything you need to know about the training set. That is,

$$\varphi_i = f_\theta(\mathcal{D}_i^{\text{train}}) \qquad \theta^* = \arg\min_\theta \sum_{i=1}^{n} \mathcal{L}(\mathcal{D}_i^{\text{test}};\varphi_i).$$

In the example where f_θ is an RNN, it will produce some hidden state h_i, which is passed in as the parameters to another network parameterized by θ_p producing an output y from input x. Meta-learning involves learning both the RNN and the classifier, i.e. $\varphi_i = [h_i, \theta_p]$.

Meta–Reinforcement Learning

We use the same setup as in the previous section but rename the variables. In RL,

$$\theta^* = \arg\max_\theta \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)].$$

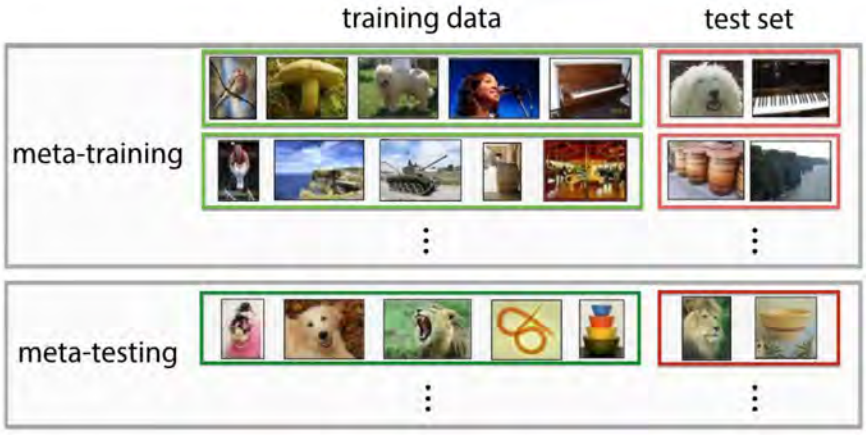
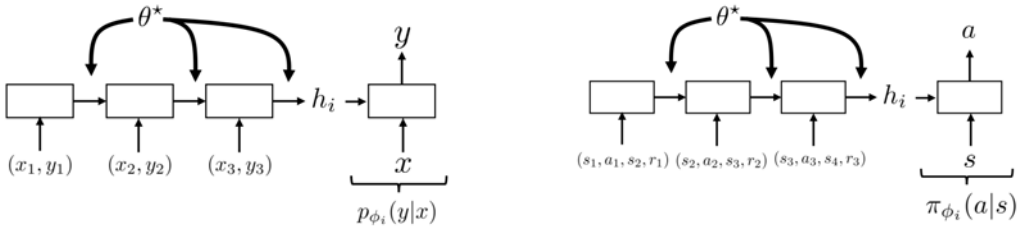


Figure 26: Meta-learning with supervised learning



(a) Supervised case (b) RL case

Figure 27: Meta-learning with an RNN

Equivalently, this is a function of the MDP $\mathcal{M} = \{\mathcal{S}, \mathcal{A}, \mathcal{P}, r\}$.

In meta-RL, we learn a representation for the MDP,

$$\varphi_i = f_\theta(\mathcal{M}_i) \qquad \theta^* = \arg\max_\theta \sum_{i=1}^{n} \mathbb{E}_{\tau \sim \pi_{\varphi_i}}[R(\tau)]. \tag{4}$$

We assume that the meta-training MDPs $\mathcal{M}_i$ comes from a distribution p, and at meta–test time, we sample $\mathcal{M}_{\text{test}}$ from the same distribution.

Another view of this setup is that running a learning procedure f_θ on $\mathcal{M}_i$ to gather the relevant context φ_i is all the information you need to roughly equivalently deploy a policy that is conditioned on all of the experience in the test MDP, $z_t = \{(s_i, a_i, r_i)\}_{i=1}^{t-1}$. That is,

$$\pi_\theta(a_t|s_t, \varphi_i) \approx \pi_\theta(a_t|s_t, z_t).$$

This motivates us to use the same setup as before: we collect transitions $\{(s_t, a_t, s_{t+1}, r_t)\}_{t=1}^{T}$ (which may or may not come from the same episode). Then we feed these through the RNN to produce a hidden state h_i, which, along with s and policy parameters θ_π, is used to produce an action a. As before, $\varphi_i = [h_i, \theta_\pi]$. See Figure 27 to confirm for yourself that these methods are really quite similar.

Put simply, this is equivalent to training an RNN policy. The reason why this works at all is that the hidden state is not reset between episodes, so the network should be able to figure out more things about the MDP from the experience it's seen so far. The reason it learns to explore is that

through successful meta-training, it will learn that it cannot achieve rewards by trying the same things repeatedly, and so optimizing the total reward over the entire meta-episode with an RL policy automatically learns to explore.

High level analysis:

- Conceptually simple.
- Easy to apply.
- Vulnerable to meta-overfitting: if the task at test-time is a bit OOD from the training tasks, the h might not be very good.
- RNNs are hard to optimize, but transformers have worked well.

Gradient-based meta RL

We can take a step back and think about what the algorithm is actually doing. $f_\theta(\mathcal{M}_i)$ produces some representation φ_i, which parameterizes the policy. We are learning θ, so equivalently we are trying to improve the policy with our experience from $\mathcal{M}_i$.

In standard RL, our objective is

$$\theta^* = \arg\max_\theta \underbrace{\mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)]}_{J(\theta)},$$

where we run gradient steps $\theta^{(k+1)} = \theta^{(k)} + \alpha\nabla_{\theta^{(k)}} J(\theta^{(k)})$. Following (4), we can define

$$f_\theta(\mathcal{M}_i) = \theta + \alpha\nabla_\theta J_i(\theta)$$

as the result of one gradient step.[3] Now, you can take a gradient step on φ_i,

$$\theta \leftarrow \theta + \beta\sum_{i=1}^{n} \nabla_\theta J_i(\theta + \alpha\nabla_\theta J_i(\theta)).$$

We call this **model-agnostic meta-learning (MAML)** [7]. Of course, the same method works in supervised learning.

High-level analysis:

- It is good at extrapolating, i.e. you can run more gradient steps at test-time to do better.
- Conceptually elegant.
- It is complex and requires many samples in RL.
- It does not generalize well beyond policy gradient.

Meta-RL as a POMDP

Recall that a POMDP is $\mathcal{M} = \{\mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{P}, \mathcal{E}, r\}$, where $\mathcal{E}$ are emission probabilities $p(o_t|s_t)$.

Moreover, recall that the meta-RL objective $\pi_\theta(a|s,z)$ involves inferring z (all the information you need to solve the current task) from context $\{(s_t, a_t, s_{t+1}, r_t)\}$. If we regard this as a combined

[3] Of course you can also do more than one, but the math gets ugly. Higher-order derivatives are difficult in RL. In practice, usually people just do one step, even outside of RL.

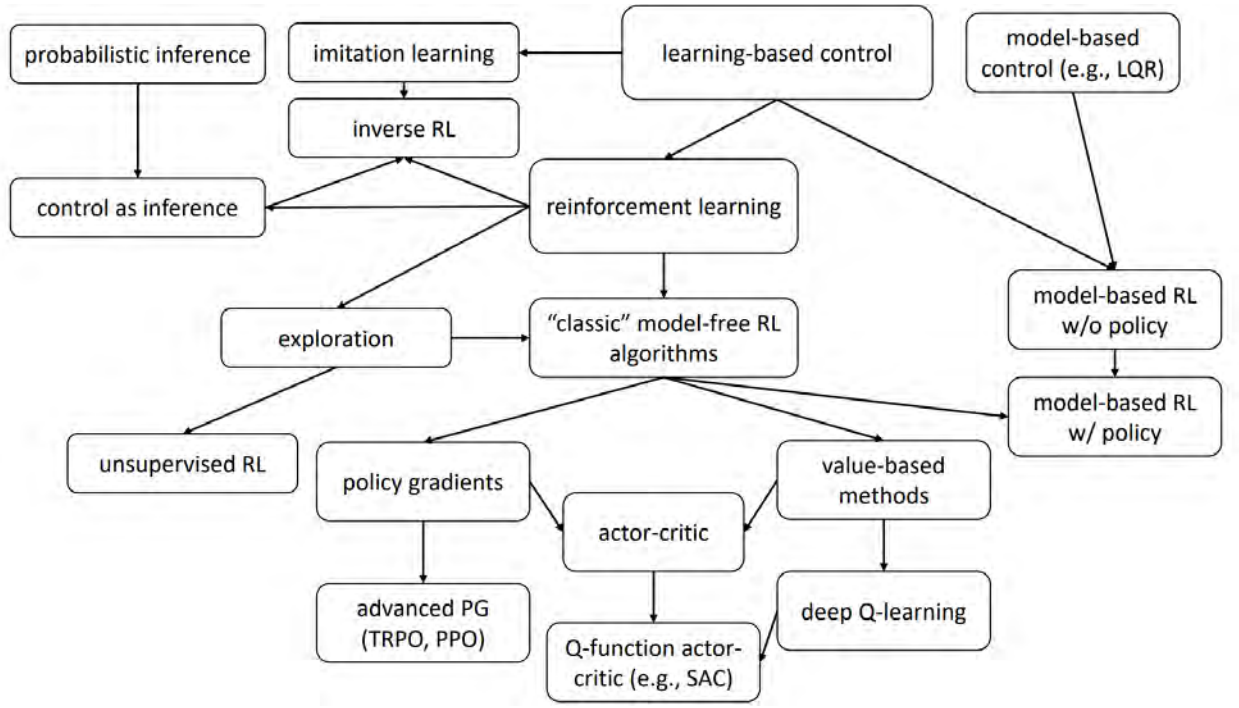


Figure 28: An overview of topics we've covered

state space where $\tilde{\mathcal{S}} = \mathcal{S} \times \mathcal{Z}$, $\tilde{\mathcal{O}} = \mathcal{S}$, $\tilde{s} = [s, z]$, and $\tilde{o} = z$, we have constructed a POMDP $\mathcal{M} = \{\tilde{\mathcal{S}}, \mathcal{A}, \tilde{\mathcal{O}}, \tilde{\mathcal{P}}, \mathcal{E}, r\}$.

To do so, we are missing $\tilde{\mathcal{P}}$, i.e. $p(z_t|s_{1:t}, a_{1:t}, r_{1:t})$. But we can just get this via variational inference: define a policy $\pi_\theta(a_t|s_t, z_t)$ and an inference network $q_\varphi(z_t|s_1, a_1, r_1, \ldots, s_t, a_t, r_t)$. Then we will optimize

$$(\theta, \varphi) = \arg\max_{\theta,\varphi} \mathbb{E}_{z\sim q_\varphi, \tau\sim\pi_\theta}[R_i(\tau) - D_{\mathrm{KL}}(q(z|\cdots)\|p(z))].$$

Finally, we sample z from the posterior, and act according to $\pi_\theta(a|s, z)$.

High-level analysis:

- Simple, effective exploration with posterior sampling.
- Elegant reduction to POMDP.
- Vulnerable to meta-overfitting.
- Challenging to optimize in practice.

23 Challenges and Open Problems (11/20)

The videos begin here.

Challenges with core algorithms

Stability: does your algorithm converge?

- Reinforcement learning solves a much harder objective than supervised learning. Thus, devising stable RL algorithms is very hard.
- **Challenge:** **Fitted Q/fitted value methods** with deep network function estimators are not contractions, so there are no guarantees of convergence. This makes lots of hyperparameters required for stability: a target network, replay buffer size, clipping, learning rates. It requires regularization: large networks, normalization, data augmentation.
- **Open problem:** Why do large models work so well in supervised learning? There is some magic that makes it work, since naively we would expect catastrophic overfitting.
- **Open problem:** Does this same regularizing effect hold in value-based methods?
- **Challenge:** We learned that the **gradient estimator** has very high variance, and in pathological examples you need exponentially more samples to counter this effect. Thus, we are careful with batch size, learning rate, and the design of the baseline.
- **Challenge:** Naively, **model-based RL** seems straightforward because the fitting process is supervised learning. However, due to the iterative nature of RL requiring backprop through time, this optimization process is nontrivial; the model class and the training method end up being very sensitive, and moreover having a better model does not guarantee a better policy. Intuitively, while the model is trained to minimize error, not all errors in the policy are created equal. Even worse, your policy might exploit your model, taking an action that the model predicts is erroneously good. Naturally, a model-based method is adversarial.

Efficiency: how long does it take to converge? From least to most efficient (most to least samples):

- Gradient-free methods (NES, CMA)
- Fully online, on-policy methods (A3C)
- Batchwise policy gradient methods (TRPO)
- Replay buffer value estimation methods (Q-learning, DDPG, NAF, SAC)
- Model-based deep RL (PETS, guided policy search)
- Model-based "shallow" RL (PILCO)

In fact, each "level" corresponds to a $10\times$ efficiency increase in terms of samples. (The first is on the order of 15 days; the last is on the order of a few seconds.)

Sample efficiency is important because it makes real-world learning practical. In simulation, however, they are not the whole story; training a model takes a lot of time, so if samples are fast or easily parallelizable, you might end up using TRPO over SAC for example.

Generalization: after it converges, does it generalize? In supervised learning and generative methods, we focus primarily on large-scale data and models, emphasizing diversity and evaluated on generalization. In RL, we evaluate our models on their performance on small-scale tasks, emphasizing mastery.

- **Challenge:** In supervised learning, typically things are run once. If the model is not performing up to par, you just run it for more epochs. By contrast, in deep RL, we re-run the entire training loop multiple times until we are satisfied. This works well on Gym tasks, but if you are working with Internet-scale data, we'll need improvements to RL methods that provide more suitable workflows for deep RL research.
- **Challenge:** In the past we've trained a humanoid robot to run in a straight line on a flat infinite plane. However, really we want it to be able to accomplish several other tasks, e.g. climbing or interacting with other objects. An idea is to use off-policy RL, where we build a large dataset from past interaction and occasionally we get more data if we are not satisfied with the results.
- **Open problem:** Last lecture, we discussed framing multi-task learning as a single task, where you sample an MDP in your first timestep. It is compatible with existing methods. But how can we devise solutions with lower variance or sample complexity?

Challenges with assumptions

- Is this even the right problem formulation? (e.g. maybe you don't have a ground truth reward function)
- What is the source of supervision? (How does the reward function look? What does a sparse reward function mean, intuitively?)
  - If you want to learn from many different tasks, you need to get those tasks somewhere. Specifying what you want is quite hard; for example, if you ask a robot to pour a glass of water, simply understanding whether the glass is full requires a complex perception system.
  - With the advent of LMs, it is becoming more natural to learn whether you are making a user happy. There are lots of avenues for exploration:
    - **Open problem:** Learning objectives and rewards from demonstrations (inverse RL)
    - **Open problem:** Generating objectives automatically (so you can generalize to different tasks)
    - **Open problem:** Leverage language and human preferences
- Rethinking the problem formulation. The basic RL problem is not set in stone; the assumptions vary based on the problem setting. For example, consider defining a control problem. Some considerations are as follows:
  - What is the data? (It may be very hard to simulate a real-world process.)
  - What is the goal (reward, demonstration, preferences)?
  - Is the supervision the same as the goal, or if not, how can we provide hints without biasing the agent?

Q: What is RL?

Here, we will discuss an overall view of how RL can be viewed from different perspectives.

A: An engineering tool

Suppose you are flying a rocket and want to find an optimal trajectory for it. Traditionally, you'd write down lots of differential equations describing its physics, solve for its position over time, linearize, and apply a feedback controller. Equivalently, you "invert the physics," computing the controls required to achieve a desired outcome.

A simulator gives you the forward process: you plug in some controls and see how the system evolves over time. (Of course, implementing this is hard, but the equations are the same in either case.) RL in that simulator lets you invert the physical system, serving as an optimization tool.

As RL methods get more sophisticated and we have a system we understand well (e.g. airplane flight, driving a car), RL will become the go-to controller.

A: Learning in the real world

A strong motivator is Moravec's paradox:

We are all prodigious olympians in perceptual and motor areas, so good that we make the difficult look easy. Abstract thought, though, is a new trick, perhaps less than 100 thousand years old. We have not yet mastered it. It is not all that intrinsically difficult; it just seems so when we do it.

Hans Moravec

In the historic games where DeepBlue and AlphaGo beat the world champions in chess and Go, respectively, there was a human on the other side of the board, simply present for the ability to move their bodies.

Collectively, the conclusion is quite suggestive: evolutionarily, we are all very good at perceiving the world and moving, and it's not really that hard to beat the world champion at chess or Go. We're just not very good at it.

The main lesson of thirty-five years of AI research is that the hard problems are easy and the easy problems are hard. The mental abilities of a fouryear-old that we take for granted – recognizing a face, lifting a pencil, walking across a room, answering a question – in fact solve some of the hardest engineering problems ever conceived.

Steven Pinker

So the key problem is in the physical universe: we need to find good ways to learn the visual physical properties of the world, perhaps in the same way (or some alternative) that humans do.

fix this tran tion

Open problem: How do we engineer a system that can deal with the unexpected?

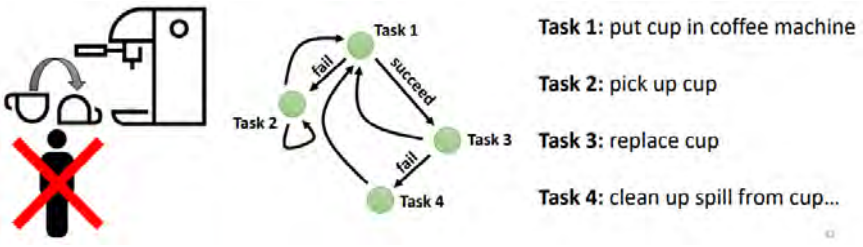


Figure 29: Multi-task learning with a coffee example

Consider the problem of putting an RL agent on a deserted island. A human will discover solutions autonomously, e.g. finding water, building a shelter. Current AI systems are very bad at this, given that there is minimal external supervision about what to do. RL is our only hope for this, but there is rarely research in these "hard" universes.

This raises some more specific questions:

- **Open problem:** How do we tell RL agents what we want them to do?
  - Christiano et al. [2] introduces deep RL from preferences: for example, teaching a robot to do a flip by asking a human which one of two demos it prefers, even though the reward is quite hard to specify.
- **Open problem:** How can we learn fully autonomously? (In Atari games we get lots of episodes, but you can only die once in real life.)
  - Suppose you are training an agent to fill up a cup with coffee (in the real world). If it fails, and the coffee spills, you'd need a human to pick up the cup and help the robot reset. To remove the human, you can consider it a multi-task problem. See Figure 29.
- **Open problem:** How do we remain robust as the environment changes around us?
- **Open problem:** What is the right way to generalize using experience and prior data?
- **Open problem:** What is the right way to bootstrap exploration with prior experience?
  - When you train on a new task, you can leverage previous experience that is initialized with a behavioral prior — that is, a collection of tasks that were useful in the past, not intended for any particular task. When you put it in a new environment, it might not be doing your desired task, but at least it's doing *something* useful.

A: Universal learning

The formula for LMs is a giant unlabeled dataset (i.e. the Internet) and a small labeled dataset telling the model what to do.

Remark. Prompting is a bit of an art. The distribution an LM learns is quite poor, yet we are asking it for high-quality results.

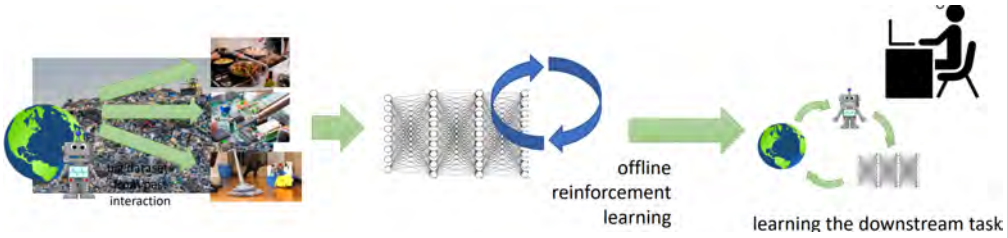


Figure 30: Pipeline for RL on everything

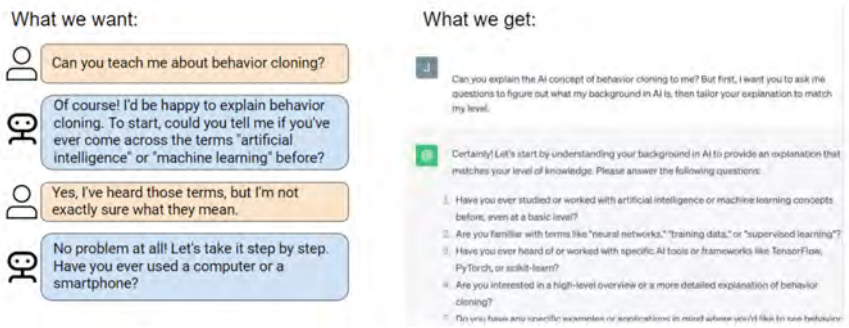


Figure 31: Increasing LM interactivity for teaching

Concept 23.1. We need machine learning only to *produce adaptable and complex decisions*.

The idea, then, is to use RL in place of supervised methods. The data we collect from the world tells us what *could* be done, rather than what *should* be. A limited amount of supervision would specify which tasks an agent should learn to perform. The essence is that we are not copying the world, as an LM does, but rather use it to understand the possibilities available to us and select the ones that are most optimal for accomplishing particular tasks.

The issue is that, naively, this is costly because RL requires interaction with the world. However, using methods we learned in the course, this is possible to establish:

- Take a large, diverse, possibly low-quality dataset
- Run an offline or model-based RL–like process on human-defined skills or goal-conditioned RL or self-supervised skill discovery
- Run self-supervised learning of decision-making: learn how to make decisions that lead to particular outcomes
- With limited supervision, learn outcomes that humans actually want

See Figure 30. ChatGPT is a version of this pipeline, but Sergey suggests using RL for the entire process. That is, our RL should be unsupervised, then adapted to downstream tasks.

Hong et al. [13] provides an example of how this paradigm might be used, where the authors train an agent LM that can act more interactively, as in Figure 31.

A model-based RL approach would look like the following: have a LM simulate plausible human responses, then optimize for the kinds of responses that are useful to us. That is, we treat the entire process as a POMDP. (In this case, instead of inverting rocket science, we are inverting an LM.)

Takeaway 23.2. Most of our learning should come from the unsupervised or self-supervised sort, and we need a model-based structure.

References

[1] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay, 2018.

[2] Paul Christiano, Jan Leike, Tom B. Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences, 2023.

[3] Kurtland Chua, Roberto Calandra, Rowan McAllister, and Sergey Levine. Deep reinforcement learning in a handful of trials using probabilistic dynamics models. (arXiv:1805.12114), November 2018. doi: 10.48550/arXiv.1805.12114. URL http://arxiv.org/abs/1805.12114. arXiv:1805.12114 [cs, stat].

[4] Benjamin Eysenbach, Ruslan Salakhutdinov, and Sergey Levine. C-learning: Learning to achieve goals via recursive classification, 2021.

[5] Vladimir Feinberg, Alvin Wan, Ion Stoica, Michael I. Jordan, Joseph E. Gonzalez, and Sergey Levine. Model-based value estimation for efficient model-free reinforcement learning. (arXiv:1803.00101), February 2018. URL http://arxiv.org/abs/1803.00101. arXiv:1803.00101 [cs, stat].

[6] Chelsea Finn, Sergey Levine, and Pieter Abbeel. Guided cost learning: Deep inverse optimal control via policy optimization. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 49–58, New York, New York, USA, 20–22 Jun 2016. PMLR. URL https://proceedings.mlr.press/v48/finn16.html.

[7] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks, 2017.

[8] Justin Fu, John D. Co-Reyes, and Sergey Levine. Ex2: Exploration with exemplar models for deep reinforcement learning. (arXiv:1703.01260), May 2017. doi: 10.48550/arXiv.1703.01260. URL http://arxiv.org/abs/1703.01260. arXiv:1703.01260 [cs].

[9] Dibya Ghosh, Abhishek Gupta, Ashwin Reddy, Justin Fu, Coline Devin, Benjamin Eysenbach, and Sergey Levine. Learning to reach goals via iterated supervised learning, 2020.

[10] Shixiang Gu, Timothy Lillicrap, Ilya Sutskever, and Sergey Levine. Continuous deep q-learning with model-based acceleration. (arXiv:1603.00748), March 2016. doi: 10.48550/arXiv.1603.00748. URL http://arxiv.org/abs/1603.00748. arXiv:1603.00748 [cs].

[11] Tuomas Haarnoja, Haoran Tang, Pieter Abbeel, and Sergey Levine. Reinforcement learning with deep energy-based policies, 2017.

[12] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. (arXiv:1801.01290), 8 2018. doi: 10.48550/arXiv.1801.01290. URL http://arxiv.org/abs/1801.01290. arXiv:1801.01290 [cs, stat].

[13] Joey Hong, Sergey Levine, and Anca Dragan. Zero-shot goal-directed dialogue via rl on imagined conversations, 2023.

[14] Michael Janner, Qiyang Li, and Sergey Levine. Offline reinforcement learning as one big sequence modeling problem, June 2021. URL https://arxiv.org/abs/2106.02039v4.

[15] Leslie Pack Kaelbling. Learning to achieve goals. In *International Joint Conference on Artificial Intelligence*, 1993. URL https://api.semanticscholar.org/CorpusID:5538688.

[16] Sham M Kakade. A natural policy gradient. In *Advances in Neural Information Processing Systems*, vol-

ume 14. MIT Press, 2001. URL https://proceedings.neurips.cc/paper_files/paper/2001/hash/4b86abe48d358ecf194c56c69108433e-Abstract.html.

[17] Ilya Kostrikov, Ashvin Nair, and Sergey Levine. Offline reinforcement learning with implicit q-learning, 2021.

[18] Aviral Kumar, Aurick Zhou, George Tucker, and Sergey Levine. Conservative q-learning for offline reinforcement learning. (arXiv:2006.04779), August 2020. doi: 10.48550/arXiv.2006.04779. URL http://arxiv.org/abs/2006.04779. arXiv:2006.04779 [cs, stat].

[19] Aviral Kumar, Joey Hong, Anikait Singh, and Sergey Levine. When should we prefer offline reinforcement learning over behavioral cloning? (arXiv:2204.05618), April 2022. doi: 10.48550/arXiv.2204.05618. URL http://arxiv.org/abs/2204.05618. arXiv:2204.05618 [cs].

[20] Sergey Levine. Reinforcement learning and control as probabilistic inference: Tutorial and review, 2018.

[21] Sergey Levine and Vladlen Koltun. Guided policy search. In Sanjoy Dasgupta and David McAllester, editors, *Proceedings of the 30th International Conference on Machine Learning*, volume 28 of *Proceedings of Machine Learning Research*, pages 1–9, Atlanta, Georgia, USA, 6 2013. PMLR. URL https://proceedings.mlr.press/v28/levine13.html.

[22] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2019.

[23] Corey Lynch, Mohi Khansari, Ted Xiao, Vikash Kumar, Jonathan Tompson, Sergey Levine, and Pierre Sermanet. Learning latent plans from play, 2019.

[24] D. Q. Mayne, J. B. Rawlings, C. V. Rao, and P. O. M. Scokaert. Constrained model predictive control: Stability and optimality. *Automatica*, 36(6):789–814, June 2000. ISSN 0005-1098. doi: 10.1016/S0005-1098(99)00214-9.

[25] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[26] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. (arXiv:1602.01783), 6 2016. doi: 10.48550/arXiv.1602.01783. URL http://arxiv.org/abs/1602.01783. arXiv:1602.01783 [cs].

[27] Xue Bin Peng, Aviral Kumar, Grace Zhang, and Sergey Levine. Advantage-weighted regression: Simple and scalable off-policy reinforcement learning. (arXiv:1910.00177), October 2019. doi: 10.48550/arXiv.1910.00177. URL http://arxiv.org/abs/1910.00177. arXiv:1910.00177 [cs, stat].

[28] Jan Peters and Stefan Schaal. Reinforcement learning of motor skills with policy gradients. *Neural Networks*, 21(4):682–697, 2008. ISSN 0893-6080. doi: https://doi.org/10.1016/j.neunet.2008.02.003. URL https://www.sciencedirect.com/science/article/pii/S0893608008000701. Robotics and Neuroscience.

[29] Daniel Russo and Benjamin Van Roy. Learning to optimize via information-directed sampling. (arXiv:1403.5556), July 2017. doi: 10.48550/arXiv.1403.5556. URL http://arxiv.org/abs/1403.5556. arXiv:1403.5556 [cs].

[30] Tom Schaul, Daniel Horgan, Karol Gregor, and David Silver. Universal value function approximators. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 1312–1320, Lille, France, 07–09 Jul 2015. PMLR. URL https://proceedings.mlr.press/v37/schaul15.html.

[31] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. (arXiv:1502.05477), 4 2017. doi: 10.48550/arXiv.1502.05477. URL http://arxiv.org/abs/1502.05477. arXiv:1502.05477 [cs].

[32] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation, 2018.

[33] Dhruv Shah, Ajay Sridhar, Arjun Bhorkar, Noriaki Hirose, and Sergey Levine. Gnm: A general navigation model to drive any robot, 2023.

[34] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1): 9–44, 8 1988. ISSN 1573-0565. doi: 10.1007/BF00115009.

[35] Richard S. Sutton. Dyna, an integrated architecture for learning, planning, and reacting. *ACM SIGART Bulletin*, 2(4):160–163, 7 1991. ISSN 0163-5719. doi: 10.1145/122344.122377.

[36] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 1999. URL https://proceedings.neurips.cc/paper/1999/hash/464d828b85b0bed98e80ade0a5c43b0f-Abstract.html.

[37] Haoran Tang, Rein Houthooft, Davis Foote, Adam Stooke, Xi Chen, Yan Duan, John Schulman, Filip De Turck, and Pieter Abbeel. #exploration: A study of count-based exploration for deep reinforcement learning. (arXiv:1611.04717), December 2017. doi: 10.48550/arXiv.1611.04717. URL http://arxiv.org/abs/1611.04717. arXiv:1611.04717 [cs].

[38] Katherine Tian, Eric Mitchell, Huaxiu Yao, Christopher D. Manning, and Chelsea Finn. Fine-tuning language models for factuality, 2023.

[39] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. 1989.

[40] Tianhe Yu, Garrett Thomas, Lantao Yu, Stefano Ermon, James Zou, Sergey Levine, Chelsea Finn, and Tengyu Ma. Mopo: Model-based offline policy optimization, May 2020. URL https://arxiv.org/abs/2005.13239v6.

[41] Tianhe Yu, Aviral Kumar, Rafael Rafailov, Aravind Rajeswaran, Sergey Levine, and Chelsea Finn. Combo: Conservative offline model-based policy optimization, February 2021. URL https://arxiv.org/abs/2102.08363v2.

[42] Ruiyi Zhang, Bo Dai, Lihong Li, and Dale Schuurmans. Gendice: Generalized offline estimation of stationary values. (arXiv:2002.09072), February 2020. doi: 10.48550/arXiv.2002.09072. URL http://arxiv.org/abs/2002.09072. arXiv:2002.09072 [cs, stat].

[43] Brian D Ziebart, Andrew Maas, J Andrew Bagnell, and Anind K Dey. Maximum entropy inverse reinforcement learning.