Foundations of Reinforcement Learning with Applications in Finance

Ashwin Rao, Tikhon Jelvis

1 Batch RL, Experience-Replay, DQN, LSPI, Gradient TD

In Chapters **??** and **??**, we covered the basic RL algorithms for Prediction and Control respectively. Specifically, we covered the basic Monte-Carlo (MC) and Temporal-Difference (TD) techniques. We want to highlight two key aspects of these basic RL algorithms:

1. The experiences data arrives in the form of a single unit of experience at a time (single unit is a *trace experience* for MC and an *atomic experience* for TD), the unit of experience is used by the algorithm for Value Function learning, and then that unit of experience is not used later in the algorithm (essentially, that unit of experience, once consumed, is *not re-consumed* for further learning later in the algorithm). It doesn't have to be this way - one can develop RL algorithms that re-use experience data - this approach is known as *Experience-Replay* (in fact, we saw a glimpse of Experience-Replay in Section **??** of Chapter **??**).

2. Learning occurs in a *granularly incremental* manner, by updating the Value Function after each unit of experience. It doesn't have to be this way - one can develop RL algorithms that take an entire batch of experiences (or in fact, all of the experiences that one could possibly get), and learn the Value Function directly for that entire batch of experiences. A key idea here is that if we know in advance what experiences data we have (or will have), and if we collect and organize all of that data, then we could directly (i.e., not incrementally) estimate the Value Function for *that* experiences data set. This approach to RL is known as *Batch RL* (versus the basic RL algorithms we covered in the previous chapters that can be termed as *Incremental RL*).

Thus, we have a choice or doing Experience-Replay or not, and we have a choice of doing Batch RL or Incremental RL. In fact, some of the interesting and practically effective algorithms combine both the ideas of Experience-Replay and Batch RL. This chapter starts with the coverage of Batch RL and Experience-Replay. Then, we cover some key algorithms (including Deep Q-Networks and Least Squares Policy Iteration) that effectively leverage Batch RL and/or Experience-Replay. Next, we look deeper into the issue of the *Deadly Triad* (that we had alluded to in Chapter **??**) by viewing Value Functions as Vectors (we had done this in Chapter **??**), understand Value Function Vector transformations with a balance of geometric intuition and mathematical rigor, providing insights into convergence issues for a variety of traditional loss functions used to develop RL algorithms. Finally, this treatment of Value Functions as Vectors leads us in the direction of overcoming the Deadly Triad by defining an appropriate loss function, calculating whose gradient provides a more robust set of RL algorithms known as Gradient Temporal Difference (abbreviated, as Gradient TD).

1.1 Batch RL and Experience-Replay

Let us understand Incremental RL versus Batch RL in the context of fixed finite experiences data. To make things simple and easy to understand, we first focus on understanding the difference for the case of MC Prediction (i.e., to calculate the Value Function of an MRP using Monte-Carlo). In fact, we had covered this setting in Section **??** of Chapter **??**.

To refresh this setting, specifically we have access to a fixed finite sequence/stream of MRP trace experiences (i.e., `Iterable[Iterable[TransitionStep[S]]]`), which we know can be converted to returns-augmented data of the form `Iterable[Iterable[ReturnStep[S]]]` (using the `returns` function[1]). Flattening this data to `Iterable[ReturnStep[S]]` and extracting from it the (state, return) pairs gives us the fixed, finite training data for MC Prediction, that we denote as follows:

$$\mathcal{D} = [(S_i, G_i) | 1 \leq i \leq n]$$

We've learnt in Chapter **??** that we can do an Incremental MC Prediction estimation $V(s; \boldsymbol{w})$ by updating $\boldsymbol{w}$ after each MRP trace experience with the gradient calculation $\nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w})$ for each data pair (S_i, G_i), as follows:

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

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

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

The Incremental MC Prediction algorithm performs n updates in sequence for data pairs $(S_i, G_i), i = 1, 2, \ldots, n$ using the `update` method of `FunctionApprox`. We note that Incremental RL makes inefficient use of available training data $\mathcal{D}$ because we essentially "discard" each of these units of training data after it's used to perform an update. We want to make efficient use of the given data with Batch RL. Batch MC Prediction aims to estimate the MRP Value Function $V(s; \boldsymbol{w}^*)$ such that

$$\begin{aligned}\boldsymbol{w}^* &= \arg\min_{\boldsymbol{w}} \frac{1}{2n} \cdot \sum_{i=1}^{n} (V(S_i; \boldsymbol{w}) - G_i)^2 \\ &= \arg\min_{\boldsymbol{w}} \mathbb{E}_{(S,G)\sim\mathcal{D}}[\frac{1}{2} \cdot (V(S; \boldsymbol{w}) - G)^2]\end{aligned}$$

This in fact is the `solve` method of `FunctionApprox` on training data $\mathcal{D}$. This approach is called Batch RL because we first collect and store the entire set (batch) of data $\mathcal{D}$ available to us, and then we find the best possible parameters $\boldsymbol{w}^*$ fitting this data $\mathcal{D}$. Note that unlike Incremental RL, here we are not updating the MRP Value Function estimate while the data arrives - we simply store the data as it arrives and start the MRP Value Function estimation procedure once we are ready with the entire (batch) data $\mathcal{D}$ in storage. As we know from the implementation of the `solve` method of `FunctionApprox`, finding the best possible parameters $\boldsymbol{w}^*$ from the batch $\mathcal{D}$ involves calling the `update` method of `FunctionApprox` with repeated use of the available data pairs (S, G) in the stored data set $\mathcal{D}$. Each of these updates to the parameters $\boldsymbol{w}$ is as follows:

$$\Delta \boldsymbol{w} = \alpha \cdot \frac{1}{n} \cdot \sum_{i=1}^{n} (G_i - V(S_i; \boldsymbol{w})) \cdot \nabla_{\boldsymbol{w}} V(S_i; \boldsymbol{w})$$

[1] `returns` is defined in the file rl/returns.py

Note that unlike Incremental MC where each update to $\boldsymbol{w}$ uses data from a single trace experience, each update to $\boldsymbol{w}$ in Batch MC uses all of the trace experiences data (all of the batch data). If we keep doing these updates repeatedly, we will ultimately converge to the desired MRP Value Function $V(s;\boldsymbol{w}^*)$. The repeated use of the available data in $\mathcal{D}$ means that we are doing Batch MC Prediction using *Experience-Replay*. So we see that this makes more efficient use of the available training data $\mathcal{D}$ due to the re-use of the data pairs in $\mathcal{D}$.

The code for this Batch MC Prediction algorithm (`batch_mc_prediction`) is shown below.[2] From the input trace experiences (`traces` in the code below), we first create the set of `ReturnStep` transitions that span across the set of all input trace experiences (`return_steps` in the code below). This involves calculating the return associated with each state encountered in `traces` (across all trace experiences). From `return_steps`, we create the (state, return) pairs that constitute the fixed, finite training data $\mathcal{D}$, which is then passed to the `solve` method of `approx: ValueFunctionApprox[S]`.

```
import rl.markov_process as mp
from rl.returns import returns
from rl.approximate_dynamic_programming import ValueFunctionApprox
import itertools

def batch_mc_prediction(
    traces: Iterable[Iterable[mp.TransitionStep[S]]],
    approx: ValueFunctionApprox[S],
    gamma: float,
    episode_length_tolerance: float = 1e-6,
    convergence_tolerance: float = 1e-5
) -> ValueFunctionApprox[S]:
    '''traces is a finite iterable'''
    return_steps: Iterable[mp.ReturnStep[S]] = \
        itertools.chain.from_iterable(
            returns(trace, gamma, episode_length_tolerance) for trace in traces
        )
    return approx.solve(
        [(step.state, step.return_) for step in return_steps],
        convergence_tolerance
    )
```

Now let's move on to Batch TD Prediction. Here we have fixed, finite experiences data $\mathcal{D}$ available as:

$$\mathcal{D} = [(S_i, R_i, S_i')|1 \leq i \leq n]$$

where (R_i, S_i') is the pair of reward and next state from a state S_i. So, Experiences Data $\mathcal{D}$ is presented in the form of a fixed, finite number of atomic experiences. This is represented in code as an `Iterable[TransitionStep[S]]`.

Just like Batch MC Prediction, here in Batch TD Prediction, we first collect and store the data as it arrives, and once we are ready with the batch of data $\mathcal{D}$ in storage, we start the MRP Value Function estimation procedure. The parameters $\boldsymbol{w}$ are updated with repeated use of the atomic experiences in the stored data $\mathcal{D}$. Each of these updates to the parameters $\boldsymbol{w}$ is as follows:

$$\Delta \boldsymbol{w} = \alpha \cdot \frac{1}{n} \cdot \sum_{i=1}^{n} (R_i + \gamma \cdot V(S_i';\boldsymbol{w}) - V(S_i;\boldsymbol{w})) \cdot \nabla_{\boldsymbol{w}} V(S_i;\boldsymbol{w})$$

Note that unlike Incremental TD where each update to $\boldsymbol{w}$ uses data from a single atomic experience, each update to $\boldsymbol{w}$ in Batch TD uses all of the atomic experiences data (all of

[2] `batch_mc_prediction` is defined in the file rl/monte_carlo.py.

the batch data). The repeated use of the available data in $\mathcal{D}$ means that we are doing Batch TD Prediction using *Experience-Replay*. So we see that this makes more efficient use of the available training data $\mathcal{D}$ due to the re-use of the data pairs in $\mathcal{D}$.

The code for this Batch TD Prediction algorithm (`batch_td_prediction`) is shown below.[3] We create a `Sequence[TransitionStep]` from the fixed, finite-length input atomic experiences $\mathcal{D}$ (`transitions` in the code below), and call the `update` method of `FunctionApprox` repeatedly, passing the data $\mathcal{D}$ (now in the form of a `Sequence[TransitionStep]`) to each invocation of the `update` method (using the function `itertools.repeat`). This repeated invocation of the `update` method is done by using the function `iterate.accumulate`. This is done until convergence (convergence based on the `done` function in the code below), at which point we return the converged `FunctionApprox`.

```
import rl.markov_process as mp
from rl.approximate_dynamic_programming import ValueFunctionApprox, extended_vf
import rl.iterate as iterate
import itertools
import numpy as np

def batch_td_prediction(
    transitions: Iterable[mp.TransitionStep[S]],
    approx_0: ValueFunctionApprox[S],
    gamma: float,
    convergence_tolerance: float = 1e-5
) -> ValueFunctionApprox[S]:
    '''transitions is a finite iterable'''

    def step(
        v: ValueFunctionApprox[S],
        tr_seq: Sequence[mp.TransitionStep[S]]
    ) -> ValueFunctionApprox[S]:
        return v.update([(
            tr.state, tr.reward + gamma * extended_vf(v, tr.next_state)
        ) for tr in tr_seq])

    def done(
        a: ValueFunctionApprox[S],
        b: ValueFunctionApprox[S],
        convergence_tolerance=convergence_tolerance
    ) -> bool:
        return b.within(a, convergence_tolerance)

    return iterate.converged(
        iterate.accumulate(
            itertools.repeat(list(transitions)),
            step,
            initial=approx_0
        ),
        done=done
```

Likewise, we can do Batch TD(λ) Prediction. Here we are given a fixed, finite number of trace experiences

$$\mathcal{D} = [(S_{i,0}, R_{i,1}, S_{i,1}, R_{i,2}, S_{i,2}, \ldots, R_{i,T_i}, S_{i,T_i}) | 1 \leq i \leq n]$$

For trace experience i, for each time step t in the trace experience, we calculate the eligibility traces as follows:

$$\boldsymbol{E}_{i,t} = \gamma\lambda \cdot \boldsymbol{E}_{i,t-1} + \nabla_{\boldsymbol{w}} V(S_{i,t}; \boldsymbol{w}) \text{ for all } t = 1, 1, \ldots T_i - 1$$

with the eligiblity traces initialized at time 0 for trace experience i as $\boldsymbol{E}_{i,0} = \nabla_{\boldsymbol{w}} V(S_{i,0}; \boldsymbol{w})$.

[3] `batch_td_prediction` is defined in the file rl/td.py.

Then, each update to the parameters $\boldsymbol{w}$ is as follows:

$$\Delta \boldsymbol{w} = \alpha \cdot \frac{1}{n} \cdot \sum_{i=1}^{n} \frac{1}{T_i} \cdot \sum_{t=0}^{T_i - 1} (R_{i,t+1} + \gamma \cdot V(S_{i,t+1}; \boldsymbol{w}) - V(S_{i,t}; \boldsymbol{w})) \cdot \boldsymbol{E}_{i,t} \tag{1.1}$$

1.2 A generic implementation of Experience-Replay

Before we proceed to more algorithms involving Experience-Replay and/or Batch RL, it is vital to recognize that the concept of Experience-Replay stands on it's own, independent of it's use in Batch RL. In fact, Experience-Replay is a much broader concept, beyond it's use in RL. The idea of Experience-Replay is that we have a stream of data coming in and instead of consuming it in an algorithm as soon as it arrives, we store each unit of incoming data in memory (which we shall call *Experience-Replay-Memory*, abbreviated as ER-Memory), and use samples of data from ER-Memory (with replacement) for our algorithm's needs. Thus, we are routing the incoming stream of data to ER-Memory and sourcing data needed for our algorithm from ER-Memory (by sampling with replacement). This enables re-use of the incoming data stream. It also gives us flexibility to sample an arbitrary number of data units at a time, so our algorithm doesn't need to be limited to using a single unit of data at a time. Lastly, we organize the data in ER-Memory in such a manner that we can assign different sampling weights to different units of data, depending on the arrival time of the data. This is quite useful for many algorithms that wish to give more importance to recently arrived data and de-emphasize/forget older data.

Let us now write some code to implement all of these ideas described above. The code below uses an arbitrary data type `T`, which means that the unit of data being handled with Experience-Replay could be any data structure (specifically, not limited to the `TransitionStep` data type that we care about for RL with Experience-Replay).

The attribute `saved_transitions: List[T]` is the data structure storing the incoming units of data, with the most recently arrived unit of data at the end of the list (since we `append` to the list). The attribute `time_weights_func` lets the user specify a function from the reverse-time-stamp of a unit of data to the sampling weight to assign to that unit of data ("reverse-time-stamp" means the most recently-arrived unit of data has a time-index of 0, although physically it is stored at the end of the list, rather than at the start). The attribute `weights` simply stores the sampling weights of all units of data in `saved_transitions`, and the attribute `weights_sum` stores the sum of the `weights` (the attributes `weights` and `weights_sum` are there purely for computational efficiency to avoid too many calls to `time_weights_func` and avoidance of summing a long list of weights, which is required to normalize the weights to sum up to 1).

`add_data` appends an incoming unit of data (`transition: T`) to `self.saved_transitions` and updates `self.weights` and `self.weights_sum`. `sample_mini_batches` returns a sample of specified size `mini_batch_size`, using the sampling weights in `self.weights`. We also have a method `replay` that takes as input an `Iterable` of `transitions` and a `mini_batch_size`, and returns an `Iterator` of `mini_batch_sized` data units. As long as the input `transitions: Iterable[T]` is not exhausted, `replay` appends each unit of data in `transitions` to `self.saved_transitions` and then `yields` a `mini_batch_sized` sample of data. Once `transitions: Iterable[T]` is exhausted, it simply `yields` the samples of data. The `Iterator` generated by `replay` can be piped to any algorithm that expects an `Iterable` of the units of data as input, essentially enabling us to replace the pipe carrying an input data stream with a pipe carrying the data stream sourced from ER-Memory.

```
T = TypeVar('T')

class ExperienceReplayMemory(Generic[T]):
    saved_transitions: List[T]
    time_weights_func: Callable[[int], float]
    weights: List[float]
    weights_sum: float

    def __init__(
        self,
        time_weights_func: Callable[[int], float] = lambda _: 1.0,
    ):
        self.saved_transitions = []
        self.time_weights_func = time_weights_func
        self.weights = []
        self.weights_sum = 0.0

    def add_data(self, transition: T) -> None:
        self.saved_transitions.append(transition)
        weight: float = self.time_weights_func(len(self.saved_transitions) - 1)
        self.weights.append(weight)
        self.weights_sum += weight

    def sample_mini_batch(self, mini_batch_size: int) -> Sequence[T]:
        num_transitions: int = len(self.saved_transitions)
        return Categorical(
            {tr: self.weights[num_transitions - 1 - i] / self.weights_sum
             for i, tr in enumerate(self.saved_transitions)}
        ).sample_n(min(mini_batch_size, num_transitions))

    def replay(
        self,
        transitions: Iterable[T],
        mini_batch_size: int
    ) -> Iterator[Sequence[T]]:

        for transition in transitions:
            self.add_data(transition)
            yield self.sample_mini_batch(mini_batch_size)

        while True:
            yield self.sample_mini_batch(mini_batch_size)
```

The code above is in the file rl/experience_replay.py. We encourage you to implement Batch MC Prediction and Batch TD Prediction using this `ExperienceReplayMemory` class.

1.3 Least-Squares RL Prediction

We've seen how Batch RL Prediction is an iterative process of weight updates until convergence - the MRP Value Function is updated with repeated use of the fixed, finite (batch) data that is made available. However, if we assume that the MRP Value Function approximation $V(s; \boldsymbol{w})$ is a linear function approximation (linear in a set of feature functions of the state space), then we can solve for the MRP Value Function with direct and simple linear algebra operations (ie., without the need for iterative weight updates until convergence). Let us see how.

We define a sequence of feature functions $\phi_j : \mathcal{N} \to \mathbb{R}, j = 1, 2, \ldots, m$ and we assume the parameters $\boldsymbol{w}$ is a weights vector $\boldsymbol{w} = (w_1, w_2, \ldots, w_m) \in \mathbb{R}^m$. Therefore, the MRP Value Function is approximated as:

$$V(s; \boldsymbol{w}) = \sum_{j=1}^{m} \phi_j(s) \cdot w_j = \boldsymbol{\phi}(s)^T \cdot \boldsymbol{w} \text{ for all } s \in \mathcal{N}$$

where $\boldsymbol{\phi}(s) \in \mathbb{R}^m$ is the feature vector for state s.

The direct solution of the MRP Value Function using simple linear algebra operations is known as Least-Squares (abbreviated as LS) solution. We start with Batch MC Prediction for the case of linear function approximation, which is known as Least-Squares Monte-Carlo (abbreviated as LSMC).

1.3.1 Least-Squares Monte-Carlo (LSMC)

For the case of linear function approximation, the loss function for Batch MC Prediction with data $[(S_i, G_i)|1 \leq i \leq n]$ is:

$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{2n} \cdot \sum_{i=1}^{n} (\sum_{j=1}^{m} \phi_j(S_i) \cdot w_j - G_i)^2 = \frac{1}{2n} \cdot \sum_{i=1}^{n} (\boldsymbol{\phi}(S_i)^T \cdot \boldsymbol{w} - G_i)^2$$

We set the gradient of this loss function to 0, and solve for $\boldsymbol{w}^*$. This yields:

$$\sum_{i=1}^{n} \boldsymbol{\phi}(S_i) \cdot (\boldsymbol{\phi}(S_i)^T \cdot \boldsymbol{w^*} - G_i) = 0$$

We can calculate the solution $\boldsymbol{w}^*$ as $\boldsymbol{A}^{-1} \cdot \boldsymbol{b}$, where the $m \times m$ Matrix $\boldsymbol{A}$ is accumulated at each data pair (S_i, G_i) as:

$$\boldsymbol{A} \leftarrow \boldsymbol{A} + \boldsymbol{\phi}(S_i) \cdot \boldsymbol{\phi}(S_i)^T \text{ (i.e., outer-product of } \boldsymbol{\phi}(S_i) \text{ with itself)}$$

and the m-Vector $\boldsymbol{b}$ is accumulated at each data pair (S_i, G_i) as:

$$\boldsymbol{b} \leftarrow \boldsymbol{b} + \boldsymbol{\phi}(S_i) \cdot G_i$$

To implement this algorithm, we can simply call `batch_mc_prediction` that we had written earlier by setting the argument `approx` as `LinearFunctionApprox` and by setting the attribute `direct_solve` in `approx: LinearFunctionApprox[S]` as `True`. If you read the code under `direct_solve=True` branch in the `solve` method, you will see that it indeed performs the above-described linear algebra calculations. The inversion of the matrix $\boldsymbol{A}$ is $O(m^3)$ complexity. However, we can speed up the algorithm to be $O(m^2)$ with a different implementation - we can maintain the inverse of $\boldsymbol{A}$ after each (S_i, G_i) update to $\boldsymbol{A}$ by applying the Sherman-Morrison formula for incremental inverse (Sherman and Morrison 1950). The Sherman-Morrison incremental inverse for $\boldsymbol{A}$ is as follows:

$$(\boldsymbol{A} + \boldsymbol{\phi}(S_i) \cdot \boldsymbol{\phi}(S_i)^T)^{-1} = \boldsymbol{A}^{-1} - \frac{\boldsymbol{A}^{-1} \cdot \boldsymbol{\phi}(S_i) \cdot \boldsymbol{\phi}(S_i)^T \cdot \boldsymbol{A}^{-1}}{1 + \boldsymbol{\phi}(S_i)^T \cdot \boldsymbol{A}^{-1} \cdot \boldsymbol{\phi}(S_i)}$$

with $\boldsymbol{A}^{-1}$ initialized to $\frac{1}{\epsilon} \cdot \boldsymbol{I}_m$, where $\boldsymbol{I}_m$ is the $m \times m$ identity matrix, and $\epsilon \in \mathbb{R}^+$ is a small number provided as a parameter to the algorithm. $\frac{1}{\epsilon}$ should be considered to be a proxy for the step-size α which is not required for least-squares algorithms. If ϵ is too small, the sequence of inverses of $\boldsymbol{A}$ can be quite unstable and if ϵ is too large, the learning is slowed.

This brings down the computational complexity of this algorithm to $O(m^2)$. We won't implement the Sherman-Morrison incremental inverse for LSMC, but in the next subsection we shall implement it for Least-Squares Temporal Difference (LSTD).

1.3.2 Least-Squares Temporal-Difference (LSTD)

For the case of linear function approximation, the loss function for Batch TD Prediction with data $[(S_i, R_i, S_i')|1 \leq i \leq n]$ is:

$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{2n} \cdot \sum_{i=1}^{n} (\boldsymbol{\phi}(S_i)^T \cdot \boldsymbol{w} - (R_i + \gamma \cdot \boldsymbol{\phi}(S_i')^T \cdot \boldsymbol{w}))^2$$

We set the semi-gradient of this loss function to 0, and solve for $\boldsymbol{w}^*$. This yields:

$$\sum_{i=1}^{n} \boldsymbol{\phi}(S_i) \cdot (\boldsymbol{\phi}(S_i)^T \cdot \boldsymbol{w^*} - (S_i + \gamma \cdot \boldsymbol{\phi}(S_i')^T \cdot \boldsymbol{w}^*)) = 0$$

We can calculate the solution $\boldsymbol{w}^*$ as $\boldsymbol{A}^{-1} \cdot \boldsymbol{b}$, where the $m \times m$ Matrix $\boldsymbol{A}$ is accumulated at each atomic experience (S_i, R_i, S_i') as:

$$\boldsymbol{A} \leftarrow \boldsymbol{A} + \boldsymbol{\phi}(S_i) \cdot (\boldsymbol{\phi}(S_i) - \gamma \cdot \boldsymbol{\phi}(S_i'))^T \text{ (note the Outer-Product)}$$

and the m-Vector $\boldsymbol{b}$ is accumulated at each atomic experience (S_i, R_i, S_i') as:

$$\boldsymbol{b} \leftarrow \boldsymbol{b} + \boldsymbol{\phi}(S_i) \cdot R_i$$

With Sherman-Morrison incremental inverse, we can reduce the computational complexity from $O(m^3)$ to $O(m^2)$, as follows:

$$(\boldsymbol{A} + \boldsymbol{\phi}(S_i) \cdot (\boldsymbol{\phi}(S_i) - \gamma \cdot \boldsymbol{\phi}(S_i))^T)^{-1} = \boldsymbol{A}^{-1} - \frac{\boldsymbol{A}^{-1} \cdot \boldsymbol{\phi}(S_i) \cdot (\boldsymbol{\phi}(S_i) - \gamma \cdot \boldsymbol{\phi}(S_i'))^T \cdot \boldsymbol{A}^{-1}}{1 + (\boldsymbol{\phi}(S_i) - \gamma \cdot \boldsymbol{\phi}(S_i'))^T \cdot \boldsymbol{A}^{-1} \cdot \boldsymbol{\phi}(S_i)}$$

with $\boldsymbol{A}^{-1}$ initialized to $\frac{1}{\epsilon} \cdot \boldsymbol{I}_m$, where $\boldsymbol{I}_m$ is the $m \times m$ identity matrix, and $\epsilon \in \mathbb{R}^+$ is a small number provided as a parameter to the algorithm.

This algorithm is known as the Least-Squares Temporal-Difference (LSTD) algorithm and is due to Bradtke and Barto (Bradtke and Barto 1996).

Now let's write some code to implement this LSTD algorithm. The arguments `transitions`, `feature_functions`, `gamma` and `epsilon` of the function `least_squares_td` below are quite self-explanatory. This is a batch method with direct calculation of the estimated Value Function from batch data (rather than iterative weight updates), so `least_squares_td` returns the estimated Value Function of type `LinearFunctionApprox[NonTerminal[S]]`, rather than an `Iterator` over the updated function approximations (as was the case in Incremental RL algorithms).

The code below should be fairly self-explanatory. `a_inv` refers to $\boldsymbol{A}^{-1}$ which is updated with the Sherman-Morrison incremental inverse method. `b_vec` refers to the $\boldsymbol{b}$ vector. `phi1` refers to $\boldsymbol{\phi}(S_i)$, `phi2` refers to $\boldsymbol{\phi}(S_i) - \gamma \cdot \boldsymbol{\phi}(S_i')$ (except when S_i' is a terminal state, in which case `phi2` is simply $\boldsymbol{\phi}(S_i)$). The temporary variable `temp` refers to $(\boldsymbol{A}^{-1})^T \cdot (\boldsymbol{\phi}(S_i) - \gamma \cdot \boldsymbol{\phi}(S_i'))$ and is used both in the numerator and denominator in the Sherman-Morrison formula to update $\boldsymbol{A}^{-1}$.

```
from rl.function_approx import LinearFunctionApprox
import rl.markov_process as mp
import numpy as np

def least_squares_td(
    transitions: Iterable[mp.TransitionStep[S]],
    feature_functions: Sequence[Callable[[NonTerminal[S]], float]],
    gamma: float,
```

```
    epsilon: float
) -> LinearFunctionApprox[NonTerminal[S]]:
    ''' transitions is a finite iterable '''
    num_features: int = len(feature_functions)
    a_inv: np.ndarray = np.eye(num_features) / epsilon
    b_vec: np.ndarray = np.zeros(num_features)
    for tr in transitions:
        phi1: np.ndarray = np.array([f(tr.state) for f in feature_functions])
        if isinstance(tr.next_state, NonTerminal):
            phi2 = phi1 - gamma * np.array([f(tr.next_state)
                                            for f in feature_functions])
        else:
            phi2 = phi1
        temp: np.ndarray = a_inv.T.dot(phi2)
        a_inv = a_inv - np.outer(a_inv.dot(phi1), temp) / (1 + phi1.dot(temp))
        b_vec += phi1 * tr.reward
    opt_wts: np.ndarray = a_inv.dot(b_vec)
    return LinearFunctionApprox.create(
        feature_functions=feature_functions,
        weights=Weights.create(opt_wts)
    )
```

The code above is in the file rl/td.py.

Now let's test this on transitions data sampled from the `RandomWalkMRP` example we had constructed in Chapter **??**. As a reminder, this MRP consists of a random walk across states $\{0, 1, 2, \ldots, B\}$ with 0 and B as the terminal states (think of these as terminating barriers of a random walk) and the remaining states as the non-terminal states. From any non-terminal state i, we transition to state $i + 1$ with probability p and to state $i - 1$ with probability $1 - p$. The reward is 0 upon each transition, except if we transition from state $B - 1$ to terminal state B which results in a reward of 1. The code for `RandomWalkMRP` is in the file rl/chapter10/random_walk_mrp.py.

First we set up a `RandomWalkMRP` object with $B = 20, p = 0.55$ and calculate it's true Value Function (so we can later compare against Incremental TD and LSTD methods).

```
from rl.chapter10.random_walk_mrp import RandomWalkMRP
import nump as np
this_barrier: int = 20
this_p: float = 0.55
random_walk: RandomWalkMRP = RandomWalkMRP(
    barrier=this_barrier,
    p=this_p
)
gamma = 1.0
true_vf: np.ndarray = random_walk.get_value_function_vec(gamma=gamma)
```

Let's say we have access to only 10,000 transitions (each transition is an object of the type `TransitionStep`). First we generate these 10,000 sampled transitions from the `RandomWalkMRP` object we created above.

```
from rl.approximate_dynamic_programming import NTStateDistribution
from rl.markov_process import TransitionStep
import itertools
num_transitions: int = 10000
nt_states: Sequence[NonTerminal[int]] = random_walk.non_terminal_states
start_distribution: NTStateDistribution[int] = Choose(set(nt_states))
traces: Iterable[Iterable[TransitionStep[int]]] = \
    random_walk.reward_traces(start_distribution)
transitions: Iterable[TransitionStep[int]] = \
    itertools.chain.from_iterable(traces)
```

```
td_transitions: Iterable[TransitionStep[int]] = \
    itertools.islice(transitions, num_transitions)
```

Before running LSTD, let's run Incremental Tabular TD on the 10,000 transitions in `td_transitions` and obtain the resultant Value Function (`td_vf` in the code below). Since there are only 10,000 transitions, we use an aggressive initial learning rate of 0.5 to promote fast learning, but we let this high learning rate decay quickly so the learning stabilizes.

```
from rl.function_approx import Tabular
import rl.iterate as iterate

initial_learning_rate: float = 0.5
half_life: float = 1000
exponent: float = 0.5
approx0: Tabular[NonTerminal[int]] = Tabular(
    count_to_weight_func=learning_rate_schedule(
        initial_learning_rate=initial_learning_rate,
        half_life=half_life,
        exponent=exponent
    )
)
td_func: Tabular[NonTerminal[int]] = \
    iterate.last(itertools.islice(
        td_prediction(
            transitions=td_transitions,
            approx_0=approx0,
            gamma=gamma
        ),
        num_transitions
    ))
td_vf: np.ndarray = td_func.evaluate(nt_states)
```

Finally, we run the LSTD algorithm on 10,000 transitions. Note that the Value Function of `RandomWalkMRP`, for $p \neq 0.5$, is non-linear as a function of the integer states. So we use non-linear features that can approximate arbitrary non-linear shapes - a good choice is the set of (orthogonal) Laguerre Polynomials. In the code below, we use the first 5 Laguerre Polynomials (i.e., upto degree 4 polynomial) as the feature functions for the linear function approximation of the Value Function. Then we invoke the LSTD algorithm we wrote above to calculate the `LinearFunctionApprox` based on this batch of 10,000 transitions.

```
from rl.chapter12.laguerre import laguerre_state_features
from rl.function_approx import LinearFunctionApprox

num_polynomials: int = 5
features: Sequence[Callable[[NonTerminal[int]], float]] = \
    laguerre_state_features(num_polynomials)
lstd_transitions: Iterable[TransitionStep[int]] = \
    itertools.islice(transitions, num_transitions)
epsilon: float = 1e-4

lstd_func: LinearFunctionApprox[NonTerminal[int]] = \
    least_squares_td(
        transitions=lstd_transitions,
        feature_functions=features,
        gamma=gamma,
        epsilon=epsilon
    )
lstd_vf: np.ndarray = lstd_func.evaluate(nt_states)
```

Figure 1.1 depicts how the LSTD Value Function estimate (for 10,000 transitions) `lstd_vf` compares against Incremental Tabular TD Value Function estimate (for 10,000 transitions)

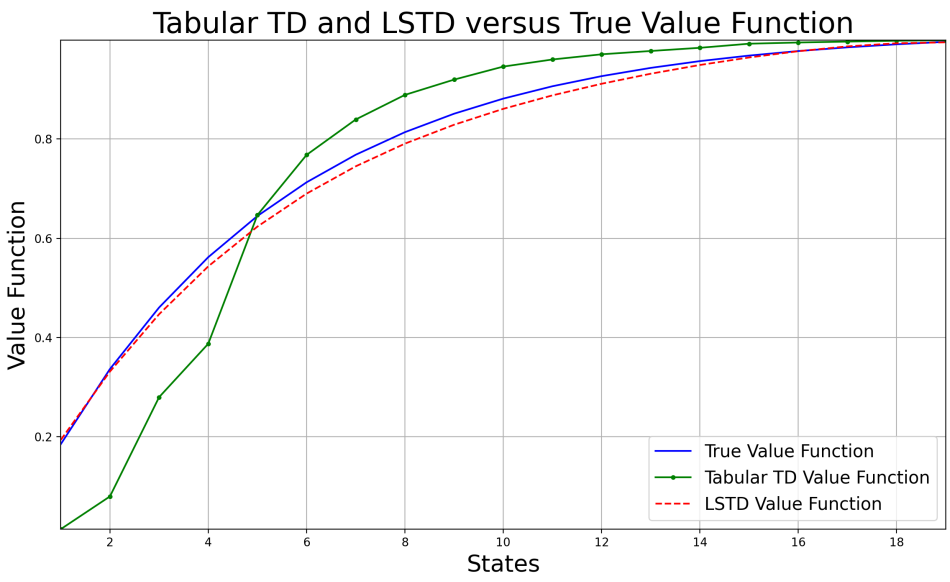

Figure 1.1: LSTD and Tabular TD Value Functions

`td_vf` and against the true value function `true_vf` (obtained using the linear-algebra-solver-based calculation of the MRP Value Function). We encourage you to modify the parameters used in the code above to see how it alters the results - specifically play around with `this_barrier`, `this_p`, `gamma`, `num_transitions`, the learning rate trajectory for Incremental Tabular TD, the number of Laguerre polynomials, and `epsilon`. The above code is in the file rl/chapter12/random_walk_lstd.py.

1.3.3 LSTD(λ)

Likewise, we can do LSTD(λ) using Eligibility Traces. Here we are given a fixed, finite number of trace experiences

$$\mathcal{D} = [(S_{i,0}, R_{i,1}, S_{i,1}, R_{i,2}, S_{i,2}, \ldots, R_{i,T_i}, S_{i,T_i})|1 \leq i \leq n]$$

Denote the Eligibility Traces of trace experience i at time t as $\boldsymbol{E}_{i,t}$. Note that the eligibility traces accumulate $\nabla_{\boldsymbol{w}} V(s; \boldsymbol{w}) = \boldsymbol{\phi}(s)$ in each trace experience. When accumulating, the previous time step's eligibility traces is discounted by $\lambda\gamma$. By setting the right-hand-side of Equation (1.1) to 0 (i.e., setting the update to $\boldsymbol{w}$ over all atomic experiences data to 0), we get:

$$\sum_{i=1}^{n} \frac{1}{T_i} \cdot \sum_{t=0}^{T_i-1} \boldsymbol{E}_{i,t} \cdot (\boldsymbol{\phi}(S_{i,t})^T \cdot \boldsymbol{w^*} - (R_{i,t+1} + \gamma \cdot \boldsymbol{\phi}(S_{i,t+1})^T \cdot \boldsymbol{w}^*)) = 0$$

We can calculate the solution $\boldsymbol{w}^*$ as $\boldsymbol{A}^{-1} \cdot \boldsymbol{b}$, where the $m \times m$ Matrix $\boldsymbol{A}$ is accumulated at each atomic experience $(S_{i,t}, R_{i,t+1}, S_{i,t+1})$ as:

$$\boldsymbol{A} \leftarrow \boldsymbol{A} + \frac{1}{T_i} \cdot \boldsymbol{E}_{i,t} \cdot (\boldsymbol{\phi}(S_{i,t}) - \gamma \cdot \boldsymbol{\phi}(S_{i,t+1}))^T \text{ (note the Outer-Product)}$$

On/Off Policy	Algorithm	Tabular	Linear	Non-Linear
On-Policy	MC	✓	✓	✓
	LSMC	✓	✓	-
	TD	✓	✓	✗
	LSTD	✓	✓	-
	Gradient TD	✓	✓	✓
Off-Policy	MC	✓	✓	✓
	LSMC	✓	✗	-
	TD	✓	✗	✗
	LSTD	✓	✗	-
	Gradient TD	✓	✓	✓

Figure 1.2: Convergence of RL Prediction Algorithms

and the m-Vector $\boldsymbol{b}$ is accumulated at each atomic experience $(S_{i,t}, R_{i,t+1}, S_{i,t+1})$ as:

$$\boldsymbol{b} \leftarrow \boldsymbol{b} + \frac{1}{T_i} \cdot \boldsymbol{E}_{i,t} \cdot R_{i,t+1}$$

With Sherman-Morrison incremental inverse, we can reduce the computational complexity from $O(m^3)$ to $O(m^2)$.

1.3.4 Convergence of Least-Squares Prediction

Before we move on to Least-Squares for the Control problem, we want to point out that the convergence behavior of Least-Squares Prediction algorithms are identical to their counterpart Incremental RL Prediction algorithms, with the exception that Off-Policy LSMC does not have convergence guarantees. Figure 1.2 shows the updated summary table for convergence of RL Prediction algorithms (that we had displayed at the end of Chapter **??**) to now also include Least-Squares Prediction algorithms.

This ends our coverage of Least-Squares Prediction. Before we move on to Least-Squares Control, we need to cover Incremental RL Control with Experience-Replay as it serves as a stepping stone towards Least-Squares Control.

1.4 Q-Learning with Experience-Replay

In this section, we cover Off-Policy Incremental TD Control with Experience-Replay. Specifically, we revisit the Q-Learning algorithm we covered in Chapter **??**, but we tweak that algorithm such that the transitions used to make the Q-Learning updates are sourced from an experience-replay memory, rather than from a behavior policy derived from the current Q-Value estimate. While investigating the challenges with Off-Policy TD methods with deep learning function approximation, researchers identified two challenges:

1) The sequences of states made available to deep learning through trace experiences are highly correlated, whereas deep learning algorithms are premised on data samples being independent.
2) The data distribution changes as the RL algorithm learns new behaviors, whereas deep learning algorithms are premised on a fixed underlying distribution (i.e., stationary).

Experience-Replay serves to smooth the training data distribution over many past behaviors, effectively resolving the correlation issue as well as the non-stationary issue. Hence, Experience-Replay is a powerful idea for Off-Policy TD Control. The idea of using Experience-Replay for Off-Policy TD Control is due to the Ph.D. thesis of Long Lin (Lin 1993).

To make this idea of Q-Learning with Experience-Replay clear, we make a few changes to the `q_learning` function we had written in Chapter **??** with the following function `q_learning_experience_replay`

```
from rl.markov_decision_process import TransitionStep
from rl.approximate_dynamic_programming import QValueFunctionApprox
from rl.approximate_dynamic_programming import NTStateDistribution
from rl.experience_replay import ExperienceReplayMemory

PolicyFromQType = Callable[
    [QValueFunctionApprox[S, A], MarkovDecisionProcess[S, A]],
    Policy[S, A]
]

def q_learning_experience_replay(
    mdp: MarkovDecisionProcess[S, A],
    policy_from_q: PolicyFromQType,
    states: NTStateDistribution[S],
    approx_0: QValueFunctionApprox[S, A],
    gamma: float,
    max_episode_length: int,
    mini_batch_size: int,
    weights_decay_half_life: float
) -> Iterator[QValueFunctionApprox[S, A]]:
    exp_replay: ExperienceReplayMemory[TransitionStep[S, A]] = \
        ExperienceReplayMemory(
            time_weights_func=lambda t: 0.5 ** (t / weights_decay_half_life),
        )
    q: QValueFunctionApprox[S, A] = approx_0
    yield q
    while True:
        state: NonTerminal[S] = states.sample()
        steps: int = 0
        while isinstance(state, NonTerminal) and steps < max_episode_length:
            policy: Policy[S, A] = policy_from_q(q, mdp)
            action: A = policy.act(state).sample()
            next_state, reward = mdp.step(state, action).sample()
            exp_replay.add_data(TransitionStep(
                state=state,
                action=action,
                next_state=next_state,
                reward=reward
            ))
            trs: Sequence[TransitionStep[S, A]] = \
                exp_replay.sample_mini_batch(mini_batch_size)
            q = q.update(
                [(
                    (tr.state, tr.action),
                    tr.reward + gamma * (
                        max(q((tr.next_state, a))
                            for a in mdp.actions(tr.next_state))
                        if isinstance(tr.next_state, NonTerminal) else 0.)
                ) for tr in trs],
            )
            yield q
            steps += 1
            state = next_state
```

The key difference between the `q_learning` algorithm we wrote in Chapter **??** and this `q_learning_experience_replay` algorithm is that here we have an experience-replay memory (using the `ExperienceReplayMemory` class we had implemented earlier). In the `q_learning`

algorithm, the (`state`, `action`, `next_state`, `reward`) 4-tuple comprising `TransitionStep` (that is used to perform the Q-Learning update) was the result of `action` being sampled from the behavior policy (derived from the current estimate of the Q-Value Function, eg: ϵ-greedy), and then the `next_state` and `reward` being generated from the (`state`, `action`) pair using the `step` method of `mdp`. Here in `q_learning_experience_replay`, we don't use this 4-tuple `TransitionStep` to perform the update - rather, we append this 4-tuple to the `ExperienceReplayMemory` (using the `add_data` method), then we sample `mini_batch_sized` `TransitionSteps` from the `ExperienceReplayMemory` (giving more sampling weightage to the more recently added `TransitionSteps`), and use those 4-tuple `TransitionSteps` to perform the Q-Learning update. Note that these sampled `TransitionSteps` might be from old behavior policies (derived from old estimates of the Q-Value estimate). The key is that this algorithm re-uses atomic experiences that were previously prepared by the algorithm, which also means that it re-uses behavior policies that were previously constructed by the algorithm.

The argument `mini_batch_size` refers to the number of `TransitionSteps` to be drawn from the `ExperienceReplayMemory` at each step. The argument `weights_decay_half_life` refers to the half life of an exponential decay function for the weights used in the sampling of the `TransitionSteps` (the most recently added `TransitionStep` has the highest weight). With this understanding, the code should be self-explanatory.

The above code is in the file rl/td.py.

1.4.1 Deep Q-Networks (DQN) Algorithm

DeepMind developed an innovative and practically effective RL Control algorithm based on Q-Learning with Experience-Replay - an algorithm they named as Deep Q-Networks (abberviated as DQN). Apart from reaping the above-mentioned benefits of Experience-Replay for Q-Learning with a Deep Neural Network approximating the Q-Value function, they also benefited from employing a second Deep Neural Network (let us call the main DNN as the Q-Network, refering to it's parameters at $\boldsymbol{w}$, and the second DNN as the target network, refering to it's parameters as $\boldsymbol{w}^-$). The parameters $\boldsymbol{w}^-$ of the target network are infrequently updated to be made equal to the parameters $\boldsymbol{w}$ of the Q-network. The purpose of the Q-Network is to evaluate the Q-Value of the current state s and the purpose of the target network is to evaluate the Q-Value of the next state s', which in turn is used to obtain the Q-Learning target (note that the Q-Value of the current state is $Q(s, a; \boldsymbol{w})$ and the Q-Learning target is $r + \gamma \cdot \max_{a'} Q(s', a'; \boldsymbol{w}^-)$ for a given atomic experience (s, a, r, s')).

Deep Learning is premised on the fact that the supervised learning targets (response values y corresponding to predictor values x) are pre-generated fixed values. This is not the case in TD learning where the targets are dependent on the Q-Values. As Q-Values are updated at each step, the targets also get updated, and this correlation between the current state's Q-Value estimate and the target value typically leads to oscillations or divergence of the Q-Value estimate. By infrequently updating the parameters $\boldsymbol{w}^-$ of the target network (providing the target values) to be made equal to the parameters $\boldsymbol{w}$ of the Q-network (which are updated at each iteration), the targets in the Q-Learning update are essentially kept fixed. This goes a long way in resolving the core issue of correlation between the current state's Q-Value estimate and the target values, helping considerably with convergence of the Q-Learning algorithm. Thus, DQN reaps the benefits of not just Experience-Replay in Q-Learning (which we articulated earlier), but also the benefits of having "fixed" targets. DNN utilizes a parameter C such that the updating of $\boldsymbol{w}^-$ to be made equal to $\boldsymbol{w}$ is done once every C updates to $\boldsymbol{w}$ (updates to $\boldsymbol{w}$ are based on the usual

Q-Learning update equation).

We won't implement the DQN algorithm in Python code - however, we sketch the outline of the algorithm, as follows:

At each time t for each episode:

- Given state S_t, take action A_t according to ϵ-greedy policy extracted from Q-network values $Q(S_t, a; \boldsymbol{w})$.
- Given state S_t and action A_t, obtain reward R_{t+1} and next state S_{t+1} from the environment.
- Append atomic experience $(S_t, A_t, R_{t+1}, S_{t+1})$ in experience-replay memory $\mathcal{D}$.
- Sample a random mini-batch of atomic experiences $(s_i, a_i, r_i, s_i') \sim \mathcal{D}$.
- Using this mini-batch of atomic experiences, update the Q-network parameters $\boldsymbol{w}$ with the Q-learning targets based on "frozen" parameters $\boldsymbol{w}^-$ of the target network.

$$\Delta \boldsymbol{w} = \alpha \cdot \sum_i (r_i + \gamma \cdot \max_{a_i'} Q(s_i', a_i'; \boldsymbol{w}^-) - Q(s_i, a_i; \boldsymbol{w})) \cdot \nabla_{\boldsymbol{w}} Q(s_i, a_i; \boldsymbol{w})$$

- $S_t \leftarrow S_{t+1}$
- Once every C time steps, set $\boldsymbol{w}^- \leftarrow \boldsymbol{w}$.

To learn more about the effectiveness of DQN for Atari games, see the Original DQN Paper (Mnih et al. 2013) and the DQN Nature Paper (Mnih et al. 2015) that DeepMind has published.

Now we are ready to cover Batch RL Control (specifically Least-Squares TD Control), which combines the ideas of Least-Squares TD Prediction and Q-Learning with Experience-Replay.

1.5 Least-Squares Policy Iteration (LSPI)

Having seen Least-Squares Prediction, the natural question is whether we can extend the Least-Squares (batch with linear function approximation) methodology to solve the Control problem. For On-Policy MC Control and On-Policy TD Control, we take the usual route of Generalized Policy Iteration (GPI) with:

1. Policy Evaluation as Least-Squares Q-Value Prediction. Specifically, the Q-Value for a policy π is approximated as:

   $$Q^\pi(s, a) \approx Q(s, a; \boldsymbol{w}) = \boldsymbol{\phi}(s, a)^T \cdot \boldsymbol{w} \text{ for all } s \in \mathcal{N}, \text{ for all } a \in \mathcal{A}$$

   with a direct linear-algebraic solve for the linear function approximation weights $\boldsymbol{w}$ using batch experiences data generated using policy π.
2. ϵ-Greedy Policy Improvement.

In this section, we focus on Off-Policy Control with Least-Squares TD. This algorithm is known as Least-Squares Policy Iteration, abbreviated as LSPI, developed by Lagoudakis and Parr (Lagoudakis and Parr 2003). LSPI has been an important go-to algorithm in the history of RL Control because of it's simplicity and effectiveness. The basic idea of LSPI is that it does Generalized Policy Iteration (GPI) in the form of *Q-Learning with Experience-Replay*, with the key being that instead of doing the usual Q-Learning update after each atomic experience, we do *batch Q-Learning* for the Policy Evaluation phase of GPI. We

spend the rest of this section describing LSPI in detail and then implementing it in Python code.

The input to LSPI is a fixed finite data set $\mathcal{D}$, consisting of a set of (s, a, r, s') atomic experiences, i.e., a set of `rl.markov_decision_process.TransitionStep` objects, and the task of LSPI is to determine the Optimal Q-Value Function (and hence, Optimal Policy) based on this experiences data set $\mathcal{D}$ using an experience-replayed, batch Q-Learning technique described below. Assume $\mathcal{D}$ consists of n atomic experiences, indexed as $i = 1, 2, \ldots n$, with atomic experience i denoted as (s_i, a_i, r_i, s_i').

In LSPI, each iteration of GPI involves access to:

- The experiences data set $\mathcal{D}$.
- A *Deterministic Target Policy* (call it π_D), that is made available from the previous iteration of GPI.

Given $\mathcal{D}$ and π_D, the goal of each iteration of GPI is to solve for weights $\boldsymbol{w}^*$ that minimizes:

$$\begin{aligned}\mathcal{L}(\boldsymbol{w}) &= \sum_{i=1}^{n}(Q(s_i, a_i; \boldsymbol{w}) - (r_i + \gamma \cdot Q(s_i', \pi_D(s_i'); \boldsymbol{w})))^2 \\ &= \sum_{i=1}^{n}(\boldsymbol{\phi}(s_i, a_i)^T \cdot \boldsymbol{w} - (r_i + \gamma \cdot \boldsymbol{\phi}(s_i', \pi_D(s_i'))^T \cdot \boldsymbol{w}))^2\end{aligned}$$

The solution for the weights $\boldsymbol{w}^*$ is attained by setting the semi-gradient of $\mathcal{L}(\boldsymbol{w})$ to 0, i.e.,

$$\sum_{i=1}^{n} \phi(s_i, a_i) \cdot (\boldsymbol{\phi}(s_i, a_i)^T \cdot \boldsymbol{w}^* - (r_i + \gamma \cdot \boldsymbol{\phi}(s_i', \pi_D(s_i'))^T \cdot \boldsymbol{w}^*)) = 0 \tag{1.2}$$

We can calculate the solution $\boldsymbol{w}^*$ as $\boldsymbol{A}^{-1} \cdot \boldsymbol{b}$, where the $m \times m$ Matrix $\boldsymbol{A}$ is accumulated for each `TransitionStep` (s_i, a_i, r_i, s_i') as:

$$\boldsymbol{A} \leftarrow \boldsymbol{A} + \boldsymbol{\phi}(s_i, a_i) \cdot (\boldsymbol{\phi}(s_i, a_i) - \gamma \cdot \boldsymbol{\phi}(s_i', \pi_D(s_i')))^T$$

and the m-Vector $\boldsymbol{b}$ is accumulated at each atomic experience (s_i, a_i, r_i, s_i') as:

$$\boldsymbol{b} \leftarrow \boldsymbol{b} + \boldsymbol{\phi}(s_i, a_i) \cdot r_i$$

With Sherman-Morrison incremental inverse, we can reduce the computational complexity from $O(m^3)$ to $O(m^2)$.

This solved $\boldsymbol{w}^*$ defines an updated Q-Value Function as follows:

$$Q(s, a; \boldsymbol{w}^*) = \boldsymbol{\phi}(s, a)^T \cdot \boldsymbol{w}^* = \sum_{j=1}^{m} \phi_j(s, a) \cdot w_j^*$$

This defines an updated, improved deterministic policy π_D' (serving as the *Deterministic Target Policy* for the next iteration of GPI):

$$\pi_D'(s) = \arg\max_a Q(s, a; \boldsymbol{w}^*) \text{ for all } s \in \mathcal{N}$$

This least-squares solution of $\boldsymbol{w}^*$ (Prediction) is known as Least-Squares Temporal Difference for Q-Value, abbreviated as *LSTDQ*. Thus, LSPI is GPI with LSTDQ and greedy

policy improvements. Note how LSTDQ in each iteration re-uses the same data $\mathcal{D}$, i.e., LSPI does experience-replay.

We should point out here that the LSPI algorithm we described above should be considered as the *standard variant* of LSPI. However, we can design several other variants of LSPI, in terms of how the experiences data is sourced and used. Firstly, we should note that the experiences data $\mathcal{D}$ essentially provides the behavior policy for Q-Learning (along with the consequent reward and next state transition). In the *standard variant* we described above, since $\mathcal{D}$ is provided from an external source, the behavior policy that generates this data $\mathcal{D}$ must come from an external source. It doesn't have to be this way - we could generate the experiences data from a behavior policy derived from the Q-Value estimates produced by LSTDQ (eg: ϵ-greedy policy). This would mean the experiences data used in the algorithm is not a fixed, finite data set, rather a variable, incrementally-produced data set. Even if the behavior policy was external, the data set $\mathcal{D}$ might not be a fixed finite data set - rather, it could be made available as an on-demand, variable data stream. Furthermore, in each iteration of GPI, we could use a subset of the experiences data made available until that point of time (rather than the approach of the standard variant of LSPI that uses all of the available experiences data). If we choose to sample a subset of the available experiences data, we might give more sampling-weightage to the more recently generated data. This would especially be the case if the experiences data was being generated from a policy derived from the Q-Value estimates produced by LSTDQ. In this case, we would leverage the `ExperienceReplayMemory` class we'd written earlier.

Next, we write code to implement the *standard variant* of LSPI we described above. First, we write a function to implement LSTDQ. As described above, the inputs to LSTDQ are the experiences data $\mathcal{D}$ (`transitions` in the code below) and a deterministic target policy π_D (`target_policy` in the code below). Since we are doing a linear function approximation, the input also includes a set of features, described as functions of state and action (`feature_functions` in the code below). Lastly, the inputs also include the discount factor γ and the numerical control parameter ϵ. The code below should be fairly self-explanatory, as it is a straightforward extension of LSTD (implemented in function `least_squares_td` earlier). The key differences are that this is an estimate of the Action-Value (Q-Value) function, rather than the State-Value Function, and the target used in the least-squares calculation is the Q-Learning target (produced by the `target_policy`).

```
def least_squares_tdq(
    transitions: Iterable[TransitionStep[S, A]],
    feature_functions: Sequence[Callable[[Tuple[NonTerminal[S], A]], float]],
    target_policy: DeterministicPolicy[S, A],
    gamma: float,
    epsilon: float
) -> LinearFunctionApprox[Tuple[NonTerminal[S], A]]:
    '''transitions is a finite iterable'''
    num_features: int = len(feature_functions)
    a_inv: np.ndarray = np.eye(num_features) / epsilon
    b_vec: np.ndarray = np.zeros(num_features)
    for tr in transitions:
        phi1: np.ndarray = np.array([f((tr.state, tr.action))
                                     for f in feature_functions])
        if isinstance(tr.next_state, NonTerminal):
            phi2 = phi1 - gamma * np.array([
                f((tr.next_state, target_policy.action_for(tr.next_state.state)))
                for f in feature_functions])
        else:
            phi2 = phi1
        temp: np.ndarray = a_inv.T.dot(phi2)
        a_inv = a_inv - np.outer(a_inv.dot(phi1), temp) / (1 + phi1.dot(temp))
```

```
            b_vec += phi1 * tr.reward
        opt_wts: np.ndarray = a_inv.dot(b_vec)
        return LinearFunctionApprox.create(
            feature_functions=feature_functions,
            weights=Weights.create(opt_wts)
        )
```

Now we are ready to write the standard variant of LSPI. The code below is a straightforward implementation of our description above, looping through the iterations of GPI, `yield`ing the Q-Value `LinearFunctionApprox` after each iteration of GPI.

```
def least_squares_policy_iteration(
    transitions: Iterable[TransitionStep[S, A]],
    actions: Callable[[NonTerminal[S]], Iterable[A]],
    feature_functions: Sequence[Callable[[Tuple[NonTerminal[S], A]], float]],
    initial_target_policy: DeterministicPolicy[S, A],
    gamma: float,
    epsilon: float
) -> Iterator[LinearFunctionApprox[Tuple[NonTerminal[S], A]]]:
    '''transitions is a finite iterable'''
    target_policy: DeterministicPolicy[S, A] = initial_target_policy
    transitions_seq: Sequence[TransitionStep[S, A]] = list(transitions)
    while True:
        q: LinearFunctionApprox[Tuple[NonTerminal[S], A]] = \
            least_squares_tdq(
                transitions=transitions_seq,
                feature_functions=feature_functions,
                target_policy=target_policy,
                gamma=gamma,
                epsilon=epsilon,
            )
        target_policy = greedy_policy_from_qvf(q, actions)
        yield q
```

The above code is in the file rl/td.py.

1.5.1 Saving your Village from a Vampire

Now we consider a Control problem we'd like to test the above LSPI algorithm on. We call it the Vampire problem that can be described as a good old-fashioned bedtime story, as follows:

> *A village is visited by a vampire every morning who uniform-randomly eats 1 villager upon entering the village, then retreats to the hills, planning to come back the next morning. The villagers come up with a plan. They will poison a certain number of villagers each night until the vampire eats a poisoned villager the next morning, after which the vampire dies immediately (due to the poison in the villager the vampire ate). Unfortunately, all villagers who get poisoned also die the day after they are given the poison. If the goal of the villagers is to maximize the expected number of villagers at termination (termination is when either the vampire dies or all villagers die), what should be the optimal poisoning strategy? In other words, if there are n villagers on any day, how many villagers should be poisoned (as a function of n)?*

It is straightforward to model this problem as an MDP. The *State* is the number of villagers at risk on any given night (if the vampire is still alive, the *State* is the number of villagers and if the vampire is dead, the *State* is 0, which is the only *Terminal State*). The

Action is the number of villagers poisoned on any given night. The *Reward* is zero as long as the vampire is alive, and is equal to the number of villagers remaining if the vampire dies. Let us refer to the initial number of villagers as I. Thus,

$$\mathcal{S} = \{0, 1, \ldots, I\}, \mathcal{T} = \{0\}$$

$$\mathcal{A}(s) = \{0, 1, \ldots, s-1\} \text{ where } s \in \mathcal{N}$$

For all $s \in \mathcal{N}$, for all $a \in \mathcal{A}(s)$,

$$\mathcal{P}_R(s, a, r, s') = \begin{cases} \frac{s-a}{s} & \text{if } r = 0 \text{ and } s' = s - a - 1 \\ \frac{a}{s} & \text{if } r = s - a \text{ and } s' = 0 \\ 0 & \text{otherwise} \end{cases}$$

It is rather straightforward to solve this with Dynamic Programming (say, Value Iteration) since we know the transition probabilities and rewards function and since the state and action spaces are finite. However, in a situation where we don't know the exact probabilities with which the vampire operates, and we only had access to observations on specific days, we can attempt to solve this problem with Reinforcement Learning (assuming we had access to observations of many vampires operating on many villages). In any case, our goal here is to test LSPI using this vampire problem as an example. So we write some code to first model this MDP as described above, solve it with value iteration (to obtain the benchmark, i.e., true Optimal Value Function and true Optimal Policy to compare against), then generate atomic experiences data from the MDP, and then solve this problem with LSPI using this stream of generated atomic experiences.

```
from rl.markov_decision_process import TransitionStep
from rl.distribution import Categorical, Choose
from rl.function_approx import LinearFunctionApprox
from rl.policy import DeterministicPolicy, FiniteDeterministicPolicy
from rl.dynamic_programming import value_iteration_result, V
from rl.chapter11.control_utils import get_vf_and_policy_from_qvf
from rl.td import least_squares_policy_iteration
from numpy.polynomial.laguerre import lagval
import itertools
import rl.iterate as iterate
import numpy as np

class VampireMDP(FiniteMarkovDecisionProcess[int, int]):

    initial_villagers: int

    def __init__(self, initial_villagers: int):
        self.initial_villagers = initial_villagers
        super().__init__(self.mdp_map())

    def mdp_map(self) -> \
            Mapping[int, Mapping[int, Categorical[Tuple[int, float]]]]:
        return {s: {a: Categorical(
            {(s - a - 1, 0.): 1 - a / s, (0, float(s - a)): a / s}
        ) for a in range(s)} for s in range(1, self.initial_villagers + 1)}

    def vi_vf_and_policy(self) -> \
            Tuple[V[int], FiniteDeterministicPolicy[int, int]]:
        return value_iteration_result(self, 1.0)

    def lspi_features(
        self,
        factor1_features: int,
        factor2_features: int
    ) -> Sequence[Callable[[Tuple[NonTerminal[int], int]], float]]:
```

```
        ret: List[Callable[[Tuple[NonTerminal[int], int]], float]] = []
        ident1: np.ndarray = np.eye(factor1_features)
        ident2: np.ndarray = np.eye(factor2_features)
        for i in range(factor1_features):
            def factor1_ff(x: Tuple[NonTerminal[int], int], i=i) -> float:
                return lagval(
                    float((x[0].state - x[1]) ** 2 / x[0].state),
                    ident1[i]
                )
            ret.append(factor1_ff)
        for j in range(factor2_features):
            def factor2_ff(x: Tuple[NonTerminal[int], int], j=j) -> float:
                return lagval(
                    float((x[0].state - x[1]) * x[1] / x[0].state),
                    ident2[j]
                )
            ret.append(factor2_ff)
        return ret

    def lspi_transitions(self) -> Iterator[TransitionStep[int, int]]:
        states_distribution: Choose[NonTerminal[int]] = \
            Choose(self.non_terminal_states)
        while True:
            state: NonTerminal[int] = states_distribution.sample()
            action: int = Choose(range(state.state)). sample()
            next_state, reward = self.step(state, action).sample()
            transition: TransitionStep[int, int] = TransitionStep(
                state=state,
                action=action,
                next_state=next_state,
                reward=reward
            )
            yield transition

    def lspi_vf_and_policy(self) -> \
            Tuple[V[int], FiniteDeterministicPolicy[int, int]]:
        transitions: Iterable[TransitionStep[int, int]] = itertools.islice(
            self.lspi_transitions(),
            20000
        )
        qvf_iter: Iterator[LinearFunctionApprox[Tuple[
            NonTerminal[int], int]]] = least_squares_policy_iteration(
                transitions=transitions,
                actions=self.actions,
                feature_functions=self.lspi_features(4, 4),
                initial_target_policy=DeterministicPolicy(
                    lambda s: int(s / 2)
                ),
                gamma=1.0,
                epsilon=1e-5
            )
        qvf: LinearFunctionApprox[Tuple[NonTerminal[int], int]] = \
            iterate.last(
                itertools.islice(
                    qvf_iter,
                    20
                )
            )
        return get_vf_and_policy_from_qvf(self, qvf)
```

The above code should be self-explanatory. The main challenge with LSPI is that we need to construct features function of the state and action such that the Q-Value Function is linear in those features. In this case, since we simply want to test the correctness of our LSPI implementation, we define feature functions (in method `lspi_feature` above) based on our knowledge of the true optimal Q-Value Function from the Dynamic Programming

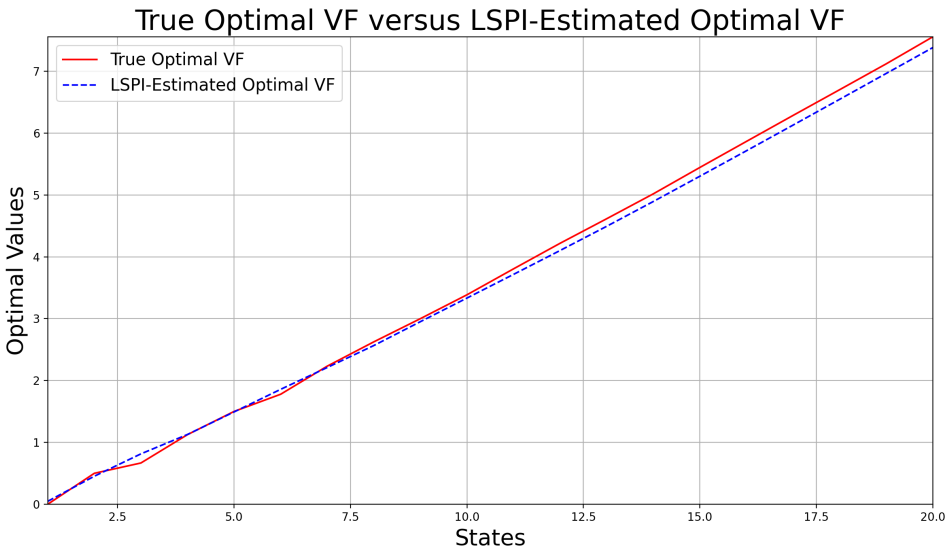

Figure 1.3: True versus LSPI Optimal Value Function

solution. The atomic experiences comprising the experiences data $\mathcal{D}$ for LSPI to use is generated with a uniform distribution of non-terminal states and a uniform distribution of actions for a given state (in method `lspi_transitions` above).

Figure 1.3 shows the plot of the True Optimal Value Function (from Value Iteration) versus the LSPI-estimated Optimal Value Function.

Figure 1.4 shows the plot of the True Optimal Policy (from Value Iteration) versus the LSPI-estimated Optimal Policy.

The above code is in the file rl/chapter12/vampire.py. As ever, we encourage you to modify some of the parameters in this code (including choices of feature functions, nature and number of atomic transitions used, number of GPI iterations, choice of ϵ, and perhaps even a different dynamic for the vampire behavior), and see how the results change.

1.5.2 Least-Squares Control Convergence

We wrap up this section by including the convergence behavior of LSPI in the summary table for convergence of RL Control algorithms (that we had displayed at the end of Chapter **??**). Figure 1.5 shows the updated summary table for convergence of RL Control algorithms to now also include LSPI. Note that (✓) means it doesn't quite hit the Optimal Value Function, but bounces around near the Optimal Value Function. But this is better than Q-Learning in the case of linear function approximation.

1.6 RL for Optimal Exercise of American Options

We learnt in Chapter **??** that the American Options Pricing problem is an Optimal Stopping problem and can be modeled as an MDP so that solving the Control problem of the MDP gives us the fair price of an American Option. We can solve it with Dynamic Programming or Reinforcement Learning, as appropriate.

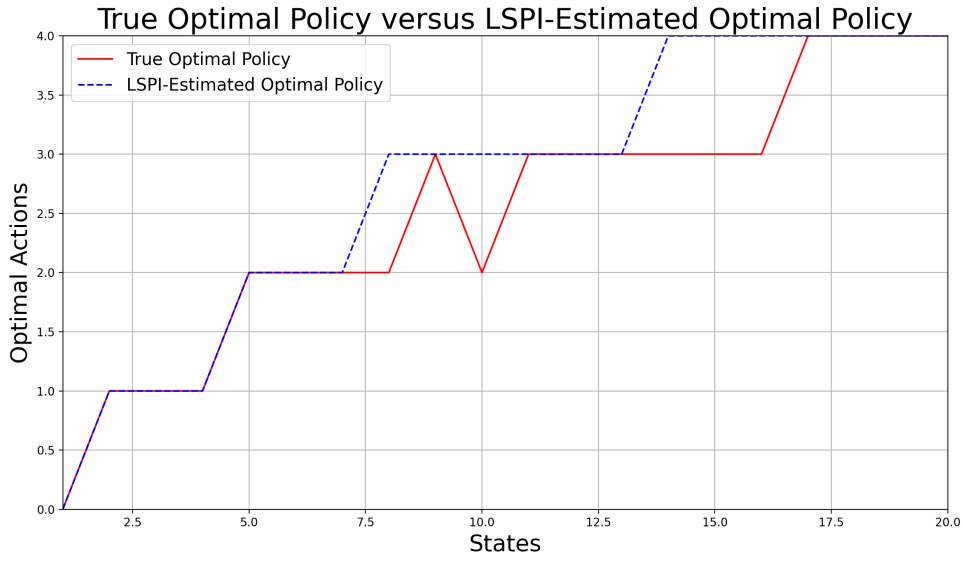

Figure 1.4: True versus LSPI Optimal Policy

Algorithm	Tabular	Linear	Non-Linear
MC Control	✓	(✓)	✗
SARSA	✓	(✓)	✗
Q-Learning	✓	✗	✗
LSPI	✓	(✓)	-
Gradient Q-Learning	✓	✓	✗

Figure 1.5: Convergence of RL Control Algorithms

In the financial trading industry, it has traditionally not been a common practice to explicitly view the American Options Pricing problem as an MDP. Specialized algorithms have been developed to price American Options. We now provide a quick overview of the common practice in pricing American Options in the financial trading industry. Firstly, we should note that the price of some American Options is equal to the price of the corresponding European Option, for which we have a closed-form solution under the assumption of a lognormal process for the underlying - this is the case for a plain-vanilla American call option whose price (as we proved in Chapter **??**) is equal to the price of a plain-vanilla European call option. However, this is not the case for a plain-vanilla American put option. Secondly, we should note that if the payoff of an American option is dependent on only the current price of the underlying (and not on the past prices of the underlying) - in which case, we say that the option payoff is not "history-dependent" - and if the dimension of the state space is not large, then we can do a simple backward induction on a binomial tree (as we showed in Chapter **??**). In practice, a more detailed data structure such as a trinomial tree or a lattice is often used for more accurate backward-induction calculations. However, if the payoff is history-dependent (i.e., payoff depends on past prices of the underlying) or if the payoff depends on the prices of several underlying assets, then the state space is too large for backward induction to handle. In such cases, the standard approach in the financial trading industry is to use the Longstaff-Schwartz pricing algorithm (Longstaff and Schwartz 2001). We won't cover the Longstaff-Schwartz pricing algorithm in detail in this book - it suffices to share here that the Longstaff-Schwartz pricing algorithm combines 3 ideas:

- The Pricing is based on a set of sampling traces of the underlying prices.
- Function approximation of the continuation value for in-the-money states
- Backward-recursive determination of early exercise states

The goal of this section is to explain how to price American Options with Reinforcement Learning, as an alternative to the Longstaff-Schwartz algorithm.

1.6.1 LSPI for American Options Pricing

A paper by Li, Szepesvari, Schuurmans (Li, Szepesvári, and Schuurmans 2009) showed that LSPI can be an attractive alternative to the Longstaff-Schwartz algorithm in pricing American Options. Before we dive into the details of pricing American Options with LSPI, let's review the MDP model for American Options Pricing.

- *State* is [Current Time, Relevant History of Underlying Security Prices].
- *Action* is Boolean: Exercise (i.e., Stop) or Continue.
- *Reward* is always 0, except upon Exercise (when the *Reward* is equal to the Payoff).
- *State*-transitions are based on the Underlying Securities' Risk-Neutral Process.

The key is to create a linear function approximation of the state-conditioned *continuation value* of the American Option (*continuation value* is the price of the American Option at the current state, conditional on not exercising the option at the current state, i.e., continuing to hold the option). Knowing the continuation value in any state enables us to compare the continuation value against the exercise value (i.e., payoff), thus providing us with the Optimal Stopping criteria (as a function of the state), which in turn enables us to determine the Price of the American Option. Furthermore, we can customize the LSPI algorithm to the nuances of the American Option Pricing problem, yielding a specialized version of

LSPI. The key customization comes from the fact that there are only two actions. The action to exercise produces a (state-conditioned) reward (i.e., option payoff) and transition to a terminal state. The action to continue produces no reward and transitions to a new state at the next time step. Let us refer to these 2 actions as: $a = c$ (continue the option) and $a = e$ (exercise the option).

Since we know the exercise value in any state, we only need to create a linear function approximation for the continuation value, i.e., for the Q-Value $Q(s, c)$ for all non-terminal states s. If we denote the payoff in non-terminal state s as $g(s)$, then $Q(s, e) = g(s)$. So we write

$$\hat{Q}(s, a; \boldsymbol{w}) = \begin{cases} \boldsymbol{\phi}(s)^T \cdot \boldsymbol{w} & \text{if } a = c \\ g(s) & \text{if } a = e \end{cases} \text{ for all } s \in \mathcal{N}$$

for feature functions $\boldsymbol{\phi}(\cdot) = [\phi_i(\cdot)|i = 1, \ldots, m]$, which are feature functions of only state (and not action).

Each iteration of GPI in the LSPI algorithm starts with a deterministic target policy $\pi_D(\cdot)$ that is made available as a greedy policy derived from the previous iteration's LSTDQ-solved $\hat{Q}(s; a; \boldsymbol{w}^*)$. The LSTDQ solution for $\boldsymbol{w}^*$ is based on pre-generated training data and with the Q-Learning target policy set to be π_D. Since we learn the Q-Value function for only $a = c$, the behavior policy μ generating experiences data for training is a constant function $\mu(s) = c$. Note also that for American Options, the reward for $a = c$ is 0. So each atomic experience for training is of the form $(s, c, 0, s')$. This means we can represent each atomic experience for training as a 2-tuple (s, s'). This reduces the LSPI Semi-Gradient Equation (1.2) to:

$$\sum_i \boldsymbol{\phi}(s_i) \cdot (\boldsymbol{\phi}(s_i)^T \cdot \boldsymbol{w}^* - \gamma \cdot \hat{Q}(s_i', \pi_D(s_i'); \boldsymbol{w}^*)) = 0 \tag{1.3}$$

We need to consider two cases for the term $\hat{Q}(s_i', \pi_D(s_i'); \boldsymbol{w}^*)$:

- $C1$: If s_i' is non-terminal and $\pi_D(s_i') = c$ (i.e., $\boldsymbol{\phi}(s_i')^T \cdot \boldsymbol{w} \geq g(s_i')$): Substitute $\boldsymbol{\phi}(s_i')^T \cdot \boldsymbol{w}^*$ for $\hat{Q}(s_i', \pi_D(s_i'); \boldsymbol{w}^*)$ in Equation (1.3)
- $C2$: If s_i' is a terminal state or $\pi_D(s_i') = e$ (i.e., $g(s_i') > \boldsymbol{\phi}(s_i')^T \cdot \boldsymbol{w}$): Substitute $g(s_i')$ for $\hat{Q}(s_i', \pi_D(s_i'); \boldsymbol{w}^*)$ in Equation (1.3)

So we can rewrite Equation (1.3) using indicator notation $\mathbb{I}$ for cases $C1, C2$ as:

$$\sum_i \boldsymbol{\phi}(s_i) \cdot (\boldsymbol{\phi}(s_i)^T \cdot \boldsymbol{w}^* - \mathbb{I}_{C1} \cdot \gamma \cdot \boldsymbol{\phi}(s_i')^T \cdot \boldsymbol{w}^* - \mathbb{I}_{C2} \cdot \gamma \cdot g(s_i')) = 0$$

Factoring out $\boldsymbol{w}^*$, we get:

$$(\sum_i \boldsymbol{\phi}(s_i) \cdot (\boldsymbol{\phi}(s_i) - \mathbb{I}_{C1} \cdot \gamma \cdot \boldsymbol{\phi}(s_i'))^T) \cdot \boldsymbol{w}^* = \gamma \cdot \sum_i \mathbb{I}_{C2} \cdot \boldsymbol{\phi}(s_i) \cdot g(s_i')$$

This can be written in the familiar vector-matrix notation as: $\boldsymbol{A} \cdot \boldsymbol{w}^* = \boldsymbol{b}$

$$\boldsymbol{A} = \sum_i \boldsymbol{\phi}(s_i) \cdot (\boldsymbol{\phi}(s_i) - \mathbb{I}_{C1} \cdot \gamma \cdot \boldsymbol{\phi}(s_i'))^T$$

$$\boldsymbol{b} = \gamma \cdot \sum_i \mathbb{I}_{C2} \cdot \boldsymbol{\phi}(s_i) \cdot g(s_i')$$

The $m \times m$ Matrix $\boldsymbol{A}$ is accumulated at each atomic experience (s_i, s_i') as:

$$\boldsymbol{A} \leftarrow \boldsymbol{A} + \boldsymbol{\phi}(s_i) \cdot (\boldsymbol{\phi}(s_i) - \mathbb{I}_{C1} \cdot \gamma \cdot \boldsymbol{\phi}(s_i'))^T$$

The m-Vector $\boldsymbol{b}$ is accumulated at each atomic experience (s_i, s_i') as:

$$\boldsymbol{b} \leftarrow \boldsymbol{b} + \gamma \cdot \mathbb{I}_{C2} \cdot \boldsymbol{\phi}(s_i) \cdot g(s_i')$$

With Sherman-Morrison incremental inverse of $\boldsymbol{A}$, we can reduce the time-complexity from $O(m^3)$ to $O(m^2)$.

This solved $\boldsymbol{w}^*$ updates the Q-Value Function Approximation to $\hat{Q}(s, a; \boldsymbol{w}^*)$. This defines an updated, improved deterministic policy π_D' (serving as the *Deterministic Target Policy* for the next iteration of GPI):

$$\pi_D'(s) = \arg\max_a \hat{Q}(s, a; \boldsymbol{w}^*) \text{ for all } s \in \mathcal{N}$$

Li, Szepesvari, Schuurmans (Li, Szepesvári, and Schuurmans 2009) recommend in their paper to use 7 feature functions, the first 4 Laguerre polynomials that are functions of the underlying price and 3 functions of time. Precisely, the feature functions they recommend are:

- $\phi_0(S_t) = 1$
- $\phi_1(S_t) = e^{-\frac{M_t}{2}}$
- $\phi_2(S_t) = e^{-\frac{M_t}{2}} \cdot (1 - M_t)$
- $\phi_3(S_t) = e^{-\frac{M_t}{2}} \cdot (1 - 2M_t + M_t^2/2)$
- $\phi_0^{(t)}(t) = sin(\frac{\pi(T-t)}{2T})$
- $\phi_1^{(t)}(t) = \log(T - t)$
- $\phi_2^{(t)}(t) = (\frac{t}{T})^2$

where $M_t = \frac{S_t}{K}$ (S_t is the current underlying price and K is the American Option strike), t is the current time, and T is the expiration time (i.e., $0 \leq t < T$).

1.6.2 Deep Q-Learning for American Options Pricing

LSPI is data-efficient and compute-efficient, but linearity is a limitation in the function approximation. The alternative is (incremental) Q-Learning with neural network function approximation, which we cover in this subsection. We employ the same set up as LSPI (including Experience Replay) - specifically, the function approximation is required only for continuation value. Precisely,

$$\hat{Q}(s, a; \boldsymbol{w}) = \begin{cases} f(s; \boldsymbol{w}) & \text{if } a = c \\ g(s) & \text{if } a = e \end{cases} \text{ for all } s \in \mathcal{N}$$

where $f(s; \boldsymbol{w})$ is the deep neural network function approximation.

The Q-Learning update for each atomic experience (s_i, s_i') is:

$$\Delta \boldsymbol{w} = \alpha \cdot (\gamma \cdot \hat{Q}(s_i', \pi(s_i'); \boldsymbol{w}) - f(s_i; \boldsymbol{w})) \cdot \nabla_{\boldsymbol{w}} f(s_i; \boldsymbol{w})$$

When s_i' is a non-terminal state, the update is:

$$\Delta \boldsymbol{w} = \alpha \cdot (\gamma \cdot \max(g(s_i'), f(s_i'; \boldsymbol{w})) - f(s_i; \boldsymbol{w})) \cdot \nabla_{\boldsymbol{w}} f(s_i; \boldsymbol{w})$$

When s_i' is a terminal state, the update is:

$$\Delta \boldsymbol{w} = \alpha \cdot (\gamma \cdot g(s_i') - f(s_i; \boldsymbol{w})) \cdot \nabla_{\boldsymbol{w}} f(s_i; \boldsymbol{w})$$

1.7 Value Function Geometry

Now we look deeper into the issue of the *Deadly Triad* (that we had alluded to in Chapter **??**) by viewing Value Functions as Vectors (we had done this in Chapter **??**), understand Value Function Vector transformations with a balance of geometric intuition and mathematical rigor, providing insights into convergence issues for a variety of traditional loss functions used to develop RL algorithms. As ever, the best way to understand Vector transformations is to visualize it and so, we loosely refer to this topic as Value Function Geometry. The geometric intuition is particularly useful for linear function approximations. To promote intuition, we shall present this content for linear function approximations of the Value Function and stick to Prediction (rather than Control) although many of the concepts covered in this section are well-extensible to non-linear function approximations and to the Control problem.

This treatment was originally presented in the LSPI paper by Lagoudakis and Parr (Lagoudakis and Parr 2003) and has been covered in detail in the RL book by Sutton and Barto (Richard S. Sutton and Barto 2018). This treatment of Value Functions as Vectors leads us in the direction of overcoming the Deadly Triad by defining an appropriate loss function, calculating whose gradient provides a more robust set of RL algorithms known as Gradient Temporal-Difference (abbreviated, as Gradient TD), which we shall cover in the next section.

Along with visual intuition, it is important to write precise notation for Value Function transformations and approximations. So we start with a set of formal definitions, keeping the setting fairly simple and basic for ease of understanding.

1.7.1 Notation and Definitions

Assume our state space is finite without any terminal states, i.e. $\mathcal{S} = \mathcal{N} = \{s_1, s_2, \ldots, s_n\}$. Assume our action space $\mathcal{A}$ consists of a finite number of actions. This coverage can be extended to infinite/continuous spaces, but we shall stick to this simple setting in this section. Also, as mentioned above, we restrict this coverage to the case of a fixed (potentially stochastic) policy denoted as $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$. This means we are restricting to the case of the Prediction problem (although it's possible to extend some of this coverage to the case of Control).

We denote the Value Function for a policy π as $\boldsymbol{V}^\pi : \mathcal{S} \to \mathbb{R}$. Consider the n-dimensional vector space $\mathbb{R}^n$, with each dimension corresponding to a state in $\mathcal{S}$. Think of a Value Function (typically denoted $\boldsymbol{V}$): $\mathcal{S} \to \mathbb{R}$ as a vector in the $\mathbb{R}^n$ vector space. Each dimension's coordinate is the evaluation of the Value Function for that dimension's state. The coordinates of vector $\boldsymbol{V}^\pi$ for policy π are: $[\boldsymbol{V}^\pi(s_1), \boldsymbol{V}^\pi(s_2), \ldots, \boldsymbol{V}^\pi(s_n)]$. Note that this treatment is the same as the treatment in our coverage of Dynamic Programming in Chapter **??**.

Our interest is in identifying an appropriate function approximation of the Value Function $\boldsymbol{V}^\pi$. For the function approximation, assume there are m feature functions $\phi_1, \phi_2, \ldots, \phi_m : \mathcal{S} \to \mathbb{R}$, with $\boldsymbol{\phi}(s) \in \mathbb{R}^m$ denoting the feature vector for any state $s \in \mathcal{S}$. To keep things simple and to promote understanding of the concepts, we limit ourselves to linear function approximations. For linear function approximation of the Value Function with weights $\boldsymbol{w} = (w_1, w_2, \ldots, w_m)$, we use the notation $\boldsymbol{V_w} : \mathcal{S} \to \mathbb{R}$, defined as:

$$\boldsymbol{V_w}(s) = \boldsymbol{\phi}(s)^T \cdot \boldsymbol{w} = \sum_{j=1}^{m} \phi_j(s) \cdot w_j \text{ for all } s \in \mathcal{S}$$

.

Assuming independence of the feature functions, the m feature functions give us m independent vectors in the vector space $\mathbb{R}^n$. Feature function ϕ_j gives us the vector $[\phi_j(s_1), \phi_j(s_2), \ldots, \phi_j(s_n)] \in \mathbb{R}^n$. These m vectors are the m columns of the $n \times m$ matrix $\boldsymbol{\Phi} = [\phi_j(s_i)], 1 \leq i \leq n, 1 \leq j \leq m$. The span of these m independent vectors is an m-dimensional vector subspace within this n-dimensional vector space, spanned by the set of all $\boldsymbol{w} = (w_1, w_2, \ldots, w_m) \in \mathbb{R}^m$. The vector $\boldsymbol{V_w} = \boldsymbol{\Phi} \cdot \boldsymbol{w}$ in this vector subspace has coordinates $[\boldsymbol{V_w}(s_1), \boldsymbol{V_w}(s_2), \ldots, \boldsymbol{V_w}(s_n)]$. The vector $\boldsymbol{V_w}$ is fully specified by $\boldsymbol{w}$ (so we often say $\boldsymbol{w}$ to mean $\boldsymbol{V_w}$). Our interest is in identifying an appropriate $\boldsymbol{w} \in \mathbb{R}^m$ that represents an adequate linear function approximation $\boldsymbol{V_w} = \boldsymbol{\Phi} \cdot \boldsymbol{w}$ of the Value Function $\boldsymbol{V}^\pi$.

We denote the probability distribution of occurrence of states under policy π as $\boldsymbol{\mu_\pi} : \mathcal{S} \to [0, 1]$. In accordance with the notation we used in Chapter **??**, $\mathcal{R}(s, a)$ refers to the Expected Reward upon taking action a in state s, and $\mathcal{P}(s, a, s')$ refers to the probability of transition from state s to state s' upon taking action a. Define

$$\boldsymbol{\mathcal{R}}^\pi(s) = \sum_{a \in \mathcal{A}} \pi(s, a) \cdot \mathcal{R}(s, a) \text{ for all } s \in \mathcal{S}$$

$$\boldsymbol{\mathcal{P}}^\pi(s, s') = \sum_{a \in \mathcal{A}} \pi(s, a) \cdot \mathcal{P}(s, a, s') \text{ for all } s, s' in \mathcal{S}$$

to denote the Expected Reward and state transition probabilities respectively of the π-implied MRP.

$\boldsymbol{\mathcal{R}}^\pi$ refers to vector $[\boldsymbol{\mathcal{R}}^\pi(s_1), \boldsymbol{\mathcal{R}}^\pi(s_2), \ldots, \boldsymbol{\mathcal{R}}^\pi(s_n)]$ and $\boldsymbol{\mathcal{P}}^\pi$ refers to matrix $[\boldsymbol{\mathcal{P}}^\pi(s_i, s_{i'})], 1 \leq i, i' \leq n$. Denote $\gamma < 1$ (since there are no terminal states) as the MDP discount factor.

1.7.2 Bellman Policy Operator and Projection Operator

In Chapter **??**, we introduced the Bellman Policy Operator $\boldsymbol{B}^\pi$ for policy π operating on any Value Function vector $\boldsymbol{V}$. As a reminder,

$$\boldsymbol{B}^\pi(\boldsymbol{V}) = \boldsymbol{\mathcal{R}}^\pi + \gamma \boldsymbol{\mathcal{P}}^\pi \cdot \boldsymbol{V} \text{ for any VF vector } \boldsymbol{V} \in \mathbb{R}^n$$

Note that $\boldsymbol{B}^\pi$ is a linear operator in vector space $\mathbb{R}^n$. So we henceforth denote and treat $\boldsymbol{B}^\pi$ as an $n \times n$ matrix, representing the linear operator. We've learnt in Chapter **??** that $\boldsymbol{V}^\pi$ is the fixed point of $\boldsymbol{B}^\pi$. Therefore, we can write:

$$\boldsymbol{B}^\pi \cdot \boldsymbol{V}^\pi = \boldsymbol{V}^\pi$$

This means, if we start with an arbitrary Value Function vector $\boldsymbol{V}$ and repeatedly apply $\boldsymbol{B}^\pi$, by Banach Fixed-Point Theorem **??**, we will reach the fixed point $\boldsymbol{V}^\pi$. We've learnt in Chapter **??** that this is in fact the Dynamic Programming Policy Evaluation algorithm. Note that Tabular Monte Carlo also converges to $\boldsymbol{V}^\pi$ (albeit slowly).

Next, we introduce the Projection Operator $\boldsymbol{\Pi_\Phi}$ for the subspace spanned by the column vectors (feature functions) of $\boldsymbol{\Phi}$. We define $\boldsymbol{\Pi_\Phi}(\boldsymbol{V})$ as the vector in the subspace spanned by the column vectors of $\boldsymbol{\Phi}$ that represents the orthogonal projection of Value Function vector $\boldsymbol{V}$ on the $\boldsymbol{\Phi}$ subspace. To make this precise, we first define "distance" $d(\boldsymbol{V_1}, \boldsymbol{V_2})$ between Value Function vectors $\boldsymbol{V_1}, \boldsymbol{V_2}$, weighted by $\boldsymbol{\mu_\pi}$ across the n dimensions of $\boldsymbol{V_1}, \boldsymbol{V_2}$. Specifically,

$$d(\boldsymbol{V_1}, \boldsymbol{V_2}) = \sum_{i=1}^{n} \boldsymbol{\mu_\pi}(s_i) \cdot (\boldsymbol{V_1}(s_i) - \boldsymbol{V_2}(s_i))^2 = (\boldsymbol{V_1} - \boldsymbol{V_2})^T \cdot \boldsymbol{D} \cdot (\boldsymbol{V_1} - \boldsymbol{V_2})$$

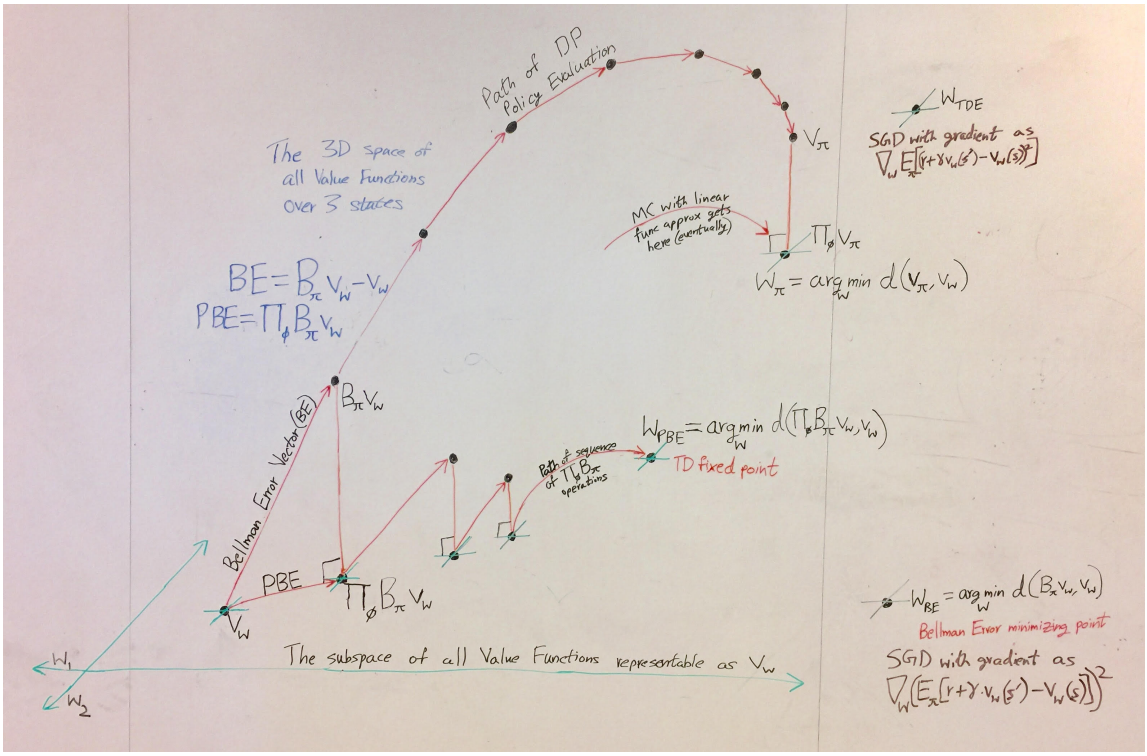

Figure 1.6: Value Function Geometry (Image Credit: Sutton-Barto's RL Book)

where $\boldsymbol{D}$ is the square diagonal matrix consisting of the diagonal elements $\boldsymbol{\mu_\pi}(s_i), 1 \leq i \leq n$.

With this "distance" metric, we define $\mathbf{\Pi_\Phi}(\boldsymbol{V})$ as the Value Function vector in the subspace spanned by the column vectors of $\mathbf{\Phi}$ that is given by $\arg\min_{\boldsymbol{w}} d(\boldsymbol{V}, \boldsymbol{V_w})$. This is a weighted least squares regression with solution:

$$\boldsymbol{w}^* = (\mathbf{\Phi}^T \cdot \boldsymbol{D} \cdot \mathbf{\Phi})^{-1} \cdot \mathbf{\Phi}^T \cdot \boldsymbol{D} \cdot \boldsymbol{V}$$

Since $\mathbf{\Pi_\Phi}(\boldsymbol{V}) = \mathbf{\Phi} \cdot \boldsymbol{w}^*$, we henceforth denote and treat Projection Operator $\mathbf{\Pi_\Phi}$ as the following $n \times n$ matrix:

$$\mathbf{\Pi_\Phi} = \mathbf{\Phi} \cdot (\mathbf{\Phi}^T \cdot \boldsymbol{D} \cdot \mathbf{\Phi})^{-1} \cdot \mathbf{\Phi}^T \cdot \boldsymbol{D}$$

1.7.3 Vectors of interest in the Φ subspace

In this section, we cover 4 Value Function vectors of interest in the $\mathbf{\Phi}$ subspace, as candidate linear function approximations of the Value Function $\boldsymbol{V}^\pi$. To lighten notation, we will refer to the $\mathbf{\Phi}$-subspace Value Function vectors by their corresponding weights $\boldsymbol{w}$. All 4 of these Value Function vectors are depicted in Figure 1.6, an image we are borrowing from Sutton and Barto's RL book (Richard S. Sutton and Barto 2018). We spend the rest of this section going over these 4 Value Function vectors in detail.

The first Value Function vector of interest in the $\mathbf{\Phi}$ subspace is the Projection $\mathbf{\Pi_\Phi} \cdot \boldsymbol{V}^\pi$, denoted as $\boldsymbol{w}_\pi = \arg\min_{\boldsymbol{w}} d(\boldsymbol{V}^\pi, \boldsymbol{V_w})$. This is the linear function approximation of the Value Function $\boldsymbol{V}^\pi$ we seek because it is the Value Function vector in the $\mathbf{\Phi}$ subspace that is "closest" to $\boldsymbol{V}^\pi$. Monte-Carlo with linear function approximation will (slowly) converge to $\boldsymbol{w}_\pi$. Figure 1.6 provides the visualization. We've learnt that Monte-Carlo can be

slow to converge, so we seek function approximations in the $\boldsymbol{\Phi}$ subspace that are based on Temporal-Difference (TD), i.e., bootstrapped methods. The remaining three Value Function vectors in the $\boldsymbol{\Phi}$ subspace are based on TD methods.

We denote the second Value Function vector of interest in the $\boldsymbol{\Phi}$ subspace as $\boldsymbol{w}_{BE}$. The acronym BE stands for *Bellman Error*. To understand this, consider the application of the Bellman Policy Operator $\boldsymbol{B}^\pi$ on a Value Function vector $\boldsymbol{V}_{\boldsymbol{w}}$ in the $\boldsymbol{\Phi}$ subspace. Applying $\boldsymbol{B}^\pi$ on $\boldsymbol{V}_{\boldsymbol{w}}$ typically throws $\boldsymbol{V}_{\boldsymbol{w}}$ out of the $\boldsymbol{\Phi}$ subspace. The idea is to find a Value Function vector $\boldsymbol{V}_{\boldsymbol{w}}$ in the $\boldsymbol{\Phi}$ subspace such that the "distance" between $\boldsymbol{V}_{\boldsymbol{w}}$ and $\boldsymbol{B}^\pi \cdot \boldsymbol{V}_{\boldsymbol{w}}$ is minimized, i.e. we minimize the "error vector" $BE = \boldsymbol{B}^\pi \cdot \boldsymbol{V}_{\boldsymbol{w}} - \boldsymbol{V}_{\boldsymbol{w}}$ (Figure 1.6 provides the visualization). Hence, we say we are minimizing the *Bellman Error* (or simply that we are minimizing BE), and we refer to w_{BE} as the Value Function vector in the $\boldsymbol{\Phi}$ subspace for which BE is minimized. Formally, we define it as:

$$\begin{aligned}
\boldsymbol{w}_{BE} &= \arg\min_{\boldsymbol{w}} d(\boldsymbol{B}^\pi \cdot \boldsymbol{V}_{\boldsymbol{w}}, \boldsymbol{V}_{\boldsymbol{w}}) \\
&= \arg\min_{\boldsymbol{w}} d(\boldsymbol{V}_{\boldsymbol{w}}, \boldsymbol{\mathcal{R}}^\pi + \gamma \boldsymbol{\mathcal{P}}^\pi \cdot \boldsymbol{V}_{\boldsymbol{w}}) \\
&= \arg\min_{\boldsymbol{w}} d(\boldsymbol{\Phi} \cdot \boldsymbol{w}, \boldsymbol{\mathcal{R}}^\pi + \gamma \boldsymbol{\mathcal{P}}^\pi \cdot \boldsymbol{\Phi} \cdot \boldsymbol{w}) \\
&= \arg\min_{\boldsymbol{w}} d(\boldsymbol{\Phi} \cdot \boldsymbol{w} - \gamma \boldsymbol{\mathcal{P}}^\pi \cdot \boldsymbol{\Phi} \cdot \boldsymbol{w}, \boldsymbol{\mathcal{R}}^\pi) \\
&= \arg\min_{\boldsymbol{w}} d((\boldsymbol{\Phi} - \gamma \boldsymbol{\mathcal{P}}^\pi \cdot \boldsymbol{\Phi}) \cdot \boldsymbol{w}, \boldsymbol{\mathcal{R}}^\pi)
\end{aligned}$$

This is a weighted least-squares linear regression of $\boldsymbol{\mathcal{R}}^\pi$ against $\boldsymbol{\Phi} - \gamma \boldsymbol{\mathcal{P}}^\pi \cdot \boldsymbol{\Phi}$ with weights $\boldsymbol{\mu}_{\boldsymbol{\pi}}$, whose solution is:

$$\boldsymbol{w}_{BE} = ((\boldsymbol{\Phi} - \gamma \boldsymbol{\mathcal{P}}^\pi \cdot \boldsymbol{\Phi})^T \cdot \boldsymbol{D} \cdot (\boldsymbol{\Phi} - \gamma \boldsymbol{\mathcal{P}}^\pi \cdot \boldsymbol{\Phi}))^{-1} \cdot (\boldsymbol{\Phi} - \gamma \boldsymbol{\mathcal{P}}^\pi \cdot \boldsymbol{\Phi})^T \cdot \boldsymbol{D} \cdot \boldsymbol{\mathcal{R}}^\pi$$

The above formulation can be used to compute $\boldsymbol{w}_{BE}$ if we know the model probabilities $\boldsymbol{\mathcal{P}}^\pi$ and reward function $\boldsymbol{\mathcal{R}}^\pi$. But often, in practice, we don't know $\boldsymbol{\mathcal{P}}^\pi$ and $\boldsymbol{\mathcal{R}}^\pi$, in which case we seek model-free learning of $\boldsymbol{w}_{BE}$, specifically with a TD (bootstrapped) algorithm.

Let us refer to

$$(\boldsymbol{\Phi} - \gamma \boldsymbol{\mathcal{P}}^\pi \cdot \boldsymbol{\Phi})^T \cdot \boldsymbol{D} \cdot (\boldsymbol{\Phi} - \gamma \boldsymbol{\mathcal{P}}^\pi \cdot \boldsymbol{\Phi})$$

as matrix $\boldsymbol{A}$ and let us refer to

$$(\boldsymbol{\Phi} - \gamma \boldsymbol{\mathcal{P}}^\pi \cdot \boldsymbol{\Phi})^T \cdot \boldsymbol{D} \cdot \boldsymbol{\mathcal{R}}^\pi$$

as vector $\boldsymbol{b}$ so that $\boldsymbol{w}_{BE} = \boldsymbol{A}^{-1} \cdot \boldsymbol{b}$.

Following policy π, each time we perform an individual transition from s to s' getting reward r, we get a sample estimate of $\boldsymbol{A}$ and $\boldsymbol{b}$. The sample estimate of $\boldsymbol{A}$ is the outer-product of vector $\boldsymbol{\phi}(s) - \gamma \cdot \boldsymbol{\phi}(s')$ with itself. The sample estimate of $\boldsymbol{b}$ is scalar r times vector $\boldsymbol{\phi}(s) - \gamma \cdot \boldsymbol{\phi}(s')$. We average these sample estimates across many such individual transitions. However, this requires m (the number of features) to be not too large.

If m is large or if we are doing non-linear function approximation or off-policy, then we seek a gradient-based TD algorithm. We defined $\boldsymbol{w}_{BE}$ as the vector in the $\boldsymbol{\Phi}$ subspace for which the Bellman Error is minimized. But Bellman Error for a state is the expectation of the TD error δ for that state when following policy π. So we want to do Stochastic Gradient

Descent with the gradient of the square of expected TD error, as follows:

$$\begin{aligned}\Delta \boldsymbol{w} &= -\alpha \cdot \frac{1}{2} \cdot \nabla_{\boldsymbol{w}} (\mathbb{E}_\pi[\delta])^2 \\ &= -\alpha \cdot \mathbb{E}_\pi[r + \gamma \cdot \boldsymbol{\phi}(s')^T \cdot \boldsymbol{w} - \boldsymbol{\phi}(s)^T \cdot \boldsymbol{w}] \cdot \nabla_{\boldsymbol{w}} \mathbb{E}_\pi[\delta] \\ &= \alpha \cdot (\mathbb{E}_\pi[r + \gamma \cdot \boldsymbol{\phi}(s')^T \cdot \boldsymbol{w}] - \boldsymbol{\phi}(s)^T \cdot \boldsymbol{w}) \cdot (\boldsymbol{\phi}(s) - \gamma \cdot \mathbb{E}_\pi[\boldsymbol{\phi}(s')])\end{aligned}$$

This is called the *Residual Gradient* algorithm, due to Leemon Baird (Baird 1995). It requires two independent samples of s' transitioning from s. If we do have that, it converges to $\boldsymbol{w}_{BE}$ robustly (even for non-linear function approximations). But this algorithm is slow, and doesn't converge to a desirable place. Another issue is that $\boldsymbol{w}_{BE}$ is not learnable if we can only access the features, and not underlying states. These issues led researchers to consider alternative TD algorithms.

We denote the third Value Function vector of interest in the $\boldsymbol{\Phi}$ subspace as $\boldsymbol{w}_{TDE}$ and define it as the vector in the $\boldsymbol{\Phi}$ subspace for which the expected square of the TD error δ (when following policy π) is minimized. Formally,

$$\boldsymbol{w}_{TDE} = \arg\min_{\boldsymbol{w}} \sum_{s \in \mathcal{S}} \boldsymbol{\mu}_{\boldsymbol{\pi}}(s) \sum_{r,s'} \mathbb{P}_\pi(r, s'|s) \cdot (r + \gamma \cdot \boldsymbol{\phi}(s')^T \cdot \boldsymbol{w} - \boldsymbol{\phi}(s)^T \cdot \boldsymbol{w})^2$$

To perform Stochastic Gradient Descent, we have to estimate the gradient of the expected square of TD error by sampling. The weight update for each gradient sample in the Stochastic Gradient Descent is:

$$\begin{aligned}\Delta \boldsymbol{w} &= -\alpha \cdot \frac{1}{2} \cdot \nabla_{\boldsymbol{w}} (r + \gamma \cdot \boldsymbol{\phi}(s')^T \cdot \boldsymbol{w} - \boldsymbol{\phi}(s)^T \cdot \boldsymbol{w})^2 \\ &= \alpha \cdot (r + \gamma \cdot \boldsymbol{\phi}(s')^T \cdot \boldsymbol{w} - \boldsymbol{\phi}(s)^T \cdot \boldsymbol{w}) \cdot (\boldsymbol{\phi}(s) - \gamma \cdot \boldsymbol{\phi}(s'))\end{aligned}$$

This algorithm is called *Naive Residual Gradient*, due to Leemon Baird (Baird 1995). Naive Residual Gradient converges robustly, but again, not to a desirable place. So researchers had to look even further.

This brings us to the fourth (and final) Value Function vector of interest in the $\boldsymbol{\Phi}$ subspace. We denote this Value Function vector as w_{PBE}. The acronym PBE stands for *Projected Bellman Error*. To understand this, first consider the composition of the Projection Operator $\boldsymbol{\Pi}_{\boldsymbol{\Phi}}$ and the Bellman Policy Operator $\boldsymbol{B}^\pi$, i.e., $\boldsymbol{\Pi}_{\boldsymbol{\Phi}} \cdot \boldsymbol{B}^\pi$ (we call this composed operator as the *Projected Bellman* operator). Visualize the application of this *Projected Bellman* operator on a Value Function vector $\boldsymbol{V}_{\boldsymbol{w}}$ in the $\boldsymbol{\Phi}$ subspace. Applying $\boldsymbol{B}^\pi$ on $\boldsymbol{V}_{\boldsymbol{w}}$ typically throws $\boldsymbol{V}_{\boldsymbol{w}}$ out of the $\boldsymbol{\Phi}$ subspace and then further applying $\boldsymbol{\Pi}_{\boldsymbol{\Phi}}$ brings it back to the $\boldsymbol{\Phi}$ subspace (call this resultant Value Function vector $\boldsymbol{V}_{\boldsymbol{w}'}$). The idea is to find a Value Function vector $\boldsymbol{V}_{\boldsymbol{w}}$ in the $\boldsymbol{\Phi}$ subspace for which the "distance" between $\boldsymbol{V}_{\boldsymbol{w}}$ and $\boldsymbol{V}_{\boldsymbol{w}'}$ is minimized, i.e. we minimize the "error vector" $PBE = \boldsymbol{\Pi}_{\boldsymbol{\Phi}} \cdot \boldsymbol{B}^\pi \cdot \boldsymbol{V}_{\boldsymbol{w}} - \boldsymbol{V}_{\boldsymbol{w}}$ (Figure 1.6 provides the visualization). Hence, we say we are minimizing the *Projected Bellman Error* (or simply that we are minimizing PBE), and we refer to w_{PBE} as the Value Function vector in the $\boldsymbol{\Phi}$ subspace for which PBE is minimized. It turns out that the minimum of PBE is actually zero, i.e., $\boldsymbol{\Phi} \cdot \boldsymbol{w}_{PBE}$ is a fixed point of operator $\boldsymbol{\Pi}_{\boldsymbol{\Phi}} \cdot \boldsymbol{B}^\pi$. Let us write out this statement formally. We know:

$$\boldsymbol{\Pi}_{\boldsymbol{\Phi}} = \boldsymbol{\Phi} \cdot (\boldsymbol{\Phi}^T \cdot \boldsymbol{D} \cdot \boldsymbol{\Phi})^{-1} \cdot \boldsymbol{\Phi}^T \cdot \boldsymbol{D}$$

$$\boldsymbol{B}^\pi \cdot \boldsymbol{V} = \boldsymbol{\mathcal{R}}^\pi + \gamma \boldsymbol{\mathcal{P}}^\pi \cdot \boldsymbol{V}$$

Therefore, the statement that $\boldsymbol{\Phi} \cdot \boldsymbol{w}_{PBE}$ is a fixed point of operator $\boldsymbol{\Pi}_{\boldsymbol{\Phi}} \cdot \boldsymbol{B}^\pi$ can be written as follows:

$$\boldsymbol{\Phi} \cdot (\boldsymbol{\Phi}^T \cdot \boldsymbol{D} \cdot \boldsymbol{\Phi})^{-1} \cdot \boldsymbol{\Phi}^T \cdot \boldsymbol{D} \cdot (\boldsymbol{\mathcal{R}}^\pi + \gamma \boldsymbol{\mathcal{P}}^\pi \cdot \boldsymbol{\Phi} \cdot \boldsymbol{w}_{PBE}) = \boldsymbol{\Phi} \cdot \boldsymbol{w}_{PBE}$$

Since the columns of $\boldsymbol{\Phi}$ are assumed to be independent (full rank),

$$\begin{aligned}
(\boldsymbol{\Phi}^T \cdot \boldsymbol{D} \cdot \boldsymbol{\Phi})^{-1} \cdot \boldsymbol{\Phi}^T \cdot \boldsymbol{D} \cdot (\boldsymbol{\mathcal{R}}^\pi + \gamma \boldsymbol{\mathcal{P}}^\pi \cdot \boldsymbol{\Phi} \cdot \boldsymbol{w}_{PBE}) &= \boldsymbol{w}_{PBE} i \\
\boldsymbol{\Phi}^T \cdot \boldsymbol{D} \cdot (\boldsymbol{\mathcal{R}}^\pi + \gamma \boldsymbol{\mathcal{P}}^\pi \cdot \boldsymbol{\Phi} \cdot \boldsymbol{w}_{PBE}) &= \boldsymbol{\Phi}^T \cdot \boldsymbol{D} \cdot \boldsymbol{\Phi} \cdot \boldsymbol{w}_{PBE} \\
\boldsymbol{\Phi}^T \cdot \boldsymbol{D} \cdot (\boldsymbol{\Phi} - \gamma \boldsymbol{\mathcal{P}}^\pi \cdot \boldsymbol{\Phi}) \cdot \boldsymbol{w}_{PBE} &= \boldsymbol{\Phi}^T \cdot \boldsymbol{D} \cdot \boldsymbol{\mathcal{R}}^\pi \qquad (1.4)
\end{aligned}$$

This is a square linear system of the form $\boldsymbol{A} \cdot \boldsymbol{w}_{PBE} = \boldsymbol{b}$ whose solution is:

$$\boldsymbol{w}_{PBE} = \boldsymbol{A}^{-1} \cdot \boldsymbol{b} = (\boldsymbol{\Phi}^T \cdot \boldsymbol{D} \cdot (\boldsymbol{\Phi} - \gamma \boldsymbol{\mathcal{P}}^\pi \cdot \boldsymbol{\Phi}))^{-1} \cdot \boldsymbol{\Phi}^T \cdot \boldsymbol{D} \cdot \boldsymbol{\mathcal{R}}^\pi$$

The above formulation can be used to compute $\boldsymbol{w}_{PBE}$ if we know the model probabilities $\boldsymbol{\mathcal{P}}^\pi$ and reward function $\boldsymbol{\mathcal{R}}^\pi$. But often, in practice, we don't know $\boldsymbol{\mathcal{P}}^\pi$ and $\boldsymbol{\mathcal{R}}^\pi$, in which case we seek model-free learning of $\boldsymbol{w}_{PBE}$, specifically with a TD (bootstrapped) algorithm.

The question is how do we construct matrix

$$\boldsymbol{A} = \boldsymbol{\Phi}^T \cdot \boldsymbol{D} \cdot (\boldsymbol{\Phi} - \gamma \boldsymbol{\mathcal{P}}^\pi \cdot \boldsymbol{\Phi})$$

and vector

$$\boldsymbol{b} = \boldsymbol{\Phi}^T \cdot \boldsymbol{D} \cdot \boldsymbol{\mathcal{R}}^\pi$$

without a model?

Following policy π, each time we perform an individual transition from s to s' getting reward r, we get a sample estimate of $\boldsymbol{A}$ and $\boldsymbol{b}$. The sample estimate of $\boldsymbol{A}$ is the outer-product of vectors $\boldsymbol{\phi}(s)$ and $\boldsymbol{\phi}(s) - \gamma \cdot \boldsymbol{\phi}(s')$. The sample estimate of $\boldsymbol{b}$ is scalar r times vector $\boldsymbol{\phi}(s)$. We average these sample estimates across many such individual transitions. Note that this algorithm is exactly the Least Squares Temporal Difference (LSTD) algorithm we've covered earlier in this chapter. Thus, we now know that LSTD converges to w_{PBE}, i.e., minimizes (in fact takes down to 0) PBE. If the number of features m is large or if we are doing non-linear function approximation or Off-Policy, then we seek a gradient-based TD algorithm. It turns out that our usual Semi-Gradient TD algorithm converges to $\boldsymbol{w}_{PBE}$ in the case of on-policy linear function approximation. Note that the update for the usual Semi-Gradient TD algorithm in the case of on-policy linear function approximation is as follows:

$$\Delta \boldsymbol{w} = \alpha \cdot (r + \gamma \cdot \boldsymbol{\phi}(s')^T \cdot \boldsymbol{w} - \boldsymbol{\phi}(s)^T \cdot \boldsymbol{w}) \cdot \boldsymbol{\phi}(s)$$

This converges to $\boldsymbol{w}_{PBE}$ because at convergence, we have: $\mathbb{E}_\pi[\Delta \boldsymbol{w}] = 0$, which can be expressed as:

$$\boldsymbol{\Phi}^T \cdot \boldsymbol{D} \cdot (\boldsymbol{\mathcal{R}}^\pi + \gamma \boldsymbol{\mathcal{P}}^\pi \cdot \boldsymbol{\Phi} \cdot \boldsymbol{w} - \boldsymbol{\Phi} \cdot \boldsymbol{w}) = 0$$

$$\Rightarrow \boldsymbol{\Phi}^T \cdot \boldsymbol{D} \cdot (\boldsymbol{\Phi} - \gamma \boldsymbol{\mathcal{P}}^\pi \cdot \boldsymbol{\Phi}) \cdot \boldsymbol{w} = \boldsymbol{\Phi}^T \cdot \boldsymbol{D} \cdot \boldsymbol{\mathcal{R}}^\pi$$

which is satisfied for $\boldsymbol{w} = \boldsymbol{w}_{PBE}$ (as seen from Equation (1.4)).

1.8 Gradient Temporal-Difference (Gradient TD)

For on-policy linear function approximation, the semi-gradient TD algorithm gives us w_{PBE}. But to obtain w_{PBE} in the case of non-linear function approximation or in the case of Off-Policy, we need a different approach. The different approach is Gradient Temporal-Difference (abbreviated, Gradient TD), the subject of this section.

The original Gradient TD algorithm, due to Sutton, Szepesvari, Maei (R. S. Sutton, Szepesvári, and Maei 2008) is typically abbreviated as GTD. Researchers then came up with a second-generation Gradient TD algorithm (R. S. Sutton et al. 2009), which is typically abbreviated as GTD-2. The same researchers also came up with a TD algorithm with Gradient Correction (R. S. Sutton et al. 2009), which is typically abbreviated as TDC.

We now cover the TDC algorithm. For simplicity of articulation and ease of understanding, we restrict to the case of linear function approximation in our coverage of the TDC algorithm below. However, do bear in mind that much of the concepts below extend to non-linear function approximation (which is where we reap the benefits of Gradient TD).

Our first task is to set up the appropriate loss function whose gradient will drive the Stochastic Gradient Descent.

$$\boldsymbol{w}_{PBE} = \arg\min_{\boldsymbol{w}} d(\boldsymbol{\Pi}_{\boldsymbol{\Phi}} \cdot \boldsymbol{B}^{\pi} \cdot \boldsymbol{V}_{\boldsymbol{w}}, \boldsymbol{V}_{\boldsymbol{w}}) = \arg\min_{\boldsymbol{w}} d(\boldsymbol{\Pi}_{\boldsymbol{\Phi}} \cdot \boldsymbol{B}^{\pi} \cdot \boldsymbol{V}_{\boldsymbol{w}}, \boldsymbol{\Pi}_{\boldsymbol{\Phi}} \cdot \boldsymbol{V}_{\boldsymbol{w}})$$

So we define the loss function (denoting $\boldsymbol{B}^{\pi} \cdot \boldsymbol{V}_{\boldsymbol{w}} - \boldsymbol{V}_{\boldsymbol{w}}$ as $\boldsymbol{\delta}_{\boldsymbol{w}}$) as:

$$\mathcal{L}(\boldsymbol{w}) = (\boldsymbol{\Pi}_{\boldsymbol{\Phi}} \cdot \boldsymbol{\delta}_{\boldsymbol{w}})^T \cdot \boldsymbol{D} \cdot (\boldsymbol{\Pi}_{\boldsymbol{\Phi}} \cdot \boldsymbol{\delta}_{\boldsymbol{w}}) = \boldsymbol{\delta}_{\boldsymbol{w}}^T \cdot \boldsymbol{\Pi}_{\boldsymbol{\Phi}}^T \cdot \boldsymbol{D} \cdot \boldsymbol{\Pi}_{\boldsymbol{\Phi}} \cdot \boldsymbol{\delta}_{\boldsymbol{w}}$$

$$= \boldsymbol{\delta}_{\boldsymbol{w}}^T \cdot (\boldsymbol{\Phi} \cdot (\boldsymbol{\Phi}^T \cdot \boldsymbol{D} \cdot \boldsymbol{\Phi})^{-1} \cdot \boldsymbol{\Phi}^T \cdot \boldsymbol{D})^T \cdot \boldsymbol{D} \cdot (\boldsymbol{\Phi} \cdot (\boldsymbol{\Phi}^T \cdot \boldsymbol{D} \cdot \boldsymbol{\Phi})^{-1} \cdot \boldsymbol{\Phi}^T \cdot \boldsymbol{D}) \cdot \boldsymbol{\delta}_{\boldsymbol{w}}$$

$$= \boldsymbol{\delta}_{\boldsymbol{w}}^T \cdot (\boldsymbol{D} \cdot \boldsymbol{\Phi} \cdot (\boldsymbol{\Phi}^T \cdot \boldsymbol{D} \cdot \boldsymbol{\Phi})^{-1} \cdot \boldsymbol{\Phi}^T) \cdot \boldsymbol{D} \cdot (\boldsymbol{\Phi} \cdot (\boldsymbol{\Phi}^T \cdot \boldsymbol{D} \cdot \boldsymbol{\Phi})^{-1} \cdot \boldsymbol{\Phi}^T \cdot \boldsymbol{D}) \cdot \boldsymbol{\delta}_{\boldsymbol{w}}$$

$$= (\boldsymbol{\delta}_{\boldsymbol{w}}^T \cdot \boldsymbol{D} \cdot \boldsymbol{\Phi}) \cdot (\boldsymbol{\Phi}^T \cdot \boldsymbol{D} \cdot \boldsymbol{\Phi})^{-1} \cdot (\boldsymbol{\Phi}^T \cdot \boldsymbol{D} \cdot \boldsymbol{\Phi}) \cdot (\boldsymbol{\Phi}^T \cdot \boldsymbol{D} \cdot \boldsymbol{\Phi})^{-1} \cdot (\boldsymbol{\Phi}^T \cdot \boldsymbol{D} \cdot \boldsymbol{\delta}_{\boldsymbol{w}})$$

$$= (\boldsymbol{\Phi}^T \cdot \boldsymbol{D} \cdot \boldsymbol{\delta}_{\boldsymbol{w}})^T \cdot (\boldsymbol{\Phi}^T \cdot \boldsymbol{D} \cdot \boldsymbol{\Phi})^{-1} \cdot (\boldsymbol{\Phi}^T \cdot \boldsymbol{D} \cdot \boldsymbol{\delta}_{\boldsymbol{w}})$$

We derive the TDC Algorithm based on $\nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w})$.

$$\nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}) = 2 \cdot (\nabla_{\boldsymbol{w}} (\boldsymbol{\Phi}^T \cdot \boldsymbol{D} \cdot \boldsymbol{\delta}_{\boldsymbol{w}})^T) \cdot (\boldsymbol{\Phi}^T \cdot \boldsymbol{D} \cdot \boldsymbol{\Phi})^{-1} \cdot (\boldsymbol{\Phi}^T \cdot \boldsymbol{D} \cdot \boldsymbol{\delta}_{\boldsymbol{w}})$$

We want to estimate this gradient from individual transitions data. So we express each of the 3 terms forming the product in the gradient expression above as expectations of functions of individual transitions $s \xrightarrow{\pi} (r, s')$. Denoting $r + \gamma \cdot \boldsymbol{\phi}(s')^T \cdot \boldsymbol{w} - \boldsymbol{\phi}(s)^T \cdot \boldsymbol{w}$ as δ, we get:

$$\boldsymbol{\Phi}^T \cdot \boldsymbol{D} \cdot \boldsymbol{\delta}_{\boldsymbol{w}} = \mathbb{E}[\delta \cdot \boldsymbol{\phi}(s)]$$

$$\nabla_{\boldsymbol{w}} (\boldsymbol{\Phi}^T \cdot \boldsymbol{D} \cdot \boldsymbol{\delta}_{\boldsymbol{w}})^T = \mathbb{E}[(\nabla_{\boldsymbol{w}} \delta) \cdot \boldsymbol{\phi}(s)^T] = \mathbb{E}[(\gamma \cdot \boldsymbol{\phi}(s') - \boldsymbol{\phi}(s)) \cdot \boldsymbol{\phi}(s)^T]$$

$$\boldsymbol{\Phi}^T \cdot \boldsymbol{D} \cdot \boldsymbol{\Phi} = \mathbb{E}[\boldsymbol{\phi}(s) \cdot \boldsymbol{\phi}(s)^T]$$

Substituting, we get:

$$\nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w}) = 2 \cdot \mathbb{E}[(\gamma \cdot \boldsymbol{\phi}(s') - \boldsymbol{\phi}(s)) \cdot \boldsymbol{\phi}(s)^T] \cdot (\mathbb{E}[\boldsymbol{\phi}(s) \cdot \boldsymbol{\phi}(s)^T])^{-1} \cdot \mathbb{E}[\delta \cdot \boldsymbol{\phi}(s)]$$

$$\Delta \boldsymbol{w} = -\alpha \cdot \frac{1}{2} \cdot \nabla_{\boldsymbol{w}} \mathcal{L}(\boldsymbol{w})$$

$$= \alpha \cdot \mathbb{E}[(\boldsymbol{\phi}(s) - \gamma \cdot \boldsymbol{\phi}(s')) \cdot \boldsymbol{\phi}(s)^T] \cdot (\mathbb{E}[\boldsymbol{\phi}(s) \cdot \boldsymbol{\phi}(s)^T])^{-1} \cdot \mathbb{E}[\delta \cdot \boldsymbol{\phi}(s)]$$
$$= \alpha \cdot (\mathbb{E}[\boldsymbol{\phi}(s) \cdot \boldsymbol{\phi}(s)^T] - \gamma \cdot \mathbb{E}[\boldsymbol{\phi}(s') \cdot \boldsymbol{\phi}(s)^T]) \cdot (\mathbb{E}[\boldsymbol{\phi}(s) \cdot \boldsymbol{\phi}(s)^T])^{-1} \cdot \mathbb{E}[\delta \cdot \boldsymbol{\phi}(s)]$$
$$= \alpha \cdot (\mathbb{E}[\delta \cdot \boldsymbol{\phi}(s)] - \gamma \cdot \mathbb{E}[\boldsymbol{\phi}(s') \cdot \boldsymbol{\phi}(s)^T] \cdot (\mathbb{E}[\boldsymbol{\phi}(s) \cdot \boldsymbol{\phi}(s)^T])^{-1} \cdot \mathbb{E}[\delta \cdot \boldsymbol{\phi}(s)])$$
$$= \alpha \cdot (\mathbb{E}[\delta \cdot \boldsymbol{\phi}(s)] - \gamma \cdot \mathbb{E}[\boldsymbol{\phi}(s') \cdot \boldsymbol{\phi}(s)^T] \cdot \boldsymbol{\theta})$$

where $\boldsymbol{\theta} = (\mathbb{E}[\boldsymbol{\phi}(s) \cdot \boldsymbol{\phi}(s)^T])^{-1} \cdot \mathbb{E}[\delta \cdot \boldsymbol{\phi}(s)]$ is the solution to the weighted least-squares linear regression of $\boldsymbol{B}^\pi \cdot \boldsymbol{V} - \boldsymbol{V}$ against $\boldsymbol{\Phi}$, with weights as μ_π.

We can perform this gradient descent with a technique known as *Cascade Learning*, which involves simultaneously updating both $\boldsymbol{w}$ and $\boldsymbol{\theta}$ (with $\boldsymbol{\theta}$ converging faster). The updates are as follows:

$$\Delta \boldsymbol{w} = \alpha \cdot \delta \cdot \boldsymbol{\phi}(s) - \alpha \cdot \gamma \cdot \boldsymbol{\phi}(s') \cdot (\boldsymbol{\phi}(s)^T \cdot \boldsymbol{\theta})$$
$$\Delta \boldsymbol{\theta} = \beta \cdot (\delta - \boldsymbol{\phi}(s)^T \cdot \boldsymbol{\theta}) \cdot \boldsymbol{\phi}(s)$$

where β is the learning rate for $\boldsymbol{\theta}$. Note that $\boldsymbol{\phi}(s)^T \cdot \boldsymbol{\theta}$ operates as an estimate of the TD error δ for current state s.

Repeating what we had said in Chapter **??**, Gradient TD converges reliably for the Prediction problem even when we are faced with the Deadly Triad of [Bootstrapping, Off-Policy, Non-Linear Function Approximation]. The picture is less rosy for Control. Gradient Q-Learning (Gradient TD for Off-Policy Control) converges reliably for both on-policy and off-policy linear function approximations, but there are divergence issues for non-linear function approximations. For Control problems with non-linear function approximations (especially, neural network approximations with off-policy learning), one can leverage the approach of the DQN algorithm (Experience Replay with fixed Target Network helps overcome the Deadly Triad).

1.9 Key Takeaways from this Chapter

- Batch RL makes efficient use of data.
- DQN uses Experience-Replay and fixed Q-learning targets, avoiding the pitfalls of time-correlation and varying TD Target.
- LSTD is a direct (gradient-free) solution of Batch TD Prediction.
- LSPI is an Off-Policy, Experience-Replay Control Algorithm using LSTDQ for Policy Evaluation.
- Optimal Exercise of American Options can be tackled with LSPI and Deep Q-Learning algorithms.
- For Prediction, the 4 Value Function vectors of interest in the $\boldsymbol{\Phi}$ subspace are $\boldsymbol{w}_\pi, \boldsymbol{w}_{BE}, \boldsymbol{w}_{TDE}, \boldsymbol{w}_{PBE}$ with $\boldsymbol{w}_{PBE}$ as the key sought-after function approximation for Value Function $\boldsymbol{V}^\pi$.
- For Prediction, Gradient TD solves for $\boldsymbol{w}_{PBE}$ efficiently and robustly in the case of non-linear function approximation and in the case of Off-Policy.

Bibliography

Baird, Leemon. 1995. "Residual Algorithms: Reinforcement Learning with Function Approximation." In *Machine Learning Proceedings 1995*, edited by Armand Prieditis and Stuart Russell, 30–37. San Francisco (CA): Morgan Kaufmann. https://doi.org/https://doi.org/10.1016/B978-1-55860-377-6.50013-X.

Bradtke, Steven J., and Andrew G. Barto. 1996. "Linear Least-Squares Algorithms for Temporal Difference Learning." *Mach. Learn.* 22 (1-3): 33–57. http://dblp.uni-trier.de/db/journals/ml/ml22.html#BradtkeB96.

Lagoudakis, Michail G., and Ronald Parr. 2003. "Least-Squares Policy Iteration." *J. Mach. Learn. Res.* 4: 1107–49. http://dblp.uni-trier.de/db/journals/jmlr/jmlr4.html#LagoudakisP03.

Li, Y., Cs. Szepesvári, and D. Schuurmans. 2009. "Learning Exercise Policies for American Options." In *AISTATS*, 5:352–59. http://www.ics.uci.edu/~aistats/.

Lin, Long J. 1993. "Reinforcement Learning for Robots Using Neural Networks." PhD thesis, Pittsburg: CMU.

Longstaff, Francis A., and Eduardo S. Schwartz. 2001. "Valuing American Options by Simulation: A Simple Least-Squares Approach." *Review of Financial Studies* 14 (1): 113–47. https://doi.org/10.1093/rfs/14.1.113.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. "Playing Atari with Deep Reinforcement Learning." http://arxiv.org/abs/1312.5602.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, et al. 2015. "Human-Level Control Through Deep Reinforcement Learning." *Nature* 518 (7540): 529–33. https://doi.org/10.1038/nature14236.

Sherman, Jack, and Winifred J. Morrison. 1950. "Adjustment of an Inverse Matrix Corresponding to a Change in One Element of a Given Matrix." *The Annals of Mathematical Statistics* 21 (1): 124–27. https://doi.org/10.1214/aoms/1177729893.

Sutton, R. S., H. R. Maei, D. Precup, S. Bhatnagar, D. Silver, Cs. Szepesvári, and E. Wiewiora. 2009. "Fast Gradient-Descent Methods for Temporal-Difference Learning with Linear Function Approximation." In *ICML*, 993–1000.

Sutton, R. S., Cs. Szepesvári, and H. R. Maei. 2008. "A Convergent O(n) Algorithm for Off-Policy Temporal-Difference Learning with Linear Function Approximation." In *NIPS*, 1609–16.

Sutton, Richard S., and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction*. Second. The MIT Press. http://incompleteideas.net/book/the-book-2nd.html.