A Guided Tour of Chapter 9: Reinforcement Learning for Prediction

Ashwin Rao

ICME, Stanford University

RL does not have access to a probability model

- DP/ADP assume access to probability model (knowledge of $\mathcal{P}_R$)
- Often in real-world, we do not have access to these probabilities
- Which means we'd need to *interact* with the *actual environment*
- Actual Environment serves up individual experiences, not probabilities
- Even if MDP model is available, model updates can be challenging
- Often real-world models end up being too large or too complex
- Sometimes estimating a *sampling model* is much more feasible
- So RL interacts with either *actual* or *simulated* environment
- Either way, we receive *individual experiences* of next state and reward
- RL learns Value Functions from a stream of individual experiences
- How does RL solve Prediction and Control with such limited access?

The RL Approach

- Like humans/animals, RL doesn't aim to estimate probability model
- Rather, RL is a "trial-and-error" approach to linking actions to returns
- This is hard because actions have overlapping reward sequences
- Also, sometimes actions result in *delayed rewards*
- The key is incrementally updating Q-Value Function from experiences
- Appropriate Approximation of Q-Value Function is also key to success
- RL algorithms are founded on the *Bellman Equations*
- Moreover, RL Control is based on *Generalized Policy Iteration*
- This lecture/chapter focuses on RL for Prediction

RL for Prediction

- Prediction: Problem of estimating MDP Value Function for a policy π
- Equivalently, problem of estimating π-implied MRP's Value Function
- Assume interface serves an *atomic experience* of (next state, reward)
- Interacting with this interface repeatedly provides a *trace experience*

$$S_0, R_1, S_1, R_2, S_2, \ldots$$

- *Value Function* $V : \mathcal{N} \rightarrow \mathbb{R}$ of an MRP is defined as:

$$V(s) = \mathbb{E}[G_t | S_t = s] \text{ for all } s \in \mathcal{N}, \text{ for all } t = 0, 1, 2, \ldots$$

where the *Return* G_t for each $t = 0, 1, 2, \ldots$ is defined as:

$$G_t = \sum_{i=t+1}^{\infty} \gamma^{i-t-1} \cdot R_i = R_{t+1} + \gamma \cdot R_{t+2} + \gamma^2 \cdot R_{t+3} + \ldots = R_{t+1} + \gamma \cdot G_{t+1}$$

Code interface for RL Prediction

An *atomic experience* is represented as a TransitionStep[S]

```
@dataclass(frozen=True)
class TransitionStep(Generic[S]):
    state: S
    next_state: S
    reward: float
```

Input to RL prediction can be either of:

- Atomic Experiences as Iterable[TransitionStep[S]], or
- Trace Experiences as Iterable[Iterable[TransitionStep[S]]]

Note that Iterable can be either a Sequence or an Iterator (i.e., *stream*)

Monte-Carlo (MC) Prediction

- Supervised learning with states and returns from trace experiences
- Incremental estimation with update method of FunctionApprox
- x-values are states S_t, y-values are returns G_t
- Note that updates can be done only at the end of a trace experience
- Returns calculated with a backward walk: $G_t = R_{t+1} + \gamma \cdot G_{t+1}$

$$\mathcal{L}_{(S_t, G_t)}(\boldsymbol{w}) = \frac{1}{2} \cdot (V(S_t; \boldsymbol{w}) - G_t)^2$$

$$\nabla_{\boldsymbol{w}} \mathcal{L}_{(S_t, G_t)}(\boldsymbol{w}) = (V(S_t; \boldsymbol{w}) - G_t) \cdot \nabla_{\boldsymbol{w}} V(S_t; \boldsymbol{w})$$

$$\Delta \boldsymbol{w} = \alpha \cdot (G_t - V(S_t; \boldsymbol{w})) \cdot \nabla_{\boldsymbol{w}} V(S_t; \boldsymbol{w})$$

Structure of the parameters update formula

$$\Delta \boldsymbol{w} = \alpha \cdot (G_t - V(S_t; \boldsymbol{w})) \cdot \nabla_{\boldsymbol{w}} V(S_t; \boldsymbol{w})$$

The update $\Delta \boldsymbol{w}$ to parameters $\boldsymbol{w}$ should be seen as product of:

- *Learning Rate* α
- *Return Residual* of the observed return G_t relative to the estimated conditional expected return $V(S_t; \boldsymbol{w})$
- *Estimate Gradient* of the conditional expected return $V(S_t; \boldsymbol{w})$ with respect to the parameters $\boldsymbol{w}$

This structure (as product of above 3 entities) will be a repeated pattern.

Tabular MC Prediction

- Finite state space, let's say non-terminal states $\mathcal{N} = \{s_1, s_2, \ldots, s_m\}$
- Denote $V_n(s_i)$ as estimate of VF after the n-th occurrence of s_i
- Denote $Y_i^{(1)}, Y_i^{(2)}, \ldots, Y_i^{(n)}$ as returns for first n occurrences of s_i
- Denote count_to_weight_func attribute of Tabular as $f(\cdot)$
- Then the Tabular update at the n-th occurrence of s_i is:

$$\begin{aligned} V_n(s_i) &= (1 - f(n)) \cdot V_{n-1}(s_i) + f(n) \cdot Y_i^{(n)} \\ &= V_{n-1}(s_i) + f(n) \cdot (Y_i^{(n)} - V_{n-1}(s_i)) \end{aligned}$$

- So update to VF for s_i is *Latest Weight* times *Return Residual*
- For default setting of count_to_weight_func as $f(n) = \frac{1}{n}$:

$$V_n(s_i) = \frac{n-1}{n} \cdot V_{n-1}(s_i) + \frac{1}{n} \cdot Y_i^{(n)} = V_{n-1}(s_i) + \frac{1}{n} \cdot (Y_i^{(n)} - V_{n-1}(s_i))$$

Tabular MC Prediction

- Expanding the incremental updates across values of n, we get:

$$V_n(s_i) = f(n) \cdot Y_i^{(n)} + (1 - f(n)) \cdot f(n-1) \cdot Y_i^{(n-1)} + \ldots$$
$$\ldots + (1 - f(n)) \cdot (1 - f(n-1)) \cdots (1 - f(2)) \cdot f(1) \cdot Y_i^{(1)}$$

- For default setting of count_to_weight_func as $f(n) = \frac{1}{n}$:

$$V_n(s_i) = \frac{1}{n} \cdot Y_i^{(n)} + \frac{n-1}{n} \cdot \frac{1}{n-1} \cdot Y_i^{(n-1)} + \ldots$$
$$\ldots + \frac{n-1}{n} \cdot \frac{n-2}{n-1} \cdots \frac{1}{2} \cdot \frac{1}{1} \cdot Y_i^{(1)} = \frac{\sum_{k=1}^{n} Y_i^{(k)}}{n}$$

- Tabular MC is simply incremental calculation of averages of returns
- Exactly the calculation in the update method of Tabular class
- View Tabular MC as an application of Law of Large Numbers

Tabular MC as a special case of Linear Func Approximation

- Features functions are indicator functions for states
- Linear-approx parameters are Value Function estimates for states
- count_to_weight_func plays the role of learning rate
- So tabular Value Function update can be written as:

$$w_i^{(n)} = w_i^{(n-1)} + \alpha_n \cdot (Y_i^{(n)} - w_i^{(n-1)})$$

- $Y_i^{(n)} - w_i^{(n-1)}$ represents the gradient of the loss function
- For non-stationary problems, algorithm needs to "forget" distant past
- With constant learning rate α, time-decaying weights:

$$\begin{aligned} V_n(s_i) &= \alpha \cdot Y_i^{(n)} + (1-\alpha) \cdot \alpha \cdot Y_i^{(n-1)} + \ldots + (1-\alpha)^{n-1} \cdot \alpha \cdot Y_i^{(1)} \\ &= \sum_{j=1}^{n} \alpha \cdot (1-\alpha)^{n-j} \cdot Y_i^{(j)} \end{aligned}$$

- Weights sum to 1 asymptotically: $\lim_{n \to \infty} \sum_{j=1}^{n} \alpha \cdot (1-\alpha)^{n-j} = 1$

Each-Visit MC and First-Visit MC

- The MC algorithm we covered is known as *Each-Visit Monte-Carlo*
- Because we include each occurrence of a state in a trace experience
- Alternatively, we can do *First-Visit Monte-Carlo*
- Only the first occurrence of a state in a trace experience is considered
- Keep track of whether a state has been visited in a trace experience
- MC Prediction algorithms are easy to understand and implement
- MC produces unbiased estimates but can be slow to converge
- Key disadvantage: MC requires complete trace experiences

Temporal-Difference (TD) Prediction

- To understand TD, we start with Tabular TD Prediction
- Key: Exploit recursive structure of VF in MRP Bellman Equation
- Replace G_t with $R_{t+1} + \gamma \cdot V(S_{t+1})$ using atomic experience data
- So we are *bootstrapping* the VF ("estimate from estimate")
- The tabular MC Prediction update (for constant α) is modified from:

$$V(S_t) \leftarrow V(S_t) + \alpha \cdot (G_t - V(S_t))$$

to:

$$V(S_t) \leftarrow V(S_t) + \alpha \cdot (R_{t+1} + \gamma \cdot V(S_{t+1}) - V(S_t))$$

- $R_{t+1} + \gamma \cdot V(S_{t+1})$ known as *TD target*
- $\delta_t = R_{t+1} + \gamma \cdot V(S_{t+1}) - V(S_t)$ known as *TD Error*
- TD Error is the crucial quantity - it represents "sample Bellman Error"
- VF is adjusted so as to bridge TD error (on an expected basis)

TD updates after each atomic experience

- Unlike MC, we can use TD when we have incomplete traces
- Often in real-world situations, experiments gets curtailed/disrupted
- Also, we can use TD in non-episodic (known as *continuing*) traces
- TD updates VF after each atomic experience ("continuous learning")
- So TD can be run on *any* stream of atomic experiences
- This means we can chop up the input stream and serve in any order

TD Prediction with Function Approximation

- Each atomic experience leads to a parameters update
- To understand how parameters update work, consider:

$$\mathcal{L}_{(S_t,S_{t+1},R_{t+1})}(\boldsymbol{w}) = \frac{1}{2} \cdot (V(S_t;\boldsymbol{w}) - (R_{t+1} + \gamma \cdot V(S_{t+1};\boldsymbol{w})))^2$$

- Above formula replaces G_t (of MC) with $R_{t+1} + \gamma \cdot V(S_{t+1}, \boldsymbol{w})$
- Unlike MC, in TD, we don't take the gradient of this loss function
- "Cheat" in gradient calc by ignoring dependency of $V(S_{t+1};\boldsymbol{w})$ on $\boldsymbol{w}$
- This "gradient with cheating" calculation is known as *semi-gradient*
- So we pretend the only dependency on $\boldsymbol{w}$ is through $V(S_t;\boldsymbol{w})$

$$\Delta\boldsymbol{w} = \alpha \cdot (R_{t+1} + \gamma \cdot V(S_{t+1};\boldsymbol{w}) - V(S_t;\boldsymbol{w})) \cdot \nabla_{\boldsymbol{w}} V(S_t;\boldsymbol{w})$$

Structure of the parameters update formula

$$\Delta \boldsymbol{w} = \alpha \cdot (R_{t+1} + \gamma \cdot V(S_{t+1}; \boldsymbol{w}) - V(S_t; \boldsymbol{w})) \cdot \nabla_{\boldsymbol{w}} V(S_t; \boldsymbol{w})$$

The update $\Delta \boldsymbol{w}$ to parameters $\boldsymbol{w}$ should be seen as product of:

- *Learning Rate* α
- *TD Error* $\delta_t = R_{t+1} + \gamma \cdot V(S_{t+1}; \boldsymbol{w}) - V(S_t; \boldsymbol{w})$
- *Estimate Gradient* of the conditional expected return $V(S_t; \boldsymbol{w})$ with respect to the parameters $\boldsymbol{w}$

So parameters update formula has same product-structure as MC

TD's many benefits

- "TD is the most significant and innovative idea in RL" - Rich Sutton
- Blends the advantages of DP and MC
- Like DP, TD learns by bootstrapping (drawing from Bellman Eqn)
- Like MC, TD learns from experiences without access to probabilities
- So TD overcomes curse of dimensionality and curse of modeling
- TD also has the advantage of not requiring entire trace experiences
- Most significantly, TD is akin to human (continuous) learning

Bias, Variance and Convergence of TD versus MC

- MC uses G_t is an unbiased estimate of the Value Function
- This helps MC with convergence even with function approximation
- TD uses $R_{t+1} + \gamma \cdot V(S_{t+1}; \boldsymbol{w})$ as a biased estimate of the VF
- Tabular TD prediction converges to true VF in the mean for const α
- And converges to true VF under Robbins-Monro learning rate schedule

$$\sum_{n=1}^{\infty} \alpha_n = \infty \text{ and } \sum_{n=1}^{\infty} \alpha_n^2 < \infty$$

- However, Robbins-Monro schedule is not so useful in practice
- TD Prediction with func-approx does not always converge to true VF
- Most convergence proofs are for Tabular, some for linear func-approx
- TD Target $R_{t+1} + \gamma \cdot V(S_{t+1}; \boldsymbol{w})$ has much lower variance that G_t
- G_t depends on many random rewards whose variances accumulate
- TD Target depends on only the next reward, so lower variance

Speed of Convergence of TD versus MC

- We typically compare algorithms based on:
  - Speed of convergence
  - Efficiency in use of limited set of experiences data
- There are no formal proofs for MC v/s TD on above criterion
- MC and TD have significant differences in their:
  - Usage of data
  - Nature of updates
  - Frequency of updates
- So unclear exactly how to compare them apples to apples
- Typically, MC and TD are compared with constant α
- Practically/empirically, TD does better than MC with constant α

Approximate Policy Evaluation code

```
def update(v: FunctionApprox[S]) -> FunctionApprox[S]:
    nt_states: Sequence[S] = \
        non_terminal_states_distribution.sample_n(
            num_state_samples
        )

    def return_(s_r: Tuple[S, float]) -> float:
        s, r = s_r
        return r + gamma * v.evaluate([s]).item()

    return v.update(
        [(s, mrp.transition_reward(s).expectation(
            return_)) for s in nt_states]
    )

return iterate(update, approx_0)
```

Approximate Value Iteration interface

```
def value_iteration(
    mdp: MarkovDecisionProcess[S, A],
    gamma: float,
    approx_0: FunctionApprox[S],
    non_terminal_states_distribution: Distribution[S],
    num_state_samples: int
) -> Iterator[FunctionApprox[S]]:
```

Approximate Value Iteration code

```
def update(v: FunctionApprox[S]) -> FunctionApprox[S]:
    nt_states: Sequence[S] = \
        non_terminal_states_distribution.sample_n(
            num_state_samples
        )

    def return_(s_r: Tuple[S, float]) -> float:
        s, r = s_r
        return r + gamma * v.evaluate([s]).item()

    return v.update(
        [(s, max(mdp.step(s, a).expectation(return_)
                 for a in mdp.actions(s)))
         for s in nt_states]
    )
return iterate(update, approx_0)
```

Finite-Horizon Approximate Dynamic Programming

- Similarly, generalize Backward Induction DP algorithms
- Each time steps' Value Function is a FunctionApprox
- Work with a separate MRP/MDP representation for each time step's transitions, that is responsible for sampling next step's (state, reward)
- x-values come from current time step's states sampling distribution
- y-values come from applying Bellman Operator on next time steps' FunctionApprox for it's Value Function
- Bellman Operator expectation is estimated by averaging over transition samples
- These (x, y) pairs constitute the data-set used to solve the current time step's FunctionApprox for it's Value Function

Constructing the Non-Terminal States Distribution

- Each ADP algorithm works with a distribution of non-terminal states
- Good choice: Stationary Distribution of uniform-policy-implied MRP
- See if you can use some mathematical property of given MDP/MRP
- Or create sampling traces and estimate with occurrence frequency
- Backup choice: Uniform Distribution of all non-terminal states
- Likewise, for backward induction, see if you can utilize some property of the given process to infer distribution of states for a fixed time step
- eg: In finance, continuous-time processes can sometimes be solved
- Or create sampling traces and estimate with occurrence frequency
- Backup choice: Uniform Distribution of all non-terminal states

Key Takeaways from this Chapter

- The FunctionApprox interface involves three key methods:
  - solve: Calculate the "best-fit" parameters that minimizes the cross-entropy loss function for the given fixed data set of (x, y) pairs
  - update: Parameters of FunctionApprox are updated based on each new (x, y) pairs data set from the available data stream
  - evaluate: Calculate the conditional expectation of response variable y, according to the model specified by FunctionApprox
- Tabular is a special case of linear function approximation with feature functions as indicator functions for each of the finite set of $\mathcal{X}$
- All the Tabular DP algorithms can be generalized to ADP algorithms
  - Tabular VF updates replaced by updates to FunctionApprox parameters
  - Sweep over all states in Tabular case replaced by state samples
  - Bellman Operators' Expectation estimated as average of calculations over transition samples (versus using explicit transition probabilities)