9
Temporal Difference Learning and Q-Learning

In the previous chapter, we covered several sample-based reinforcement learning algorithms including Fitted Q-Iteration and Approximate Policy Iteration. These methods are sometimes called *batch methods* or *offline methods* because a batch of samples is collected and a fitted value function (or action-value function) is found by minimizing the training error for these samples in one "chunk". Offline methods like these make efficient use of available training data, but are computationally expensive and suffer from high memory consumption as the number of samples increases. They also tend to suffer quite badly from the distribution mismatch problems we described in the last chapter.

In this lecture, we present several *online* techniques that perform an incremental update to a value function estimate after each state transition (x, a, r, x'). [1] Such online methods can learn a policy with relatively low computational and memory cost because the updates are made based on a single state transition. Such a state transition (x, a, r, x') may be referred to in the literature as an *experience*.

[1] Note that in this chapter we have switched from cost to reward r, as is common in the reinforcement learning literature.

First, we present the *Temporal-Difference (TD) method* for online policy evaluation. Next, we present a TD-variant denoted *SARSA* that extends online policy evaluation to the action-value function. Finally, we explore *Q-learning* as a method for finding the action-value function for the optimal policy, and hence finding the optimal policy.

In this lecture, we consider the infinite time-horizon setting, and for notational simplicity will usually assume a deterministic policy. Algorithms generalize to stochastic polices aswell. Recall that the value function for a fixed policy π satisfies the Bellman equation:

$$V^{\pi}(x) = E\left[\sum_{t=0}^{\infty} \gamma^t \, r(x_t, \pi(x_t))\right], \quad \text{where } x_0 = x. \tag{9.0.1}$$

The action-value function for a fixed policy π satisfies:

$$Q^{\pi}(x, a) = r(x, a) + E\left[\sum_{t=1}^{\infty} \gamma^t \, r(x_t, \pi(x_t))\right], \quad \text{where } x_0 = x. \tag{9.0.2}$$

The Bellman equation in this case,

$$\begin{aligned} V^{\pi}(x) &= r(x, \pi(x)) + \gamma \, \mathbb{E}_{p(x'|x,\pi(x))}[V^{\pi}(x')] \\ Q^{\pi}(x, a) &= r(x, a) + \gamma \, \mathbb{E}_{p(x'|x,a)}[Q^{\pi}(x', \pi(x'))]. \end{aligned} \tag{9.0.3}$$

The Bellman equations for the optimal value function V^* and action-value function Q^* of the optimal policy π^* are naturally,

$$\begin{aligned} V^*(x) &= \max_{a' \in \mathbb{A}} \left(r(x,a) + \gamma \, \mathbb{E}_{p(x'|x,a)}[V^*(x')] \right) \\ Q^*(x,a) &= r(x,a) + \gamma \, \mathbb{E}_{p(x'|x,a)}[\max_{a' \in \mathbb{A}} Q^*(x',a')]. \end{aligned} \tag{9.0.4}$$

9.1 *Temporal-Difference Learning*

Temporal-difference (TD) Learning is an online method for estimating the value function for a fixed policy π. The principle idea behind TD-learning is that we can learn about the value function from *every* experience (x, a, r, x') as a robot traverses the environment rather than only at the end of a trajectory or trial. [2]

[2] TD is truly one of the core algorithmic ideas in RL. It forms the heart of *TD-Gammon*, the first algorithm to beat humans at the difficult stochastic game of backgammon. That paper is a masterpiece and set the pattern for modern self-play RL in games. 's outstanding book provides **much** more details on Temporal Difference methods and is highly recommended.

; and R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998

Given an estimate of the value function $\tilde{V}^\pi(x)$ we would like to perform an update in order to minimize the squared loss,[3],

[3] Technically, this squared loss is an estimate of the Bellman error $E_{d^\pi(x)}[\frac{1}{2}\left(V^\pi(x) - \tilde{V}^\pi(x)\right)^2]$ where $d^\pi(x)$ is the probability of a state x being visit under policy π.

$$\mathcal{L} = \frac{1}{2}\left(V^\pi(x) - \tilde{V}^\pi(x)\right)^2. \tag{9.1.1}$$

Since we do not yet know the value function, evaluating this loss requires evaluating equation (9.0.1). Naïvely, this method would require waiting until the end of an episode before updating $\tilde{V}^\pi(x)$. Instead, we estimate $V^\pi(x)$ as $y = r + \gamma \, \tilde{V}^\pi(x')$ and perform an online update for each experience $(x, \pi(x), r, x')$. Plugging this estimate into the loss function we get [4]

$$\mathcal{L}_{\text{approx}}(y, \tilde{V}^\pi) = \frac{1}{2}\left(y - \tilde{V}^\pi(x)\right)^2. \tag{9.1.2}$$

The partial derivative of eq. (9.1.2) with respect to $\tilde{V}^\pi$ is:

[4] Notice the trick that has been played here! We're treating the value estimate of the future state as if it were "correct"– as if it were not a function of the parameters that define our value function. This is, of course, totally incorrect. The *Bellman residual* and the Residual Gradient (`RG`)*isthe"obvious"itemtooptimizeandit'sgradientandwedisc*

$$\nabla_{\tilde{V}^\pi(x)} \mathcal{L}_{\text{approx}} = \left(y - \tilde{V}^\pi(x)\right). \tag{9.1.3}$$

If we assume now a parametric form for $\tilde{V}^\pi(x)$ in terms of parameters θ, we can then use the chain rule we can then express $\nabla_\theta \mathcal{L} = \left(y - \tilde{V}^\pi(x)\right) \nabla_\theta \tilde{V}^\pi(x)$

In the case of a tabular representation (one value for each state), our update rule with a step-size of α would simply be:

$$\begin{aligned} \tilde{V}^\pi(x) &\leftarrow \tilde{V}^\pi(x) + \alpha \left(r + \gamma \tilde{V}^\pi(x') - \tilde{V}^\pi(x)\right) \\ &\leftarrow (1-\alpha)\tilde{V}^\pi(x) + \alpha \left(r + \gamma \tilde{V}^\pi(x')\right). \end{aligned} \tag{9.1.4}$$

The term $\left(r + \gamma \tilde{V}^\pi(x') - \tilde{V}^\pi(x)\right)$ is known as the *TD error*.

By looking at the second line of (9.1.4), one may notice that TD-learning is also closely related to an *exponential moving average*.

Algorithm TD

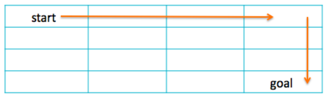
The TD-learning algorithm is shown in Algorithm 17.

Algorithm 17: The TD-learning algorithm.

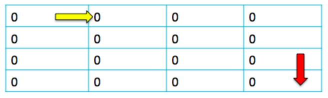
```
Initialize $\tilde{V}^\pi$
while $\tilde{V}^\pi$ not converged do
    Initialize $x$ according to a particular starting state
    while $x$ is not a terminal state do
        apply action $a \leftarrow \pi(x)$
        receive experience $\{x, \pi(x), r, x'\}$
        update $\tilde{V}^\pi(x)$
            $\tilde{V}^\pi(x) \leftarrow (1-\alpha)\tilde{V}^\pi(x) + \alpha\left(r + \gamma\tilde{V}^\pi(x')\right)$
        set $x \leftarrow x'$
return $\tilde{V}^\pi$
```

Grid-World Example

The diagram below shows a grid-based world, where the robot starts in the upper left $(0,0)$, and the goal is in the lower right $(3,3)$. The robot gets a reward of $+1$ if it reaches the goal, and 0 everywhere else. There is a discount factor of γ. The policy is for the robot to go right until it reaches the wall, and then go down.



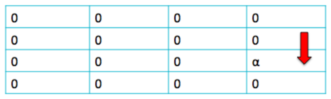
We start by initializing $\tilde{V}^\pi(x) = 0,\ \forall x \in \mathbb{X}$.



As the robot moves one cell over from the start state (yellow arrow above), the reward is 0, and the value of both the current state and the next state is 0, so the approximate gradient used in the update rule (9.1.4) evaluates to 0 and no update is performed. As the robot moves into the goal state (red arrow), the reward is 1, so the approximate gradient evaluates to 1. We then update the second-to-last cell with (9.1.4) and we get:

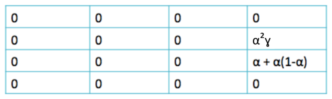
$$\begin{aligned}\tilde{V}^\pi((3,2)) &\leftarrow (1-\alpha)\tilde{V}^\pi((3,2)) + \alpha\left(1 + \gamma\tilde{V}^\pi((3,3))\right)\\ &= (1-\alpha)\times 0 + \alpha \times (1+0) = \alpha.\end{aligned}$$

Another iteration of the algorithm gives us:



0	0	0	0
0	0	0	0
0	0	0	α
0	0	0	0

$$\begin{aligned}
\tilde{V}^{\pi}((3,2)) &\leftarrow (1-\alpha)\tilde{V}^{\pi}((3,2)) + \alpha\left(1 + \gamma\tilde{V}^{\pi}((3,3)\right) \\
&= (1-\alpha)\times\alpha + \alpha\times(1+0) \\
&= \alpha + \alpha\,(1-\alpha), \\
\tilde{V}^{\pi}((3,1)) &\leftarrow (1-\alpha)\tilde{V}^{\pi}((3,1)) + \alpha\left(1 + \gamma\tilde{V}^{\pi}((3,2)\right) \\
&= (1-\alpha)\times 0 + \alpha\times(0+\gamma\times\alpha) \\
&= \alpha^2\,\gamma.
\end{aligned}$$



0	0	0	0
0	0	0	α²γ
0	0	0	α + α(1-α)
0	0	0	0

This method is slow, because we have to run the whole policy just to update the next cell. We will see that SARSA and Q-learning has similar issues of inefficient usage of experience.

9.2 *SARSA*

SARSA extends the Temporal-Difference method presented in the previous section to evaluate policies represented by a action-value functions $Q^{\pi}(x,a)$. Similar to the TD case, we wish to evaluate a policy by performing an online update to obtain an estimate, $\tilde{Q}^{\pi}(x,a)$, of the true action-value function $Q^{\pi}(x,a)$:

$$Q^{\pi}(x,a) = r(x,a) + \sum_{t=1}^{\infty}\gamma^t E[r(x_t,\pi(x_t))] \tag{9.2.1}$$

As in TD, we seek to minimize the loss

$$\mathcal{L}_{\text{approx}} = \frac{1}{2}\left(y - \tilde{Q}^{\pi}(x,a)\right)^2 \tag{9.2.2}$$

where $y = r(x,a) + \gamma\tilde{Q}^{\pi}(x',\pi(x'))$. Following a similar derivation as used for the TD update, we arrive at the SARSA update rule:

$$\tilde{Q}^{\pi}(x,a) \leftarrow (1-\alpha)\tilde{Q}^{\pi}(x,a) + \alpha\left[r(x,a) + \gamma\tilde{Q}^{\pi}(x',\pi(x'))\right]. \tag{9.2.3}$$

Algorithm SARSA

The SARSA algorithm is shown in Algorithm 18.

Algorithm 18: The TD-learning algorithm.

```
Initialize $\tilde{Q}^{\pi}$
while $\tilde{Q}^{\pi}$ not converged do
    Initialize $x$ according to a particular starting state
    while $x$ is not a terminal state do
        apply action $a \leftarrow \pi(x)$
        receive experience $(x, \pi(x), r, x', \pi(x'))$
        update $\tilde{Q}^{\pi}(s)$
            $\tilde{Q}^{\pi}(x,a) \leftarrow (1-\alpha)\tilde{Q}^{\pi}(x,a) + \alpha\left[r(x,a) + \gamma\tilde{Q}^{\pi}(x', \pi(x'))\right]$
        set $x \leftarrow x'$
return $\tilde{V}^{\pi}$
```

One may notice that TD-learning and SARSA are essentially approximate policy evaluation algorithms for the current policy. As a result of that they are examples of *on-policy* methods that can only use samples from the *current* policy to update the value and Q function. *Q-learning*, by contrast, is an *off-policy* method that can use samples from *any* policies [5] to update the optimal action-value function.

[5] Although clearly it requires exploring all actions in all states.

9.3 *Q-Learning*

Q-Learning attempts to estimate the *optimal* action-value function $Q^*(x,a)$ from an online stream of experiences. Recall that the Bellman Equation for the optmal action-value function $Q^*(x,a)$ is,

$$Q^*(x,a) = r(x,a) + \gamma\,\mathbb{E}_{p(x'|x,a)}[\max_{a' \in \mathbb{A}} Q^*(x',a')].$$

Suppose we receive experience (x,a,r,x'). If the transition model is deterministic, we could simply update the action-value function as,

$$\tilde{Q}^*(x,a) \leftarrow r + \gamma \max_{a' \in \mathbb{A}} \tilde{Q}^*(x',a').$$

However, just as in SARSA, this performs poorly when the transition or reward functions are stochastic. Instead, we update $\tilde{Q}^*$ to the weighted sum,

$$\tilde{Q}^*(x,a) \leftarrow \alpha\left[r + \gamma \max_{a' \in \mathbb{A}} \tilde{Q}^*(x',a')\right] + (1-\alpha)\tilde{Q}^*(x,a),$$

where $0 \leq \alpha \leq 1$ is the *learning rate*.

One may notice that we do not need the current policy π to update $\tilde{Q}^*$. Moreover, Q-learning approximates the *optimal* action-value function, the Bellman Equation of which does not depend on the specific policy that the agent is executing. Therefore, Q-learning is an *off-policy* algorithm that can use samples from *any* policies to update $\tilde{Q}^*$. From our experience in the last chapter, one should however, be naturally suspicious of any algorithm that claims to be able to this as it must suffer from *distributional shift* as Value or Policy Iteration do.

Q-learning is guaranteed to converge $\tilde{Q}^*$ to the optimal action-value function Q^* as number of iterations $k \to \infty$ given that the following conditions hold:

1. Each state-action pair is visited infinite times
2. $\lim_{k\to\infty} \sum_{k=0}^{\infty} \alpha_k = \infty$
3. $\lim_{k\to\infty} \sum_{k=0}^{\infty} \alpha_k^2 < \infty$,

where α_k is the learning rate at iteration k. The latter two conditions mean that the learning rate α must be annealed over time. Intuitively, this means that the agent begins by quickly updating $\tilde{Q}^*$, then slows down to refine its estimate as it receives more experience.

Fitted Q-Learning

Just as the fitted Q-iteration algorithm, we can use a function approximator to approximate the action-value function.

Suppose that we approximate Q^* with the function Q_θ with parameter θ. Instead of directly updating our action-value function, we now must update θ to achieve the desired change in Q_θ.

To fit θ, we might choose to minimize a loss function

$$\mathcal{L} = \frac{1}{2}\left(y - Q_\theta(x,a)\right)^2$$

that penalizes deviation between the approximate action-value function $Q_\theta(x,a)$ and the value $y = r + \gamma \max_{a' \in A} Q_\theta(x', a')$ predicted by a Bellman backup.

First, we must derive the "gradient" of $\mathcal{L}$. [6] By applying the chain rule, we find

$$\begin{aligned}\nabla_\theta \mathcal{L} &= \left(y - Q_\theta(x,a)\right)\left[\nabla_\theta y - \nabla_\theta Q_\theta(x,a)\right]\\ &= \left(y - Q_\theta(x,a)\right)\left[\gamma \nabla_\theta Q_\theta(x', a^*) - \nabla_\theta Q_\theta(x,a)\right]\end{aligned}$$

where $a^* = \operatorname{argmax}_{a' \in \mathbb{A}} Q_\theta(x', a')$ is the optimal action according to Q^θ. Unfortunately, it is not possible to obtain an unbiased estimate of $Q_\theta(x,a)\nabla_\theta Q_\theta(x', a^*)$ using one sample (x, a, r, x'). We can find the optimal parameter θ by performing gradient descent on $\mathcal{L}$ with the update rule,

$$\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}. \tag{9.3.1}$$

Q-learning, however, assumes that y is constant and approximates the gradient as

$$\widetilde{\nabla}_\theta \mathcal{L} = -\left(y - Q_\theta(x,a)\right) \nabla_\theta Q_\theta(x,a). \tag{9.3.2}$$

The complete fitted Q-learning update rule is found by substituting eq. (9.3.2) into eq. (9.3.1):

$$\begin{aligned}\theta &\leftarrow \theta + \alpha\left[y - Q_\theta(s,a)\right] \nabla Q_\theta(x,a)\\ &\leftarrow \theta + \alpha\left[\left(r + \gamma Q_\theta(x', a^*)\right) - Q_\theta(x,a)\right] \nabla Q_\theta(x,a).\end{aligned}$$

Bellman Residual Method

Fitted Q-learning as described above does *not* implement gradient descent and, thus, is not guaranteed to converge to a local minimum. The *Bellman residual algorithm* avoids the approximation of eq. (9.3.2) by estimating the true gradient $\nabla_\theta \mathcal{L}$.

$$\nabla_\theta \mathcal{L} = \left(y - Q_\theta(x,a)\right)\left(\gamma\, \nabla_\theta Q_\theta(x', a^*) - \nabla_\theta Q_\theta(x,a)\right).$$

[6] Note again this is the same bogus math where we pretend y is not a function of the parameters. One might naturally ask why not just compute the true gradient? This turns out to be a somewhat nuanced question. One intuitive reason is the notion that our value estimates are likely to better closer to the end of a trial/closer to a goal, and that updates should flow only backwards in time as in dynamic programming updates. If we computed the true gradient (the Bellman residual gradient as it is known , we would have the estimate of the value function in the past changing to more closely match the estimate in the future as well. A further technical difficulty, discussed in Baird's work is that unbiased estimates of the true Bellman residual gradient require **multiple samples** of an action outcome from each state visited. This point of the "derivation" is to give you intuition why you might come up with this rule by thinking about dynamic programming flowing updates backwards and time and the chain rule providing updates.

L. C. Baird. Residual algorithms: Reinforcement learning with function approximation. In *International Conference on Machine Learning*, 1995

This estimation is only unbiased if we can generate two or more independent successor states for taking action a in state s. Generating these samples is trivial if we are able to simulate the system; i.e. have access to a known or learned transition model. If we do not know the transition model, then it is only possible to perform a Bellman residual update if we postpone a backup until the same state-action pair has been observed two or more times. This is often impossible when learning on a real system that has a continuous state-action space.

Exploration Policies

Unlike SARSA, which is an *on-policy* method, Q-learning is an *off-policy* method that can learn from arbitrary (x, a, r, x') experiences, regardless of what policy was used to generate them. This means that it is possible to use an *exploration policy* training that encourages the agent to visit previously unexplored regions of the state-action space. Exploration policies guarantee that the agent visits each state an infinite number of times and ensure convergence when the function is represented by a look-up table.

Two exploration policies that are commonly used with Q-learning are:

1. **ϵ-Greedy**. Choose the greedy action $a = \text{argmax}_{a \in \mathbb{A}} \tilde{Q}(x, a)$ with probability $1 - \epsilon$. Otherwise, with probability ϵ, choose an action uniformly at random $a \sim \text{uniform}(\mathbb{A})$. Higher values of ϵ encourage more exploration. Usually we set ϵ close to 1 as learning starts, and decay $\epsilon \to 0$ as we go along.
2. **Boltzmann Exploration.** Choose action a with probability

$$\pi(a|x) = \frac{\exp\left[\beta \tilde{Q}(x, a)\right]}{\sum_{a' \in \mathbb{A}} \exp\left[\beta \tilde{Q}(x, a')\right]},$$

which is weighted towards selecting actions with higher $\tilde{Q}$-values. Lower values of β encourage more exploration: the exploration policy with $\beta = 0$ is essentially a uniform distribution, as $\beta \to \infty$ the exploration policy becomes the greedy policy

$$\pi(a|x) = \arg\max_{a' \in \mathbb{A}} \tilde{Q}(x, a').$$

Hence, we usually start with β close to 0 and gradually increase β.

9.4 *Experience Replay and Replay Buffers*

Q-learning and SARSA are computationally efficient, but make inefficient use of data. Unlike batch methods, each sample is only used exactly once. This means that the agent must observe each transition ((x, a, r, x') for Q-learning and (x, a, r, x', a') for SARSA) many times to propagate the reward backwards in time.

Experience relay allows Q-learning to re-use experience multiple times by building a database D of experiences under the currently policy, denoted the *replay buffer*. Once enough data has been collected, the agent performs a fixed number of Q-learning or SARSA updates on the batch. This technique bridges the gap between offline methods and online methods, and can potentially combine the advantages the two.

Moreover, because Q-learning is an off-policy algorithm, the experiences generated from previous trajectories and policies can be *re-used* to update the estimate of action-value functions. Therefore, we can use a replay buffer across Q-learning updates: every time a new experience is generated, it is added to the replay buffer, and the agent performs Q-learning updates using *random samples* from the replay buffer.

A common claim in the literature is that experience relay also helps address the problem of *correlated samples* for fitted Q-learning. In the case of online updates, the a experience is likely to be highly correlated with the previous/next experience because they are from the same trajectory. This makes the function approximator easily *overfit* to the current part of the state space, but fail to perform well for the entire state space. However, such correlation is mitigated when we use a batch of samples from possibly different trajectories to update the function approximator.

9.5 *The Philosophy of Temporal Differences**

This section develops a view of Bellman errors and temporal differences that is less well-studied in the literature. It provides some larger philosophical insight as well as proof techniques, and perhaps will be the root of important technical tools in the future. but can skipped for readers eager to move on to other control approaches.

While the theory of Markov Decision Processes has become a powerful foundation for reasoning about temporal difference algorithms, we might argue that there is a still more fundamental intuition that is being captured in TD-style algorithms. Consider the fully online,*Trace* access model, and the goal of predicting a quantity, like long-term reward, over an arbitrary long time sequence. The fundamental claim at the heart of temporal difference and bellman residual methods is if predictions of long term value are temporarily consistent, then they must also be good proxies for the actual long term reward; equivalently, one cannot make consistent predictions (in the sense of temporal differences) and fail to correctly predict the long term reward.

This notion is quite robust to both noise and imperfect approximation. In fact, we can show still stronger claims: if a learner's predictions compete with the best predictor in a class of learners, then they will also compete with the best in that same class at the goal of long-term value estimation. That is, the errors need not even be small– doing as well as possible at consistency implies doing well at long-term prediction as well. This holds over any possible noise sequence– even adversarial noise, establishing the centrality of the notion of Bellman consistency. The central idea is that methods such as `TD` and `RG` should be fundamentally understood as *online algorithms* as opposed to standard gradient minimization methods, and that one cannot simultaneously make consistent predictions in the sense of `TD` and `BE` while doing a poor job in terms of long-run predictions.

This basic model of relating long-term prediction and temporal differences was established in the work of Schapire and Warmuth [1996]. We follow the analysis of Sun and Bagnell [2015] to provide simple guarantees for a wide class of algorithms.

Problem Setting

Consider a sequence of observations (note they need not have the semantics of state!) that can either be Markovian as we've assumed this far in the lecture, or even adversarially chosen. We define the observation at time step t as $\mathbf{x}_t \in \mathbb{R}^n$, which represents features of the environment at time-step t. Let's assume that feature vector $\mathbf{x}$ is bounded as $\|\mathbf{x}\|_2 \leq X$. The corresponding reward at step t is defined as $r_t \in \mathbb{R}$, where we assume that reward is always bounded $|r| \leq R \in \mathbb{R}^+$. Given a sequence of observations $\{\mathbf{x}_t\}$ and a sequence of rewards $\{r_t\}$, the long-term reward at t is defined as $y_t = \sum_{k=t}^{\infty} \gamma^{k-t} r_k$, where $\gamma \in [0, 1)$ is a discounted factor. Note there is no expectation being taken here because there is no assumption of a probabilistic environment. Given a function space $\mathcal{F}$, the learner chooses a predictor f at each time step from $\mathcal{F}$ for predicting long-term rewards. In this section, we assume that any prediction made by a predictor f at a state $\mathbf{x}$ is upper bounded as $|f(\mathbf{x})| \leq P \in \mathbb{R}^+$, for any $f \in \mathcal{F}$ and $\mathbf{x}$.

At time step $t = 0$, the learner receives $\mathbf{x}_0$, initializes a predictor $f_0 \in \mathcal{F}$ and makes prediction $\hat{y}_0$ of y_0 as $f_0(\mathbf{x}_0)$. Rounds of learning then proceed as follows: the learner makes a prediction $\hat{y}_t$ of y_t at step t as $f_t(\mathbf{x}_t)$; the learner observes a reward r_t and the next state $\mathbf{x}_{t+1}$; the learner updates its predictor to f_{t+1}. This interaction repeats and is terminated after T steps.

We denote the goal of estimating the long-term discounted sequence of rewards in this setting as the problem of *online prediction of long-term reward*, PE.

Definitions

We first define the *signed Bellman Error* at step t for predictor f_t as $b_t = f_t(\mathbf{x}_t) - r_t - \gamma f_t(\mathbf{x}_{t+1})$, which measures effectively how self-consistent f_t is in its predictions between time step t and $t+1$. We define the corresponding *Bellman Loss* at time step t with respect to predictor f as:

$$\ell_t^b(f) := (f(\mathbf{x}_t) - r_t - \gamma f(\mathbf{x}_{t+1}))^2. \tag{9.5.1}$$

The *Signed Prediction Error* of long-term reward at t for f_t is defined as $e_t = f_t(\mathbf{x}_t) - y_t$ and $e_t^* = f^*(\mathbf{x}_t) - y_t$ for f^* accordingly. What we actually care about is the long term *Prediction Error* (PE) e_t^2 of a given algorithm in terms of the best possible PE within our class of hypotheses. [7]

[7] To lighten notation in the following sections, all sums over time indices implicitly run from 0 to $T-1$ unless explicitly noted otherwise.

We can also define an online version of *TD Loss* at step t as:

$$\ell_t^d(f) := (f(\mathbf{x}_t) - r_t - \gamma f_t(\mathbf{x}_t))^2. \tag{9.5.2}$$

While we won't consider proving any results in this section, we note that more sophisticated versions of the arguments for Bellman error can be applied to the TD-error allowing us to develop a theory and new set of algorithms. The results are more difficult and more limited so we defer those to the literature [8]. [9]

[8] Wen Sun and J. Andrew (Drew) Bagnell. Online bellman residual and temporal difference algorithms with predictive error guarantees. In *Proceedings of The 25th International Joint Conference on Artificial Intelligence - IJCAI 2016*, April 2016

[9] As we noted earlier, though classic TD algorithm's update step is extremely similar to stochastic gradient descent, there is actually no well-defined objective function on which TD is performing stochastic gradient descent. The online view however provides a clear sequence of objectives and a clear goal of regret minimization.

Understanding Bootstrapping in Online Learning Setting

The true loss that a learner should care about is PE: $(f_t(\mathbf{x}_t) - y_t)^2$. However directly apply no-regret online algorithms on PE is not realistic in practice since in order to get y_t—the discounted sum of future rewards, one has to wait to get all rewards $\{r_i\}$ (or some truncation of this) for $t \leq i \leq T$. On the other hand, the algorithms we've discuss in this chapter use bootstrapping, which leverages the current predictor f_t to estimate y_t as $y_t \approx r_t + \gamma f_t(\mathbf{x}_{t+1})$.

This suggests a different perspective on temporal difference learning and residual gradient learning: *In the online learning setting, RG and TD both could be understood as running Online Gradient Descent on Bellman loss ℓ_t^b and TD loss ℓ_t^d, respectively.*

At every time step t, after receiving the Bellman loss $\ell_t^b(f)$, let us consider what happens if apply online gradient descent on $\ell_t^b(f)$:

$$f_{t+1} = f_t - \mu_t b_t(\nabla_f f_t(\mathbf{x}_t) - \gamma \nabla_f f_t(\mathbf{x}_{t+1})), \tag{9.5.3}$$

where we denote $\nabla_f f(\mathbf{x})$ as the functional gradient of the evaluation functional $f(\mathbf{x})$ at function f.[10] Now for linear function approximation where $f(\mathbf{x})$ is represented as $\mathbf{w}^T\mathbf{x}$, the update step in Eq. 9.5.3 becomes:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \mu_t(\mathbf{w}_t^T\mathbf{x}_t - r_t - \gamma \mathbf{w}_t^T\mathbf{x}_{t+1})(\mathbf{x}_t - \gamma\mathbf{x}_{t+1}), \tag{9.5.4}$$

which reveals the RG algorithm proposed by [11]. [12]

[10] We assume the function $\nabla_f f(\mathbf{x})$ belongs to $\mathcal{F}$. This is true for function classes such as Reproducing Kernel Hilbert Space (RKHS).

[11]

[12] Without the use of *double samples* to handle stochasticity.

Online Gradient Descent is one of the popular no-regret online learning algorithms. The above perspective suggests that RG and TD could be understood as applying a special no-regret online algorithm—OGD, to the Bellman loss and TD loss. This new perspective then naturally motivates the question: can any other no-regret online algorithms, such as Online Newton step, Online Frank Wolf and implicit online learning, be applied to Bellman loss ℓ_t^b and TD* loss $\ell_t^{d^*}$, and achieve guarantees on the long term loss we actually care about PE?

More formally, what one might hope is that if we can find a time-sequence of predictors that achieve the no-regret guarantee on TD loss $\{\ell_t^d\}$ or Bellman loss ℓ_t^b, then for the sequence of predictors $\{f_t\}$, the **real loss** we care about $\sum e_t^2$ could also be upper bounded via some competitive ratio:

$$\lim_{T\to\infty} \frac{1}{T}\sum e_t^2 \leq C\frac{1}{T}\sum e_t^{*2}, \quad \forall f^* \in \mathcal{F}, \tag{9.5.5}$$

where $C \in \mathbb{R}^+$ is constant. Schapire and Warmuth [1996] and later Li [2008] proved variants of this for particular gradient-style algorithms on these loss functions.

Bounding Long Term Predictive Regret

What we can show is that if an online algorithm running on the sequence of loss $\{l_t(f)\}$ is no-regret and the generated sequence of predictors $\{f_t\}$ satisfies a stability condition (we'll detail below), prediction error can indeed be upper bounded in the form of Eq. 9.5.5. The analysis is elementary, and use only a telescoping of the error terms combined with classical Cauchy-Schwartz bounds. What makes the analysis cool, besides the simplicity of the toolkit requires, is that it does **not** place any probabilistic assumption whatever on the sequence of observations $\{\mathbf{x}_t\}$ nor any assumption on the form of predictors $f \in \mathcal{F}$ (e.g., $f(\mathbf{x})$ does not have to be linear).

We start by first showing two key lemmas below:

Lemma 3. *Let us define $d_t = f_t(\mathbf{x}_t) - r_t - \gamma f_{t+1}(\mathbf{x}_{t+1})$. We have:*

$$\sum d_t^2 \geq (1-\gamma)^2 \sum e_t^2 + (\gamma^2 - \gamma)(e_T^2 - e_0^2). \tag{9.5.6}$$

Proof: At the heart of dynamic programming is a tele-scoping of terms. Schapire and Warmuth [1996] used such to a telescopng of term to implicitly show that $d_t = (f_t(\mathbf{x}) - v_t + v_t - (r + \gamma f_{t+1}(\mathbf{x}_{t+1}))) = (e_t - \gamma e_{t+1})$. Squaring both sides and summing over from $t = 0$ to $t = T - 1$, we get:

$$\begin{aligned}
\sum d_t^2 &= \sum (e_t - \gamma e_{t+1})^2 \\
&= \sum e_t^2 + \gamma^2 \sum e_{t+1}^2 - 2\gamma \sum e_t e_{t+1} \\
&\geq \sum e_t^2 + \gamma^2 \sum e_{t+1}^2 - \gamma \sum e_t^2 - \gamma \sum e_{t+1}^2 \\
&= (1-\gamma)^2 \sum e_t^2 + (\gamma^2 - \gamma)(e_T^2 - e_0^2). \qquad (9.5.7)
\end{aligned}$$

The first inequality is obtained by applying Young's inequality to $2e_t e_{t+1}$ to get $2e_t e_{t+1} \leq e_t^2 + e_{t+1}^2$. □ ■

In words, this tells us squared TD *lossupperboundsthelongtermpredictiveerror*(*moduloaboundaryterms*).

Lemma 4. *For any $f^* \in \mathcal{F}$, the prediction error $\sum e_t^{*2}$ upper bounds the BE $\sum b_t^{*2}$ as follows:*

$$\sum b_t^{*2} \leq (1+\gamma)^2 \sum e_t^{*2} + (\gamma + \gamma^2)(e_0^{*2} - e_T^{*2}). \qquad (9.5.8)$$

The proof of Lemma 4 is very similar to the one for Lemma. 3 and left as an exercise.

Now let us define a measure of the change in predictors between the steps of the online algorithm as $\epsilon_t = f_t(\mathbf{x}_{t+1}) - f_{t+1}(\mathbf{x}_{t+1})$, which is closely related to notions of online stability. We're going to require this change to be asymptotically controlled to get good performance. Then b_t and d_t are then closely related with each other by ϵ_t:

$$\begin{aligned}
d_t &= f_t(\mathbf{x}_t) - r_t - \gamma f_{t+1}(\mathbf{x}_{t+1}) - \gamma f_t(\mathbf{x}_{t+1}) + \gamma f_t(\mathbf{x}_{t+1}) \\
&= b_t + \gamma \epsilon_t.
\end{aligned}$$

Squaring both sides, we get:

$$\begin{aligned}
d_t^2 &= b_t^2 + 2b_t\gamma\epsilon_t + \gamma^2\epsilon^2 \leq b_t^2 + b_t^2 + \gamma^2\epsilon_t^2 + \gamma^2\epsilon_t^2 \\
&= 2b_t^2 + 2\gamma^2\epsilon_t^2, \qquad (9.5.9)
\end{aligned}$$

where the first inequality is coming from applying Young's inequality to $2b_t\gamma\epsilon_t$ to get $2b_t\gamma\epsilon_t \leq b_t^2 + \gamma^2\epsilon_t^2$. In words, this tells us the signed temporal difference is exactly the signed Bellman error added to how much the our predictors disagree on x_{t+1}, and thus we can upper bound the TD *errorbythe* BE *andastabilityterm.Thistightlyconnectstheideaoftemporaldifferenceandbellmanerrors*.

We are now ready to state the following main theorem of this paper:

Theorem 5. *Assume a sequence of predictors $\{f_t\}$ is generated by running some online algorithm on the sequence of loss functions $\{l_t\}$. For any predictor $f^* \in \mathcal{F}$, the sum of prediction errors $\sum e_t^2$ can be upper bounded as:*

$$\begin{aligned}
(1-\gamma)^2 \sum e_t^2 \leq & 2\sum (b_t^2 - b_t^{*2}) + 2\gamma^2 \sum \epsilon_t^2 \\
& + 2(1+\gamma)^2 \sum e_t^{*2} + M, \qquad (9.5.10)
\end{aligned}$$

where

$$M = 2(\gamma + \gamma^2)(e_0^{*2} - e_T^{*2}) - (\gamma^2 - \gamma)(e_T^2 - e_0^2).$$

By running a no-regret and online stable algorithm on the loss functions $\{l_t(f)\}$, as $T \to \infty$, the average prediction error is then asymptotically upper bounded by a constant factor of the best possible prediction error in the function class:

$$\lim_{T\to\infty} : \frac{\sum e_t^2}{T} \leq \frac{2(1+\gamma)^2}{(1-\gamma)^2} \frac{\sum e_t^{*2}}{T}. \qquad (9.5.11)$$

Proof: Combining Lemma. 3 and Lemma. 4, we have:

$$\begin{aligned}&\sum d_t^2 - 2\sum b_t^{*2}\\ &\geq (1-\gamma)^2 \sum e_t^2 + (\gamma^2 - \gamma)(e_T^2 - e_0^2)\\ &- 2(1+\gamma)^2 \sum e_t^{*2}\\ &- 2(\gamma+\gamma^2)(e_0^{*2} - e_T^{*2}). \end{aligned} \tag{9.5.12}$$

Subtracting $2b_t^{*2}$ on both sides of Eq. 9.5.9, and then summing over from $t = 1$ to $T - 1$, we have:

$$\sum d_t^2 - \sum 2b_t^{*2} \leq 2\sum (b_t^2 - b_t^{*2}) + 2\gamma^2 \sum \epsilon_t^2.$$

Combining the above two inequalities together, we have:

$$\begin{aligned}&2\sum (b_t^2 - b_t^{*2}) + 2\gamma^2 \sum \epsilon_t^2\\ &\geq (1-\gamma)^2 \sum e_t^2 + (\gamma^2 - \gamma)(e_T^2 - e_0^2)\\ &- 2(1+\gamma)^2 \sum e_t^{*2} - 2(\gamma+\gamma^2)(e_0^{*2} - e_T^{*2}). \end{aligned} \tag{9.5.13}$$

Rearrange inequality (9.5.13) and define $M = 2(\gamma+\gamma^2)(e_0^{*2} - e_T^{*2}) - (\gamma^2 - \gamma)(e_T^2 - e_0^2)$, we obtain inequality (9.5.10)

Assume that the $\bar{f} = \arg\min_{f \in \mathcal{F}} \sum l_t(f)$, then if the online algorithm is no-regret, we have

$$\begin{aligned}&\frac{1}{T}\sum b_t^2 - b_t^{*2} = \frac{1}{T}\sum l_t(f_t) - l_t(f^*)\\ &\leq \frac{1}{T}\sum l_t(f_t) - l_t(\bar{f})\\ &= \frac{1}{T}\text{Regret} \leq 0, \quad T \to \infty. \end{aligned} \tag{9.5.14}$$

If, further, our choice of online algorithm satisfies a *stability condition*, we can remove a term. For the generated sequence of predictors f_t, we say the algorithm is online stable if:

$$\lim_{T\to\infty} \frac{1}{T}\sum (f_t(\mathbf{x}_{t+1}) - f_{t+1}(\mathbf{x}_{t+1}))^2 = 0. \tag{9.5.15}$$

Intuitively, this form online stability means that on average the difference between successive predictors is eventually small on average. Online stability is a general condition and does not notably limit the scope of the online learning algorithms. [13]

[13] For instance, when f is linear, the definition of stability of online learning in (see Eq. 3 in) and imply this kind of stability. In fact, almost all popular no-regret online learning algorithms satisfy this condition. Moreover, it turns out that this stability is actually essential– without this requirement we can generate counterexamples to the having any competitive ratio at all.

Ankan Saha, Prateek Jain, and Ambuj Tewari. The Interplay Between Stability and Regret in Online Learning. *arXiv preprint arXiv:1211.6158*, pages 1–19, 2012. URL http://arxiv.org/abs/1211.6158; Ankan Saha, Prateek Jain, and Ambuj Tewari. The Interplay Between Stability and Regret in Online Learning. *arXiv preprint arXiv:1211.6158*, pages 1–19, 2012. URL http://arxiv.org/abs/1211.6158; and Stephane Ross and J. Andrew Bagnell. Stability Conditions for Online Learnability. *arXiv:1108.3154*, 2011. URL http://arxiv.org/abs/1108.3154

Thus, assuming stability, we have: $\frac{1}{T}\sum \epsilon_t^2 = 0$ when $T \to \infty$.

Also, since we assume $|f(\mathbf{x})| \leq P$ and $|r| \leq R$, we can see M must be upper bounded by some constant. Hence, we must have $\frac{M}{T} = 0$, as $T \to \infty$.

Under the conditions that the online algorithm is no-regret and satisfies online stability, we get Eq. 9.5.11 by dividing both sides of Eq. 9.5.10 by T and taking T to infinity. □ ■

Note that in the Thm. 5, Eq. 9.5.10 holds for any $f^* \in \mathcal{F}$, including the f^* that minimizes the prediction error.

Conclusion

It is interesting that we can derive and prove bounds for a wide class of algorithms without making the markov assumption. A natural question is whether we can use these approaches to design algorithms that are more robust to distributional shift, or to understand the superior performance of online methods like TD as compared with offline methods like fitted value-iteration.