CS 194-129 Spring 2018 — Deep Neural Networks — Florensa — Lecture 21

1 Lecture

A value function $V(s)$ assigns a *reward-to-go* to a state s.

Reformulating the Policy Gradient

The gradient of the reward-to-go (w.r.t. θ) is

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

Concretely, $\hat{Q}_{i,t}$ is our estimate of the expected reward if we take action $a_{i,t}$ in state $s_{i,t}$. It can be seen as a single-sample estimate of the expected reward when we take an action at a state.

We will call $Q(s_t,a_t) = \sum_{t'=t}^{T}\mathbb{E}_{\pi_\theta}\left[\gamma^{t'-t}r(s_{t'},a_{t'})|s_t,a_t\right]$ the *true expected* reward-to-go.

alternatively: the total reward from taking action a_t in state s_t

Meanwhile, the value function $V^\pi(s_t) = \mathbb{E}_{a_t\sim\pi_\theta(a_t|s_t)}\left[Q^\pi(s_t,a_t)\right]$ represents the total reward from s_t.

And finally the advantage $A^\pi(s_t,a_t) = Q^\pi(s_t,a_t) - V^\pi(s_t)$ describes how much better a_t is than the average action according to π.

Our estimate of the gradient will now be $\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})A^\pi(s_{i,t},a_{i,t})$.

Value Function Fitting

The Q-function can alternatively be written as

$$\begin{aligned}Q^\pi(s_t,a_t) &= r(s_t,a_t) + \mathbb{E}_{s_{t+1}\sim p(s_{t+1}|s_t,a_t)}\left[V^\pi(s_{t+1})\right]\\ &\approx r(s_t,a_t) + V^\pi(s_{t+1}) \text{ with a single-sample estimate}\end{aligned}$$

Thus the advantage function can alternatively be written as

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

If we fit $V^\pi(s)$, we will by extension have the Q's and the A's.

Policy Evaluation

We can then perform policy evaluation using a sampling (Monte Carlo) approach:

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

The value function can be approximated by a neural network, with training data in the form of input/output pairs $\left\{\left(s_{i,t},\ \sum_{t'=t}^{T}\gamma^{t'-t}r(s_{i,t'},a_{i,t'})\right)\right\}$. (It can be fit using supervised regression.)

An improvement: since the ideal target value is $\sum_{t'=t}^{T} \gamma^{t'-t} \mathbb{E}_{\pi_\theta}[r(s_{t'}, a_{t'})|s_{i,t}] \approx r(s_{i,t}, a_{i,t}) + \gamma \hat{V}_\phi^\pi(s_{i,t+1})$, use $\left\{ \left(s_{i,t},\ r(s_{i,t}, a_{i,t}) + \gamma \hat{V}_\phi^\pi(s_{i,t+1}) \right) \right\}$ as the training data. This will reduce the variance.

($\hat{V}_\phi^\pi(s_{i,t+1})$ is the previous fitted value function.)

Actor-Critic Algorithms

The batch actor-critic algorithm:

1. Sample $\{s_i, a_i\}$ from $\pi_\theta(a|s)$ (run the policy).
2. Fit $\hat{V}_\phi^\pi(s)$ to sampled reward sums.
3. Evaluate $\hat{A}^\pi(s_i, a_i) = r(s_i, a_i) + \gamma \hat{V}_\phi^\pi(s_i') - \hat{V}_\phi^\pi(s_i)$.
4. $\nabla_\theta J(\theta) \approx \sum_i \nabla_\theta \log \pi_\theta(a_i|s_i) \hat{A}^\pi(s_i, a_i)$.
5. θ += $\alpha \nabla_\theta J(\theta)$.
6. Go back to step 1.

The online actor-critic algorithm is similar, except we take only a *single* action at each step and update $\hat{V}_\phi^\pi$ using the resulting information.

Review So Far

In policy evaluation, we fit a value function to the policy.

In actor-critic algorithms, the *actor* is the policy and the *critic* is the value function. Actor-critic algorithms reduce the variance of the policy gradient.

Omitting Policy Gradients

If we create a new policy that always takes the action $\arg\max_{a_t} A^\pi(s_t, a_t)$ (the best action from s_t, if we then follow π), this new policy will always be at least as good as the previous policy. So we have the option to forget about policy differentiation altogether and just use the policy implicitly defined by the advantage function:

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

π' is as good as π and probably better.

Suddenly, our algorithm becomes (1) generate samples, (2) fit A^π (or Q^π or V^π), and (3) set $\pi = \pi'$.

This is the flavor of value-based methods: they only fit Q-functions instead of also a parameterized policy.

Policy Iteration

1. Evaluate $A^\pi(s, a)$.
2. Set $\pi = \pi'$.

The argmax of the advantages is the same as the argmax of the Q's.

$$A^\pi(s, a) = \underbrace{r(s, a) + \gamma \mathbb{E}[V^\pi(s')]}_{Q^\pi(s,a)} - V^\pi(s)$$

$$\implies \arg\max_{a_t} A^\pi(s_t, a_t) = \arg\max_{a_t} Q^\pi(s_t, a_t)$$

Thus we can focus on the Q-function, actually.

Fitted Q-Iteration: Offline

1. Collect dataset $\{(s_i, a_i, s'_i, r_i)\}$ using some policy.
2. Repeat K times:
   (a) Set y_i (the target) $= r(s_i, a_i) + \gamma \max_{a'_i} Q_\phi(s'_i, a'_i)$.
   (b) Set ϕ (the parameters of the Q-function) $= \arg\min_\phi \frac{1}{2} \sum_i \|Q_\phi(s_i, a_i) - y_i\|^2$.

This algorithm is off-policy, because the targets don't depend on the fully trajectory and the transition [step $2a$] is independent of π (all we need is s_i and a_i). Hence we can reuse data collected under some other policy.

$\max_{a'_i} Q_\phi(s'_i, a'_i)$ *approximates the value of* π' *at* s'_i*, and is also the part that improves the policy.*

Fitted Q-Iteration: Online

We probably don't want to use *any* policy to collect data, though. Here, the online Q-iteration algorithm:

1. Take some action a_i and observe (s_i, a_i, s'_i, r_i).
2. $y_i = r(s_i, a_i) + \gamma \max_{a'} Q_\phi(s'_i, a'_i)$.
3. ϕ `-=` $\alpha \frac{dQ_\phi(s_i, a_i)}{d\phi}(Q_\phi(s_i, a_i) - y_i)$. (Then repeat.)

In choosing our actions, we can follow an epsilon-greedy approach. (With ϵ chance, take a random action. Otherwise, take the best action according to our current Q-function.)

A problem we'll encounter while using this algorithm: *correlated samples*. Sequential states are highly correlated (instead of being i.i.d. like we'd want). The solution: use a replay buffer.

Full Q-learning with replay buffer and target network:

1. Repeat:
   (a) Save target network parameters: $\phi' \leftarrow \phi$.
   (b) Repeat:
      i. Collect dataset $\{(s_i, a_i, s'_i, r_i)\}$ using some policy, add it to $\mathcal{B}$.
      ii. Repeat:
         A. Sample a batch (s_i, a_i, s'_i, r_i) from $\mathcal{B}$.
         B. ϕ `-=` $\alpha \sum_i \frac{dQ_\phi(s_i, a_i)}{d\phi}(Q_\phi(s_i, a_i) - [r(s_i, a_i) + \gamma \max_{a'} Q_{\phi'}(s'_i, a'_i)])$.

ϕ': parameters from a previous iteration, which are fixed so that the targets are fixed across all of the inner iterations.

We maintain separate target network parameters to avoid having moving targets (targets which change at every step) which might otherwise create instability. Without ϕ', we're trying to fit something different at every step; this will help stabilize the learning of the algorithm.

"Classic" DQN

1. Take some action a_i and observe (s_i, a_i, s'_i, r_i), add it to $\mathcal{B}$.
2. Sample minibatch $\{s_j, a_j, s'_j, r_j\}$ from $\mathcal{B}$ uniformly.
3. Compute $y_j = r_j + \gamma \max_{a'_j} Q_{\phi'}(s'_j, a'_j)$ using target network $Q_{\phi'}$.
4. ϕ `-=` $\alpha \sum_i \frac{dQ_\phi(s_i, a_i)}{d\phi}(Q_\phi(s_i, a_i) - y_j)$.