Part II: Approximate Solution Methods In the second part, we are dealing with arbitrarily large state spaces; in these cases, it is hard to find π_* or v_*, q_*. Our goal instead is to find a good approximate solution using limited computational resources.

The problem with large state spaces is threefold: memory, time and data. In many of our target tasks, almost every state encountered will never have been seen before. The solution is *generalization*. How can experience with a limited subset of the state space be usefully generalized to produce a good approximation over a much larger subset?

To some extent, we need only combine RL methods with existing generalization methods. One of frequent use is *function approximation*, which takes examples from a desired function (e.g. a value function) and attempts to generalize from them to construct an approximation of the entire function.

Function approximation is an instance of supervised learning.

RL with function approximation involves a number of new issues (usually not in SL), such as nonstationarity, bootstrapping, and delayed targets.

Chap 9 (prediction): on-policy training, policy is given and only its value function is approximated

Chap 10 (control): an approximation to the optimal policy is found

Chap 11: off-policy learning with function approximation

Chap 12: *eligibility traces*, which dramatically improves the computational properties of multi-step RL methods in many cases.s

Chap 13 (control): *policy-gradient*, a different approach to control, which approximates the optimal policy directly and need never form an approximate value function.

1 On-policy Prediction with Approximation

We start consider using function approximation in estimating the state-value function from on-policy data, which approximates v_π from experience generated using a known policy π.

The approximate value function is represented as a parameterized functional form with weight vector $\boldsymbol{w} \in \mathbb{R}^d$, instead of a table. We write $\hat{v}(s, \boldsymbol{w}) \approx v_\pi(s)$ for the approximate value of state s given weight vector $\boldsymbol{w}$.

For example,

- $\hat{v}$ = linear function in features of the state, $\boldsymbol{w}$ = vector of feature weights
- $\hat{v}$ = function computed by a multi-layer ANN, $\boldsymbol{w}$ = vector of connection weights in all the layers
- $\hat{v}$ = function computed by a decision tree, $\boldsymbol{w}$ = all the numbers defining the split points and leaf values of the tree

Typically, the number of weights is much less than the number of states ($d << |S|$), and one weight takes care of many states.

Extending RL to function approximation also makes it applicable to partially observable problems (in which the full state is not available to the agent).

All theoretical results for methods using function approximation apply equally well to cases of partial observability.

Function approximation cannot augment the state representation with memories of past observations.

1.1 Value-function Approximation

Definition 1.1 ***individual update notation***: *$s \to u$ (the book uses $|- >$, where s is the state updated and u is the update target. The estimated value for s should be more like u.*

- MC: $S_t \to G_t$
- TD(0): $S_t \to R_{t+1} + \gamma \hat{v}(S_{t+1}, \boldsymbol{w}_t)$
- n-step TD: $S_t \to G_{t:t+n}$
- DP: $s \to \mathbb{E}_\pi[R_{t+1} + \gamma \hat{v}(S_{t+1}, \boldsymbol{w}_t) | S_t = s]$

We want to extend a one-to-one mapping to one-to-many mapping, i.e., updating s will also update many other states.

Supervised learning methods learn to mimic input-output examples, and when the outputs are numbers, like u, the process is often called *function approximation*. We use these methods for value prediction simply by passing to them the $s \to u$ of each update as a training example.

In principle, we can use any method for supervised learning from examples, including ANN, decision trees, and various kinds of multivariate regression. However, in RL, we require the methods are able to handle nonstationarity cases as RL agents need to do online learning (interact with the environment to obtain data).

1.2 The Prediction Objective ($\overline{VE}$)

By assumption we have far more states than weights, so making one state's estimate more accurate invariably(always) means making others' less accurate. We need to find which states we care most about.

Definition 1.2 ***state distribution*** $\mu(s)$*:* $\mu \geq 0$, $\sum_s \mu(s) = 1$, *representing how much we care about the error (square distance between approx value* $\hat{v}(s, \boldsymbol{w})$ *and true value* $v_\pi(s)$*) in each state* s.

Definition 1.3 ***Mean Squared Value Error*** **(**$\overline{VE}$**)***: a natural objective function,*

$$\overline{VE}(\boldsymbol{w}) \doteq \sum_{s \in \S} \underbrace{\mu(s)}_{weight} \underbrace{\left[v_\pi(s) - \hat{v}(s, \boldsymbol{w})\right]^2}_{error}$$

Often $\mu(s)$ is chosen to be the fraction of time spent in s. Under on-policy training, this is called *on-policy distribution*. In continuing tasks, the on-policy distribution is the stationary distribution under π. The book talks about the episodic tasks (not including here).

!! NOTE that the RL community is not sure if $\overline{VE}$ is the right performance objective for RL. But this is the best we can find so far.

1.3 Stochastic-gradient and Semi-gradient Methods

In GD, the weight vector is a column vector with a fixed number of real valued components, $\boldsymbol{w} \doteq (w_1, w_2, ..., w_d)^T$, and the approximate value function $\hat{v}(s, \boldsymbol{w})$ is a differentiable function of $\boldsymbol{w}$ for all $s \in S$. However, no $\boldsymbol{w}$ can gets all the states in real world.

GD vs SGD In both GD and SGD, we update a set of parameters in an iterative manner to minimize an error function.

- GD: we run through ALL the samples to do a single update for a parameter in a particular iteration (which is not feasible)
- SGD: we run through ONLY ONE or a SUBSET of samples to do the update. If we use a SUBSET, it is called Minibatch Stochastic Gradient Descent.

The solution to $\boldsymbol{w}$ is SGD: we minimize error on the *observed samples*. The weight vector of SGD is:

$$\boldsymbol{w}_{t+1} \doteq \boldsymbol{w} - \frac{1}{2}\alpha\nabla[v_\pi(S_t) - \hat{v}(S_t, \boldsymbol{w}_t)]^2$$

$$\boldsymbol{w}_{t+1} = \boldsymbol{w} + \alpha[v_\pi(S_t) - \hat{v}(S_t, \boldsymbol{w}_t)]\nabla\hat{v}(S_t, \boldsymbol{w}_t)$$

Now we consider when we do not have the exact true value $v_\pi(S_t)$, but an approximate value U_t. Then

$$\boldsymbol{w}_{t+1} = \boldsymbol{w} + \alpha[U_t - \hat{v}(S_t, \boldsymbol{w}_t)]\nabla\hat{v}(S_t, \boldsymbol{w}_t)$$

If U_t is an *unbiased* estimate ($\mathbb{E}[U_t|S_t = s] = v_\pi(s)$), then $\boldsymbol{w}_t$ is guaranteed to converge to a local optimum under the usual stochastic approximation conditions (chap2). For example, Monte Carlo target $U_t \doteq G_t$ is an unbiased estimate of $v_\pi(S_t)$.

```
Gradient Monte Carlo for Estimating $\hat{v} \approx v_\pi$

Input: $\pi$ to be evaluated, a differentiable function $\hat{v} : S \times \mathbb{R}^d \to \mathbb{R}$
Parameter: $\alpha > 0$
Init the value-function weights $\boldsymbol{w} \in \mathbb{R}^d$ arbitrarily
Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, S_1, A_1, ..., R_T, S_T$ using $\pi$
  Loop for each step of episode, $t = 0, 1, ..., T-1$:
  Take action $A$; observe resultant reward, $R$, and state, $S'$
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha[G_t - \hat{v}(S_t, \boldsymbol{w})]\nabla\hat{v}(S_t, \boldsymbol{w})$
```

On the other hand, Bootstrapping targets, such as n-step returns $G_{t:t+n}$ or the DP target, all depend on the current value of the weight vector $\boldsymbol{w}_t$, which implies that they will be biased. These are not the true gradient-descent methods, but $semi$-gradient methods.

Although semi-gradient (bootstrapping) methods do not converge as robustly as true gradient methods, they offer important advantages: (1) enable much faster learning; (2) enable learning to be continual and online; (3) provides computational advantages. A typical semi-gradient method is semi-gradient TD(0), which uses $U_t \doteq R_{t+1} + \gamma\hat{v}(S_t, \boldsymbol{w})$

```
Semi-gradient TD(0) for Estimating $\hat{v} \approx v_\pi$

Input: $\pi$ to be evaluated, a differentiable function $\hat{v} : S \times \mathbb{R}^d \to \mathbb{R}$ where $\hat{v}(terminal, \dot{}) = 0$
Parameter: $\alpha > 0$
Init the value-function weights $\boldsymbol{w} \in \mathbb{R}^d$ arbitrarily
Loop forever (for each episode):
    Init $S$
    Loop for each step of episode:
        Choose $A\ \pi(\dot{|}S)$
        Take action $A$; observe reward, $R$, and state, $S'$
        $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha[R + \gamma\hat{v}(S_t, \boldsymbol{w} - \hat{v}(S, \boldsymbol{w})]\nabla\hat{v}(S_t, \boldsymbol{w})$
        $S \leftarrow S'$
    until $S$ is terminal
```

The target U_t in semi-gradient methods is biased, so the $\boldsymbol{w}$ may not converge to a local optimum.

state aggregation a simple form of generalizing function approximation in which states are grouped together, with one estimated value for each group.

1.4 Linear Methods

One of the most important special cases of function approximation is that, $\hat{v}(\dot{;}\boldsymbol{w})$ is a linear function of the weight vector $\boldsymbol{w}$. Corresponding to every state s, there is a real-valued vector $\boldsymbol{x}(s) \doteq (x_1(s), x_2(s), ..., x_d(s))^T$, with the same number of components as $\boldsymbol{w}$.

Linear methods approximate the state-value function by the inner product between $\boldsymbol{w}$ and $\boldsymbol{x}(s)$:

$$\hat{v}(s, \boldsymbol{w}) \doteq \boldsymbol{w}^T \boldsymbol{x}(s) \doteq \sum_{i=1}^{d} w_i x_i(s)$$

The vector $\boldsymbol{x}(s)$ is a *feature vector* representing state s. Each component $x_i(s)$ is a *feature*.

The gradient of the approximation value function w.r.t $\boldsymbol{w}$ is then $\nabla\hat{v}(s,\boldsymbol{w}) = \boldsymbol{x}(s)$. Thus, we can simplify the general SGD update to

$$\boldsymbol{w}_{t+1} = \boldsymbol{w} + \alpha[U_t - \hat{v}(S_t, \boldsymbol{w}_t)]\boldsymbol{x}(s)$$

Understand the fixed point of linear TD learning Let us start with TD update with linear function approximation on the first line, recall the value of a state $\hat{v}(s,\boldsymbol{w}) \doteq \boldsymbol{w}^T\boldsymbol{x}(s)$

$$\begin{aligned}
\boldsymbol{w}_{t+1} &\doteq \boldsymbol{w} + \alpha[R_{t+1} + \gamma\hat{v}(S_{t+1}, \boldsymbol{w}_t) - \hat{v}(S_t, \boldsymbol{w}_t)]\boldsymbol{x}(S_t) \\
&= \boldsymbol{w} + \alpha[R_{t+1} + \gamma\hat{v}(S_{t+1}, \boldsymbol{w}_t) - \hat{v}(S_t, \boldsymbol{w}_t)]\boldsymbol{x}_t \quad \text{simplify notation} \\
&= \boldsymbol{w} + \alpha[R_{t+1} + \gamma\boldsymbol{w}_t^T\boldsymbol{x}_{t+1} - \boldsymbol{w}_t^T\boldsymbol{x}_t]\boldsymbol{x}_t \quad \text{due to } \hat{v}(s,\boldsymbol{w}) \doteq \boldsymbol{w}^T\boldsymbol{x}(s) \\
&= \boldsymbol{w} + \alpha[\underbrace{R_{t+1}\boldsymbol{x}_t}_{\boldsymbol{b}} - \underbrace{\boldsymbol{x}_t(\boldsymbol{x}_t - \gamma\boldsymbol{x}_{t+1})^T}_{\boldsymbol{A}}\boldsymbol{w}_t] \quad \text{transpose doesn't change scalar}
\end{aligned}$$

The TD update can be rewritten as the expected update + a noise term, so it is largely dominated by the behavior of the expected update.

$$\begin{aligned}
\mathbb{E}[\Delta\boldsymbol{w}_t] &= \alpha(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{w}_t) \\
\boldsymbol{b} &= \mathbb{E}[R_{t+1}\boldsymbol{x}_t] \\
\boldsymbol{A} &= \mathbb{E}[\boldsymbol{x}_t(\boldsymbol{x}_t - \gamma\boldsymbol{x}_{t+1})^T] \quad \text{expct. over features}
\end{aligned}$$

Definition 1.4 ***TD Fixed Point*** $\boldsymbol{w}_{TD}$ *the weight is converged when the expected TD update = 0.*

$$\mathbb{E}[\Delta\boldsymbol{w}_{TD}] = \alpha(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{w}_{TD}) = 0$$

Describe a theoretical guarantee on the mean squared value error at the TD fixed point

1.5 Learning Objectives (UA RL MOOC)

Lesson 1: Estimating Value Functions as Supervised Learning

1. Understand how we can use parameterized functions to approximate value functions

$\hat{v}(s, \boldsymbol{w})$

2. Explain the meaning of linear value function approximation

$\hat{v}(s, \boldsymbol{w}) = \sum w_i x_i(s)$

3. Recognize that the tabular case is a special case of linear value function approximation.

Consider each state has a corresponding vector (one-hot encoding)

4. Understand that there are many ways to parameterize an approximate value function

ANN, decision tree etc

5. Understand what is meant by generalization and discrimination

generalization: updates to the value estimate of one state influence the value of other states

discrimination:the ability to make the values for two states different to distinguish between the values for these two state.

6. Understand how generalization can be beneficial

Generalization can speed up learning.

7. Explain why we want both generalization and discrimination from our function approximation

Because we want generalization to speed up learning, and discrimination to distinguish really different states.

8. Understand how value estimation can be framed as a supervised learning problem

Supervised learning involves approximating a function given a dataset of (input, target) pairs. For Monte-Carlo (S_t, G_t). For TD $(S_t, R_t + \gamma \hat{v}(S_t, \boldsymbol{w}))$

9. Recognize not all function approximation methods are well suited for reinforcement learning

Some are not suitable because they are designed for a fixed batch of data (offline learning, contrast to RL's online learning), or not designed for temporally correlated data (data in RL is always correlated).

Lesson 2: The Objective for On-policy Prediction

10. Understand the mean-squared value error objective for policy evaluation

$$\overline{VE}(\boldsymbol{w}) \doteq \sum_{s\in\S} \underbrace{\mu(s)}_{\text{weight}} \underbrace{\left[v_\pi(s) - \hat{v}(s, \boldsymbol{w})\right]^2}_{\text{error}}$$

11. Explain the role of the state distribution in the objective

$\mu(s)$ specifies how much we care for each state.

12. Understand the idea behind gradient descent and stochastic gradient descent

Calculus! (1) the gradient indicates the direction to increase/decrease the weight vectors; (2) the gradient gives the direction of steepest ascent

13. Outline the gradient Monte Carlo algorithm for value estimation

see chap 9.3

14. Understand how state aggregation can be used to approximate the value function

Group multiple states into one grouped state. One feature for each grouped state.

15. Apply Gradient Monte-Carlo with state aggregation

see chap 9.3

Lesson 3: The Objective for TD

16. Understand the TD-update for function approximation

17. Highlight the advantages of TD compared to Monte-Carlo

TD enables faster learning, continual and online learning; provides computational advantages

18. Outline the Semi-gradient TD(0) algorithm for value estimation

see chap 9.3

19. Understand that TD converges to a biased value estimate

The TD target depends on our estimate of the value in the next state. This means our update could be biased because the estimate in our target may be inaccurate.

20. Understand that TD converges much faster than Gradient Monte Carlo

TD can learn during the episode and has a lower variance update.

Lesson 4: Linear TD

21. Derive the TD-update with linear function approximation

$$\boldsymbol{w}_{t+1} = \boldsymbol{w} + \alpha[U_t - \hat{v}(S_t, \boldsymbol{w}_t)]\boldsymbol{x}(s)$$

22. Understand that tabular TD(0) is a special case of linear semi-gradient TD(0)

linear TD is a strict generalization on both tabular TD and TD with state aggregation. Recall the feature vector is like one-hot encoded vector, where the i-th=1, else 0 corresponds to i-th weight.

23. Highlight the advantages of linear value function approximation over nonlinear

- linear methods are simpler to understand and analyze mathematically
- with good features, linear methods can learn quickly and achieve good prediction accuracy