Week 5 Precept Notes: Q-learning and Deep Q-Learning

March 4, 2024

1 Introduction

Last week in precept, we covered dynamic programming and policy gradient methods. Recall that dynamic programming requires access to the transition dynamics of the MDP we are trying to solve. On the other hand, policy gradient methods are on-policy, thus requiring more data.

For today's precept, we will be discussing a new method called *Q-learning*, which very similar to value iteration, but for the case that the environment dynamics is not available. We will discuss properties of Q-learning, including the fact that it is on-policy. Furthermore, we will discuss the necessary adaptations to move from tabular Q-learning to Deep Q-learning, allowing the technique to be applied to more complex domains (such as Atari in the well-known 2015 DQN paper).

2 Q-learning

Recall the following algorithmic description of value iteration:

Algorithm 1 Value Iteration

Require: Initial value estimate V_0, number of iterations N

for $k \in \{1, \dots, N\}$ **do**
  for $s \in \mathcal{S}$ **do**
    $V_k(s) \leftarrow \max_a r(s,a) + \gamma \mathbb{E}_{s' \sim p(\cdot \mid s,a)} [V_{k-1}(s')]$

The expectation here prevents us from applying value iteration to a general RL problem, since it requires knowing the transition dynamics p. The key insight in going from value iteration to tabular Q-learning is recognizing that this expectation can be estimated using a single sample (s, a, r, s') through *bootstrapping*.

Algorithm 2 Tabular Q-Learning

Require: step size α, exploration strategy (e.g. ε-greedy) `strat`

Initialize state s and Q-value table $Q(\cdot, \cdot)$.
while True **do**
  Choose an action a using $Q(s, \cdot)$ and `strat`.
  Take action a, observe (r, s')
  $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s,a)]$
  if s' is not terminal **then**
    $s \leftarrow s'$
  else
    Reinitialize s.

What is "bootstrapping"? Bootstrapping here refers to the use of the Q-values as targets themselves when updating Q-values. Note that this is in contrast to policy gradient methods which, instead of truncating, uses the full sum of rewards.

We can think of this algorithmic choice in the context of a bias-variance tradeoff. The target value from bootstrapping is a low variance estimate of the true value due to the presence of two terms in the target, but is potentially high bias depending on the quality of the value function (and is definitely high bias at the beginning of training). On the other hand, the target value from a full Monte-Carlo estimate is zero bias (it is always correct in expectation), but is high variance in general.

One could imagine a middle ground between these two extremes; in particular, we can perform n-step TD learning, where we use rewards from n future steps before bootstrapping with our current Q-function estimate. We can also consider all of these n-step estimates simultaneously using an exponential weighing as in $TD(\lambda)$, where λ is the exponential parameter.

What is with that update rule? Since we only have a single sample estimate of $r(s, a)$, we cannot set $Q(s, a)$ to be exactly equal to $r + \gamma \max_{a'} Q(s', a')$. Note that we can rewrite the update rule as

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha \underbrace{\left[r + \gamma \max_{a'} Q(s', a')\right]}_{\text{target value}}.$$

Therefore, what the update rule is doing is actually slowly moving the Q-value estimate towards the single-sample estimates we observe, by keeping track of an exponentially-moving average. We use an exponentially-moving average here since we expect the Q-function to be changing a lot as we get more samples, so we want to weigh recent samples much more.

Why is it "off-policy"? Q-learning is considered off-policy because the target value that it computes to perform updates does not depend on the current policy. In particular, note that we only need samples from a behavioral policy in the form of (s, a, r, s') samples to perform the update, but we do not need the policy itself. This is because the target value uses the maximization operator to choose the action at s'. This is in contrast to the policy gradient-based methods we explored last week, which require $\log \pi_\theta$ at the current θ.

Note that if we instead choose a' using the current policy, we obtain an on-policy algorithm called *Sarsa*.

What are problems to keep in mind? Q-learning in general suffers from what is known as a *maximization bias*. To understand where it comes from, recall that the target value computes $\max_{a'} Q(s', a')$. Assume for the sake of discussion that $Q^*(s', \cdot) \equiv 0$. Then, throughout the estimation procedure, even if $Q(s', a')$ are unbiased estimates for all a', the target $\max_a Q(s, a)$ that is computed will always be biased upwards because we take the maximum.

This maximization bias is commonly addressed using *double-Q learning*. We will not go into too many details here, but the idea is we keep track of two estimated Q-functions Q_1 and Q_2 that are learned independently. We can then use one of the Q-values to perform the maximization step, and another to perform the evaluation. More concretely, we can use the following target value for training Q_1 (or flip Q_1 and Q_2 to get a target for Q_2):

$$r + \gamma Q_1(s', \operatorname*{argmax}_a Q_2(s', a)).$$

Since the function used to compute the argmax is different from the function used to evaluate the max, this breaks the maximization bias. We note that this technique is used even in more modern deep-learning based approaches that learn a Q-function (e.g. TD3, SAC, REDQ, etc.).

3 Q-learning with Function Approximation: DQN

3.1 Context: The Deadly Triad

In classical RL, there is a concept known as the *deadly triad* of *function approximation, bootstrapping, and off-policy learning*. In more detail:

Function Approximation. Function approximation refers to the use of a limited function class (e.g. neural networks) to approximate a value function over large state/action spaces.

Bootstrapping. Bootstrapping refers to the use of a model output as a target to regress towards (e.g. in tabular Q-learning, having the Q-function itself in the target value).

Off-Policy Learning. Off-policy learning refers to performing updates using transitions that are not generated by the policy used to collect samples.

Observe that tabular Q-learning already incorporates two parts of this deadly triad. These three components, used together, is known to introduce hard-to-manage instabilities in standard RL algorithms. As such, researchers were unsure for a while how they can modify Q-learning to make use of expressive function approximators such as neural networks, a necessary change to apply Q-learning to more complicated problem domains.

3.2 The DQN Algorithm

The DQN algorithm seeks to perform Q-learning using neural networks. Since we are moving away from a table of values, we can no longer perform independent tabular updates like we did in tabular Q-learning[1]. As such, we move away from exponential updates to simply minimizing the squared Bellman error

$$\mathcal{L}(\theta; s, a, r, s') = \left[Q_\theta(s,a) - \text{nograd}\left(r + \gamma \max_{a'} Q_\theta(s', a')\right)\right]^2$$

Note the presence of the stop-gradient operator, which means that we only want to minimize the objective above by fitting $Q_\theta(s,a)$ to the target value, and not fit the target value to $Q_\theta(s,a)$.

We provide a rough sketch of the DQN algorithm below:

[1] In a sense, this is the point of using function approximation; updates to a Q-value of a state-action pair should affect other related state-action pairs.

Algorithm 3 DQN algorithm outline.

Require: Learning rate η, ε-greedy parameter ε, target update frequency f, discount γ

Initialize replay buffer $\mathcal{B}$, Q-network Q_θ, target Q-network Q_{targ}.
Copy θ parameters into Q_{targ}.
for each rollout **do**
    Initialize state s.
    while rollout has not terminated **do**
        Select an action a using $Q_\theta(s, \cdot)$ with ε-greedy strategy.
        Play action a, obtain (r, s').
        Store (s, a, r, s') in $\mathcal{B}$.
        Sample batch $B = \{(s_i, a_i, r_i, s_i')\}$ from replay buffer.
        Take a gradient descent step on θ of size η using the objective

$$L(\theta; B) \coloneqq \frac{1}{|B|} \sum_i \left[Q(s_i, a_i) - \text{nograd} \left(r_i + \gamma \max_{a'} Q_{\text{targ}}(s_i', a') \right) \right]^2.$$

        if f steps since last target update **then**
            Copy current θ parameters into Q_{target}
        $s \leftarrow s'$ if s' is not terminal.

In which situations can we use DQN? DQN is usually used to solve MDPs with **continuous states spaces** and **discrete action spaces**, where maintaining a table of values is infeasible and generalization between states is necessary.

Why does DQN work? Given that DQN makes use of the deadly triad, how is it able to learn anything of interest? There are two important components used in DQN that help with stability: (1) the use of a replay buffer and experience replay, and (2) the use of a target network.

3.2.1 DQN Component I: Experience Replay

Experience replay is the idea of taking the (s, a, r, s') pairs observed during rollouts and saving them in a buffer for future training. This is possible in the case of DQN since it is an off-policy algorithm, and thus the data the Q-function is trained on does not have to come from the current policy. Note that although tabular Q-learning is off-policy, its most basic instantiation usually does not maintain a buffer of experiences. Sampling experiences from a replay buffer helps with stability as (1) it minimizes the impact of temporal correlations, and (2) allows neural networks to make multiple passes through the same points.

What do we mean by correlation? To better understand the correlation issue, consider what would happen if we always took a gradient step using the recent (s, a, r, s') pair as in the tabular case. Since s and s' are going to be similar in many environments, $Q(s, \cdot) \approx Q(s', \cdot)$, and so if we take gradient steps to improve our estimate of both $Q(s, \cdot)$ and $Q(s', \cdot)$ one after the other, our network would tend to overcorrect. Sampling from a replay buffer ensures that the distribution of points that the Q-network trains on looks more like an i.i.d. sample as in supervised learning, allowing the network to learn more quickly.

Making multiple passes. Remember that neural networks require a lot of data to be trained properly, rendering the use of a single point for training impractical. Using experience replay allows us to train a

network on multiple points at once, and make more efficient use of the data since we get to train on each point many times.

3.2.2 DQN Component II: Target Networks

Intuitively, combining bootstrapping with function approximation results in instabilities during optimization, as the target will be constantly changing over the course of training. The use of a target network helps mitigate this problem by maintaining a separate network that is used to compute the value-to-go portion of the target. By only updating the target network occasionally, we can ensure that the bootstrapped "regression" problem that DQN tries to solve behaves more like a supervised regression problem most of the time.

3.2.3 Additional Keywords of Interest

Prioritized experience replay; Polyak-averaged target networks; Dueling DQN; Double DQN; Rainbow DQN; Multistep TD-learning