CS 194-129 Spring 2018 — Deep Neural Networks — Canny — Lecture 20

1 Lecture

Markov Decision Processes

MDPs are the abstractions for the RL environments we'll be looking at.

- Policy $\pi_\theta(a_t|o_t)$: the program that the actor executes
  - a probabilistic model (a probability distribution over possible actions at time t, conditioned on the observation at time t)
  - for partially observed systems (observation $\neq$ state)
- Policy $\pi_\theta(a_t|s_t)$
  - a fully observed policy (which we'll assume today)
  - make action decisions based on knowledge of the actual state

We no longer have a set of demonstration trajectories; instead, we have a reward function. $r(s, a)$ is the reward function and tells us which states and actions are better. Often the reward signal occurs sparsely.

A **Markov chain** is defined by $\mathcal{S}$, a state space, and $\mathcal{T}$, a transition operator.

A **Markov decision process** is defined by $\mathcal{S}$, a state space, $\mathcal{A}$, an action space, $\mathcal{T}$, a transition operator (now a tensor which depends on previous state, next state, and previous action), and $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ the reward function.

- Trajectory $\tau = (s_0, a_0, ..., s_T, a_T)$: a sequence of states and actions
  - execution or unrolling of a policy

Reinforcement Learning

RL is like supervised learning, except instead of a label we produce an action, and instead of being able to compare a label with a ground truth label, we have to go through the environment function which returns a new state and reward. (This is assuming we're dealing with model-free algorithms, and have no model of the environment we can use for prediction.)

We don't know whether an action is a good choice or a bad choice without *acting it out in the world.*

Because we have an MDP (which obeys the Markov property), the probability of an entire trajectory factors into a simple product:

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

A policy $\pi_\theta(\tau)$ $[= p_\theta(\tau)]$ can then be thought of as an assignment of probabilities to trajectories. Our goal is to determine the policy that maximizes expected reward:

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

The expectation is over trajectories chosen according to the distribution of our current policy $p_\theta(\tau)$.

Value Functions

We define the expected (discounted) reward from state s_i, under a given policy π, as

$$V(s_i) = \sum_{a_i} \underbrace{\pi(a_i|s_i)}_{\substack{\text{probability of} \\ \text{taking action } a_i}} \left(\underbrace{r(s_i, a_i)}_{\substack{\text{actual reward} \\ \text{at current step}}} + \overbrace{\gamma \sum_{s_{i+1}} V(s_{i+1}) p(s_{i+1}|s_i, a_i)}^{\substack{\text{expected total} \\ \text{future reward}}} \right)$$

This is otherwise known as the **reward-to-go**.

Bellman Update

Here, we only consider an optimal policy, i.e. one that always takes the best (max reward) action. In other words, we maximize the expected total reward directly in the value recurrence:

$$V(s_i) = \max_{a_i} \; r(s_i, a_i) + \gamma \sum_{s_{i+1}} V(s_{i+1}) p(s_{i+1}|s_i, a_i)$$

If the state space is small enough, we can solve the recurrence exactly using this iterative calculation.

In theory, this might seem like a dynamic programming problem, with a DAG of states and actions. However, the graph may have cycles and repeated updates may be necessary for each node... meaning it isn't really a dynamic programming problem. Also, initialization matters: node values should start at a quantity $\leq$ any possible reward.

Reinforcement Learning Challenges

In supervised learning, we can train end-to-end by minimizing a differentiable loss attached to the output of our network. This doesn't work in RL, even if we have a differentiable policy $\pi_\theta(a_i|s_i)$, because (a) the action isn't continuous and (b) we don't know the reward function so we can't differentiate through it.

- *The action isn't continuous.* The output of the network isn't continuous, as it is with a softmax or regression value. Even if we output the probability of an action (as opposed to an action choice), the environment isn't an open box, meaning we can't tell how the environment's going to respond to a given probability distribution of actions.
  - "The reward $r(s_i, a_i)$ is a function of the action a_i selected by the policy, and the action set is often discrete. We can't directly differentiate through a discrete set."
- *We don't know the reward function.* It's a black box (part of the environment).

Also, since rewards can occur sparsely, we might have hundreds of actions leading up to a reward, and it's hard to tell which were the most important. Thus, in assigning weights to actions we encounter the *temporal credit assignment problem*.

Policy Gradients

Policy gradients are different from the gradients we've used so far to optimize deep networks. Since we can't do a simple end-to-end reward maximization, we instead settle for an approximation in which we run the policy to generate sample trajectories and compute the *reward* and *probability* of each trajectory. Then we can compute the gradient of the reward with respect to the policy parameters θ.

Concretely, we define our maximization objective as $J(\theta)$:

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

i is the trajectory index; t is the time step.

Since $J(\theta)$ is an expectation over trajectories, we can't compute it directly. But we can approximate it by sampling trajectories from our policy. As we collect more and more samples, the distribution goes to $p_\theta(\tau)$.

As an integral, $J(\theta)$ appears as

$$J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \underbrace{\left[\sum_t r(s_t, a_t) \right]}_{r(\tau)} = \int \pi_\theta(\tau) r(\tau) \, d\tau$$

Then

$$\nabla_\theta J(\theta) = \int \nabla_\theta \pi_\theta(\tau) r(\tau) \, d\tau$$

$$\text{convenient identity: } \pi_\theta(\tau) \nabla_\theta \log \pi_\theta(\tau) = \pi_\theta(\tau) \frac{\nabla_\theta \pi_\theta(\tau)}{\pi_\theta(\tau)} = \nabla_\theta \pi_\theta(\tau)$$

$$= \int \pi_\theta(\tau) \nabla_\theta \log \pi_\theta(\tau) r(\tau) \, d\tau$$
$$= \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[\nabla_\theta \log \pi_\theta(\tau) r(\tau) \right]$$

Thus we have reduced the *gradient of our expected reward* to an *expected value of the gradient of the log of the policy*. Expected values are nice because we can enumerate trajectories, i.e. take an average of the value across trajectories for approximation.

And we can go one step further:

$$\pi_\theta(\tau) = p(s_1) \prod_{t=1}^{T} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t)$$

$$\log \pi_\theta(\tau) = \log p(s_1) + \sum_{t=1}^{T} \left[\log \pi_\theta(a_t|s_t) + \log p(s_{t+1}|s_t, a_t) \right]$$

meaning

$$\begin{aligned}
\nabla_\theta J(\theta) &= \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[\nabla_\theta \log \pi_\theta(\tau) r(\tau) \right] \\
&= \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[\nabla_\theta \left(\log p(s_1) + \sum_{t=1}^{T} \left[\log \pi_\theta(a_t|s_t) + \log p(s_{t+1}|s_t, a_t) \right] \right) r(\tau) \right] \\
&= \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[\nabla_\theta \left(\sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) \right) r(\tau) \right] \quad \begin{array}{l}\text{because, in computing a gradient with respect to } \theta, \\ \text{we don't care about terms independent of } \theta\end{array} \\
&= \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \right) \left(\sum_{t=1}^{T} r(s_t, a_t) \right) \right]
\end{aligned}$$

Now we have an effective way of calculating the gradient:

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

(To optimize, we iterate according to θ `+=` $\alpha \nabla_\theta J(\theta)$.) Altogether we have the `REINFORCE` algorithm:

1. Sample trajectories $\{\tau^i\}$ from $\pi_\theta(a_t|s_t)$ (run the current policy).
2. Compute $\nabla_\theta J(\theta)$ as $\sum_i \left(\sum_t \nabla_\theta \log \pi_\theta(a_t^i|s_t^i) \right) \left(\sum_t r(s_t^i, a_t^i) \right)$.
3. Iterate: θ `+=` $\alpha \nabla_\theta J(\theta)$.

Reducing Variance

A big problem with policy gradients: they have a lot of variance. The actual value of the reward matters; just by moving the reward up and down, we completely change what the policy gradient does. In other words, *which samples we get* can dramatically change what each update does. And there is a lot of variance in this, meaning that anytime we estimate our gradient we might get a crappy gradient.

There is a lot of variance in our *estimate of the gradient*. We would like to reduce this.

Reducing Variance: Causality

Actions can't affect past rewards. The policy at time t' cannot affect rewards prior to time t'. Hence

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})\right)\underbrace{\left(\sum_{t'=t}^{T} r(s_{i,t'}, a_{i,t'})\right)}_{\text{"reward to go"} = \hat{Q}_{i,t}}$$

we only need to consider downstream rewards.

Reducing Variance: Baselines

There can be a lot of variation in the reward. Thus it's highly valuable to subtract a baseline b from it; then everything is relative to the baseline. For example, the average reward baseline:

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

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

Off-Policy Learning

Policy gradient methods are normally trained on a policy's own trajectories as data. With off-policy learning, we can have an algorithm that learns from its own experience, but is also able to use off-policy data from a teacher (e.g. a human expert).

Restated: typically, a policy gradient method is *on-policy* (must generate new samples after every gradient step) because $\mathbb{E}_{\tau\sim\pi_\theta(\tau)}$ depends on the current policy in the $\nabla_\theta J(\theta)$ calculation. To make it off-policy, we can instead estimate the expectation using a different distribution, with samples from $\bar{\pi}(\tau)$ instead of $\pi_\theta(\tau)$.

Importance Sampling

To estimate an expected value over a distribution $p(x)$ given samples from another distribution $q(x)$, we can use **importance sampling**:

$$\mathbb{E}_{x\sim p(x)}[V(x)] = \mathbb{E}_{x\sim q(x)}[V(x)L(x)]$$

where $\mathbb{E}_{x\sim q(x)}[L(x)] = 1$.

$L(x)$ is a correction factor (the *importance weight*); a simple choice for it is $\frac{p(x)}{q(x)}$, seen by

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

To use off-policy policy gradients with importance sampling, we can make the correction in blue:

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

Here we sample from the policy $\pi_{\theta'}$ instead of π_θ.

TRPO + PPO

Policy gradients are pretty limited in complex, high-dimensional environments. TRPO and PPO represent the state of the art for optimization of RL models in continuous environments.

Currently, our model update is θ `+=` $\alpha \nabla_\theta J(\theta)$. Each gradient step is very expensive, so we would like to take only a few large (high α) steps. However, the gradients are very noisy, so we can't take large steps without risking instability. *TRPO and PPO were developed to take safe steps while still maximizing reward gain.*

What does it mean to be a "large" gradient step? We don't care how much θ changes, since the policy parameterization is arbitrary; rather, we care how much the action probabilities change. Thus we can maximize the reward *with a penalty for large changes in π_θ (the distribution over action probabilities).*

Namely, in TRPO (trust region policy optimization) we maximize

$$L(\theta') - c \, \mathrm{KL}(\pi_\theta, \pi_{\theta'})$$

where $L(\theta') = \mathbb{E}\left[\frac{\pi_{\theta'}(a|s)}{\pi_\theta(a|s)} A(s,a) \right]$ describes the *ratio of probabilities of taking action a (in the proposed policy state θ' versus the original policy state θ) multiplied by the advantage.* $A(s,a)$ is the advantage of action a and describes how much better a is than the average action (essentially, $A(s,a)$ serves as a reward value).

- We want to maximize the expected gain; note that when we move from policy θ to policy θ', we're multiplicatively transforming the likelihood of taking action a.
- At the same time, we want to make sure that the new parameters θ' aren't giving us a policy that's too different from the current one. So we also minimize the KL divergence ("difference in distributions").
- "Trust region": a region where a probability distribution doesn't change too much.

TRPO has two terms in its objective: one a relative gain in reward, the other the KL divergence between the current and proposed parameter states. PPO (proximal policy optimization) has an additional loss term which is considerably simplified and doesn't use KL divergence.