Week 6: Approximate Dynamic Programming With Function Approximation

Ardhito Nurhadyansah[1], Mohamad Arvin Fadriansyah[2], and Ian Suryadi Timothy H[3]

[1]2106750206
[2]2006596996
[3]2106750875

1 Reinforcement Learning Developments

Recently, Sutton and Barto received the Turing Award for their contributions in developing the foundations of Reinforcement Learning. In regards of those contributions, they also adopt the concepts coming from other fields, such as Psychology and Neuroscience, as the study of true intelligence are coming from those fields.

In RL, an agent have two main properties, i.e. to **percieve** and **act**. Likely, the future developments of AI will continue upon RL, after the development of the other subfields like in Supervised and Unsupervised Learning supporting the perceiving part of RL agent. Below is the illustration of RL scheme, where besides perceiving, an agent is also taking action, then the environment will evaluate those actions with evaluative feedback (with reward) rather than instructive feedback (giving the true answer) like in Supervised Learning.

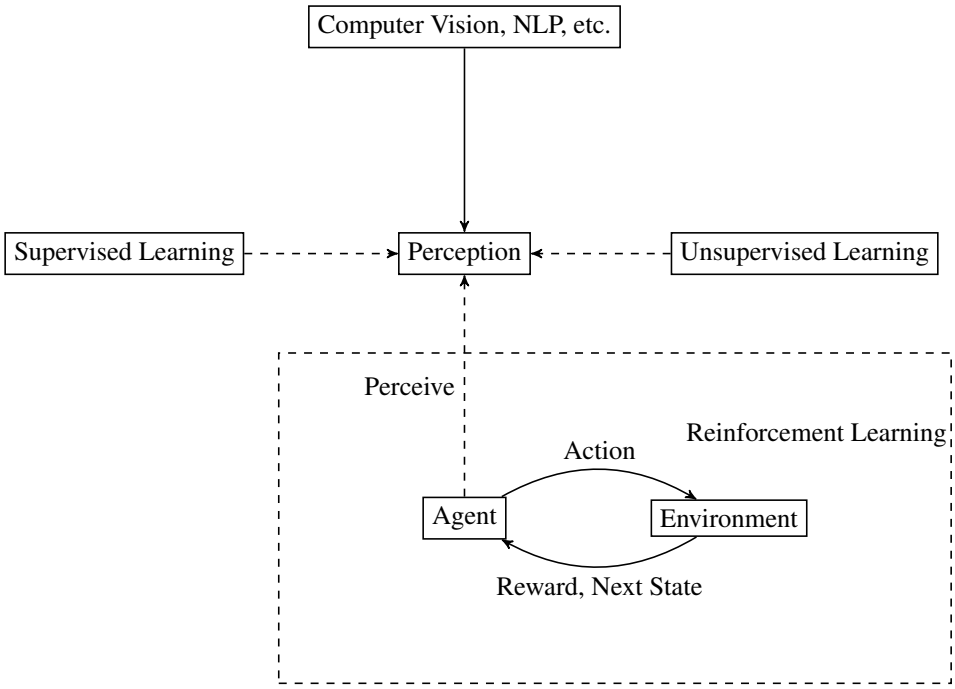

Figure 1: Reinforcement Learning Scheme with Perception

2 RL Scheme For Attaining Optimal Policy

In Finite Unichain MDP, an optimal policy means a mapping from each state to action (deterministic) that gives the most optimal value to the agent with respect to a specific optimality criterion, e.g: discounted reward, average reward, total reward, bias. In RL, the two major schemes to find that out are:

1. Value Iteration (VI)
   The notion of VI consists of these two main steps:
   - **Value Update**: Iteratively update the value function based on the Bellman equation until it converge.
   - **Policy Extraction**: Derive the optimal policy with respect to the optimal value function.

   From VI, we can derive Q_x^*-learning, where x denotes the optimality criteria, e.g., gain, bias , γ, or total reward. Note that in the average reward setting, we interested in Q_b^* instead of Q_g^*.

2. Policy Iteration (PI)
   Policy Iteration is an algorithm that iteratively evaluates and improves a policy until it converges to the optimal policy. It consists of two main steps:
   - **Policy Evaluation**: Calculate the policy's value.
   - **Policy Improvement**: Update the policy by choosing actions that maximize the value.

   In RL setting, we can't do exact calculations for PI. Thus, we need to derive the approach to the inexact (approximate) way, and this is called Generalized Policy Iteration (GPI).

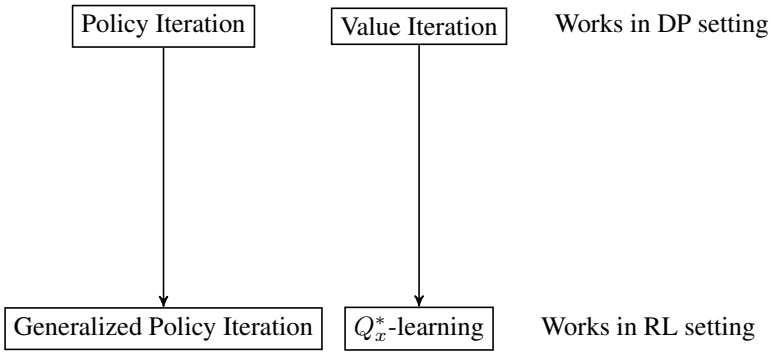

Figure 2: Major RL Schemes To Find Optimal Policy

Some Questions To Ponder:

- Which is more intuitive?
- Which is more suitable to use? Remember the "no free lunch theorem": sometimes PI is better than VI, and vice versa.

3 Inexact Policy Evaluation

In DP setting, we can exactly evaluate a policy via $v_x(\pi, s)$, where x is the optimality criteria. This can also be denoted as $v_x(s)$ whenever the policy of interest is clear from context. However, in RL setting, we can't do that as:

- The state transition probability (P) and reward scheme (R) is unknown.
- We need to use tabular method to store a separate value for each state (or state-action pair), which is infeasible for very large state spaces.

3.1 Function Approximation

To get rid of the problems that present in RL setting compared to DP setting, we can use approximation by defining policy value function approximator. Based on its characteristics, we can classify the function approximator into:

3.1.1 Parametric Function Approximators

Parametric function approximators are characterized by having a fixed and finite number of parameters. These parameters are adjusted during the learning process to approximate the true value function. Parametric approximators can be further classified into:

Linear Function Approximators: In linear function approximation, the value function is represented as a linear combination of features, e.g., we can define feature vector, weight those feature vectors, and sum them up altogether to get the approximation.

Non-linear Function Approximators: In non-linear function approximation, the value function is represented by a non-linear function of the parameters and features, e.g.,:

- Neural Networks: the value function is approximated by the output of the neural network, with the weights and biases of the network serving as the parameters.
- Decision Trees: can partition the state space into regions and assign values to each region.

3.1.2 Non-Parametric Function Approximators

Non-parametric function approximators do not have a fixed set of parameters. The complexity of the model grows with the amount of data, e.g.,:

- Nearest Neighbors: The value of a state is estimated based on the values of its nearest neighbors in the state space.
- Kernel Methods: Kernel methods use kernel functions to measure the similarity between states and estimate the value function.

4 Linear Function Approximation

As explained in 3.1.1, the approach is to represent the value function approximation as a linear combination of features. For that, we define feature vector $\mathbf{f}(s) \in \mathbb{R}^n$ for each state s, and a weight vector $\mathbf{w} \in \mathbb{R}^n$, where n is dim(w) as this is linear setting, to form the policy value function approximator:

$$\hat{v}(s; \mathbf{w}) \;=\; \mathbf{w}^\top \mathbf{f}(s). \approx v(s)$$

In matrix form (for a finite state set), we will have F as the $|S| \times dim(W)$ matrix of all feature vectors and $\mathbf{w}$ vector with size $dim(W) \times 1$ to form:

$$\hat{\mathbf{v}} \;=\; F\,\mathbf{w},$$

where $\hat{\mathbf{v}}$ is the $|S| \times 1$ vector of approximate values.

Then, what's next after we define the value function approximator is to learn the weights $\mathbf{w}$ that minimize the error between the true value function and the approximated value function. We can use Mean Squared Error (MSE) or Projected Bellman Error (PBE) as the criterion for the best approximation.

To find the optimal weights $\mathbf{w}^*$, we solve the following optimization problem:

$$\mathbf{w}^* = \underset{\mathbf{w} \in \mathbb{R}^{\dim(\mathbf{w})}}{\arg\min}\; e_x(\mathbf{w}),$$

where the subscript x denotes the type of error, e.g., mean squared (MS) or projected Bellman (PB).

5 Linear Function Approximation with (Weighted) Mean Squared Error

In this scheme, we'll have $p^\star(s)$, a stationary distribution over states that assigns a weight to each state. Note that we use it instead of uniform weighting, i.e., $\frac{1}{|S|}$, as some states may be visited less frequently or not at all by the agent. Then, we define the weighted mean squared error (MSE) as:

$$e_{\text{MS}}(\mathbf{w}) = \mathbb{E}_{s \sim p^\star(s)} \Big[\big(v_\pi(s) - \hat{v}(s; \mathbf{w}) \big)^2 \Big].$$

In discrete state spaces, we can write it down as:

$$e_{\text{MS}}(\mathbf{w}) \;=\; \sum_{s \in \mathcal{S}} p^\star(s) \, \big(v_\pi(s) - \hat{v}(s; \mathbf{w}) \big)^2.$$

$$e_{\text{MS}}(\mathbf{w}) \;=\; \sum_{s\in\mathcal{S}} p^\star(s)\,\big(v_\pi(s) - \mathbf{w}^\top \mathbf{f}(s)\big)^2.$$

In matrix form:

$$e_{\text{MS}}(\mathbf{w}) \;=\; (\mathbf{v}_\pi - F\,\mathbf{w})^\top D_{p^\star}\,(\mathbf{v}_\pi - F\,\mathbf{w}),$$

where $D_{p^\star}$ is a diagonal matrix with $D_{p^\star}(s,s) = p^\star(s)$.

To get the optimal weights $\mathbf{w}^*$ that minimize the weighted MSE, we can use the gradient descent algorithm. The update rule for the weights is:

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} e_{\text{MS}}(\mathbf{w}),$$

where α is the learning rate and $\nabla_{\mathbf{w}} e_{\text{MS}}(\mathbf{w})$ is the gradient with respect to $\mathbf{w}$ of $e_{\text{MS}}(\mathbf{w})$.

Starting with the gradient of the weighted MSE:

$$\nabla_{\mathbf{w}} e_{\text{MS}}(\mathbf{w}) = \nabla_{\mathbf{w}} \mathbb{E}_{s\sim p^\star(s)}\Big[\big(v_\pi(s) - \hat{v}(s;\mathbf{w})\big)^2\Big].$$

We move the gradient inside the expectation:

$$\nabla_{\mathbf{w}} e_{\text{MS}}(\mathbf{w}) = \mathbb{E}_{s\sim p^\star(s)}\Big[\nabla_{\mathbf{w}}\big(v_\pi(s) - \hat{v}(s;\mathbf{w})\big)^2\Big].$$

Then, we apply the chain rule:

$$\nabla_{\mathbf{w}} e_{\text{MS}}(\mathbf{w}) = \mathbb{E}_{s\sim p^\star(s)}[2(v_\pi(s) - \hat{v}(s;\mathbf{w}))(-\nabla_{\mathbf{w}}\hat{v}(s;\mathbf{w}))].$$

Introducing $-\frac{1}{2}$ on both sides to eliminate the 2:

$$\nabla_{\mathbf{w}} e_{\text{MS}}(\mathbf{w}) = -\mathbb{E}_{s\sim p^\star(s)}[(v_\pi(s) - \hat{v}(s;\mathbf{w}))\nabla_{\mathbf{w}}\hat{v}(s;\mathbf{w})].$$

Since $\hat{v}(s;\mathbf{w}) = \mathbf{w}^\top\mathbf{f}(s)$, we have $\nabla_{\mathbf{w}}\hat{v}(s;\mathbf{w}) = \mathbf{f}(s)$:

$$\nabla_{\mathbf{w}} e_{\text{MS}}(\mathbf{w}) = -\mathbb{E}_{s\sim p^\star(s)}\big[\big(v_\pi(s) - \mathbf{w}^\top\mathbf{f}(s)\big)\mathbf{f}(s)\big].$$

In RL setting, compared with Supervised Learning (SL):

1. There are no training data, so $v(s)$ is unknown. Thus, we need to approximate this with Temporal Difference (TD), Monte Carlo (MC), etc.

2. Data isn't i.i.d., but Markovian, and collected by the agent itself.

To address (1), we approximate $v(s)$ by:

$$v(s) \approx \mathbb{E}\left[\underbrace{(r(S,A) - \hat{g} + \hat{v}(s';\mathbf{w})}_{\text{approximation of } v(s) \text{ based on BEE}} - \hat{v}(s;\mathbf{w}))\mathbf{f}(s)\right],$$

where $v(s)$ is approximated based on the Bellman Error Equation (BEE). This involves the approximation of $\hat{v}(s;\mathbf{w})$, hence it's bootstrapping (using an estimation of a value to estimate another one).

And because it's an expectation, it's sampling-friendly, i.e., using a sample of (s,a,r',s') to get a sampling approximation:

$$\approx (r(s,a) - \hat{g} + \hat{v}(s';\mathbf{w}) - \hat{v}(s;\mathbf{w}))\mathbf{f}(s).$$

6 Approximate Policy Iteration with a Parametric Approach

High-level view of *approximate policy iteration* (API) can be viewed as a specialized case of *generalized policy iteration* (GPI) in Reinforcement Learning (RL). There are two main components:

1. **Approximate Policy Evaluation (Approx ValIter):** This stage aims to evaluate the value function V^π or Q^π (depending on whether we adopt a value-based or action-value-based approach) in an *approximate* manner.

- If a parametric approach is used, we define a value function $\hat{V}(s; w)$ (or $\hat{Q}(s, a; w)$) with parameter $w \in \mathbb{R}^k$.
- The goal is to minimize the *error* between $\hat{V}(s; w)$ and the true value, often through a mean-squared error or a temporal difference error.

2. **Approximate Policy Improvement (Approx PolIter):** Once we have a sufficiently accurate approximation of the value function, we perform *policy improvement*—that is, we update the policy π (potentially also parameterized).
   - For example, in a *policy gradient* method, the policy $\pi(a \mid s; w)$ itself can be a parametric function of w.
   - In some scenarios, we might select an action a that maximizes $\hat{Q}(s, a; w)$ to yield a deterministic policy.

Parametric Approach.

- We assume there is a parameter vector $w \in \mathbb{R}^k$ that defines the form of the value function or policy. For instance, if we use a linear function approximation:

$$\hat{V}(s; w) = \phi(s)^\top w,$$

  where $\phi(s) \in \mathbb{R}^k$ is a feature vector for state s.
- In some cases (e.g., a tabular setting), the dimensionality k equals the number of states, and $\phi(s)$ becomes a one-hot vector.

Parameter Optimization via SGD.

- We define an error function $e(w)$—for example, the mean-squared error between $\hat{V}(s; w)$ and the target.
- We seek w^* that minimizes $e(w)$:

$$w^* = \arg\min_w \; e(w).$$

- To achieve this, we often use *stochastic gradient descent* (SGD) or a related variant (e.g., Adam, RMSProp), updating w iteratively:

$$w \leftarrow w - \alpha \, \nabla_w e(w),$$

  where α is the learning rate.

Iterative Process in RL.

- In RL, approximate evaluation and approximate improvement typically occur iteratively while the agent gathers new data by interacting with the environment.
- Each time the value function is updated, the policy can be improved, and the new policy is then evaluated again, and so forth. This is the essence of *generalized policy iteration* (GPI).

7 Projected Bellman Error

This section explains several key ideas in Reinforcement Learning and Approximate Dynamic Programming:

In approximate dynamic programming or reinforcement learning, v may lie in a restricted function space (e.g., a parametric class of value functions). Hence, we cannot always solve

$$(I - P')\, v = r - g\, \mathbf{1}$$

exactly. Instead, we use a *projection* operator $\mathbb{P}$ that projects any function onto our approximation space. Thus, the *projected Bellman error* (PBE) is defined as:

$$\text{PBE}(v) \;=\; \mathbb{P}\Big[(r - g\,\mathbf{1}) + P'\,v\Big] \;-\; v.$$

This quantity indicates how close the projected Bellman update $\mathbb{P}[\mathbb{B}v]$ is to v, where the Bellman operator is given by

$$\mathbb{B}v \;\equiv\; r - g\,\mathbf{1} \;+\; P'v.$$

Minimizing the PBE is a common strategy in methods such as LSTD, where we seek a parameterized value function $\hat{v}(w)$ that is as close as possible (in a projected sense) to the Bellman update.

7.0.1 Norm-Based View of PBE

A common way to quantify the PBE is via a weighted ℓ_2-norm (often using the stationary distribution $p^*(s)$ as weights). Let

$$\Delta v(s) \;=\; \hat{v}(w)(s) \;-\; \mathbb{P}\big[\mathbb{B}\,\hat{v}(w)\big](s).$$

Then, the PBE can be measured as:

$$e_{\mathrm{PB}} \;=\; \big\|\Delta v\big\|_{p^*}^2 \;=\; \sum_s p^*(s)\,\big[\Delta v(s)\big]^2 \;=\; \mathbb{E}_{s\sim p^*}\big[\Delta v(s)^2\big].$$

When $e_{\mathrm{PB}} = 0$, it implies that $\hat{v}(w)$ is a fixed point of the projected Bellman operator, i.e., $\hat{v}(w) = \mathbb{P}[\mathbb{B}\,\hat{v}(w)]$.

7.1 Why It Does Not Involve the True Value

A key point is that we do *not* need the *true* value function v^* to measure or minimize the Bellman error equation (or the projected Bellman error). Instead, we only need:

- Observations of $(s, r(s), s')$ transitions (in model-free settings), or
- A model for the reward and transition probabilities (in model-based settings).

From these, we can construct the Bellman operator $\mathbb{B}$ (or an empirical version of it) and compare $\mathbb{B}v$ to v. This approach is stable because it relies on the *self-consistency* property of the Bellman equation rather than on direct knowledge of v^*.

7.2 Temporal Difference Error Connection

Methods like *Temporal Difference (TD) learning* effectively estimate the Bellman error equation incrementally. A TD update can be viewed as trying to drive the quantity

$$\delta_t = r_t - g + v(s_{t+1}) - v(s_t)$$

to zero over time, which corresponds to making v satisfy the Bellman equation in expectation.

7.3 Geometric Interpretation in Parameter Space

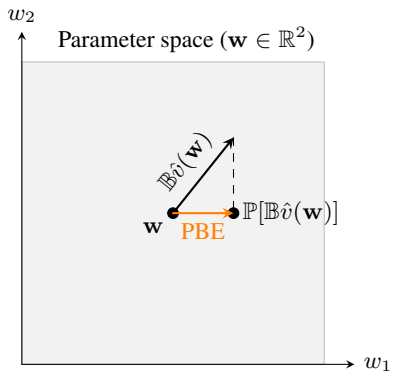

Figure 3: Conceptual illustration of the geometric interpretation in parameter space. The light gray square represents the 2D parameter space where $\mathbf{w}$ resides. Applying the Bellman operator $\mathbb{B}$ to $\hat{v}(\mathbf{w})$ can yield a value function outside the representable space (point $\mathbb{B}\hat{v}(\mathbf{w})$). The projection operator $\mathbb{P}$ then maps it back into the parameter space (point $\mathbb{P}[\mathbb{B}\hat{v}(\mathbf{w})]$). The orange arrow (labeled PBE) shows the difference between the current value function and its projected Bellman update.

Explanation of the Diagram:

- **Parameter Space:** The light gray square represents the 2D parameter space, with axes w_1 and w_2. Every point in this space corresponds to a set of parameters $\mathbf{w}$ defining the approximate value function $\hat{v}(\mathbf{w})$.

- **Current Parameter Vector:** The point labeled $\mathbf{w}$ (located at $(1, 1)$ in this example) is our current estimate in the parameter space.
- **Bellman Operator Application:** The arrow from $\mathbf{w}$ to the point labeled $\mathbb{B}\hat{v}(\mathbf{w})$ represents the application of the Bellman operator. This update may move the value function outside the representable parameter space.
- **Projection Operator:** The dashed line and the point $\mathbb{P}[\mathbb{B}\hat{v}(\mathbf{w})]$ indicate that we use a projection operator $\mathbb{P}$ to map the Bellman update back into our parameter space.
- **Projected Bellman Error (PBE):** The orange arrow from $\mathbf{w}$ to $\mathbb{P}[\mathbb{B}\hat{v}(\mathbf{w})]$ represents the Projected Bellman Error. Minimizing this error is key to ensuring that our approximate value function is as close as possible to being a fixed point of the projected Bellman operator.

7.4 Projection Operator $\mathbb{P}$

Suppose we have:

$$\text{feature matrix: } F \in \mathbb{R}^{|S|\times k}, \quad \text{diagonal weighting: } D_{p^*} \in \mathbb{R}^{|S|\times|S|}.$$

Here, $|S|$ is the number of states (or samples), and each row of F is a feature vector $f(s)^T$ for state s. The matrix D_{p^*} is diagonal with entries $p^*(s)$, representing a stationary distribution or weighting over states.

We wish to define a *projection* operator

$$\mathbb{P} : \mathbb{R}^{|S|} \;\to\; \text{Col}(F),$$

where $\text{Col}(F)$ is the column space spanned by F. The projection is taken under the ℓ_2-norm weighted by D_{p^*}, namely

$$\|x\|^2_{D_{p^*}} \;=\; x^T\, D_{p^*}\, x.$$

Formally, for any vector $v \in \mathbb{R}^{|S|}$, we define

$$\mathbb{P}(v) \;=\; \arg\min_{x\in\text{Col}(F)} \; \|v - x\|^2_{D_{p^*}}.$$

Since $x \in \text{Col}(F)$ can be written as $x = Fw$ for some $w \in \mathbb{R}^k$, the problem becomes:

$$\mathbb{P}(v) \;=\; \arg\min_{Fw} \; \|v - Fw\|^2_{D_{p^*}}.$$

We can solve for w by expanding and taking derivatives.

Step-by-Step Derivation:

1. **Rewrite the objective:**

$$\|v - Fw\|^2_{D_{p^*}} \;=\; (v - Fw)^T\, D_{p^*}\, (v - Fw).$$

2. **Take the gradient with respect to w:**

$$\frac{\partial}{\partial w}\big[(v - Fw)^T\, D_{p^*}\, (v - Fw)\big] \;=\; -2\, F^T\, D_{p^*}\, (v - Fw).$$

   Setting this to zero for optimality yields:

$$F^T\, D_{p^*}\, (v - Fw) \;=\; 0 \quad\Longrightarrow\quad F^T\, D_{p^*}\, F\, w \;=\; F^T\, D_{p^*}\, v.$$

3. **Solve for w:**

$$w^* \;=\; \big(F^T\, D_{p^*}\, F\big)^{-1}\, F^T\, D_{p^*}\, v.$$

4. **Hence, the projection of v:**

$$\mathbb{P}(v) \;=\; F\, w^* \;=\; F\,\big(F^T\, D_{p^*}\, F\big)^{-1}\, F^T\, D_{p^*}\, v.$$

Therefore, the projection operator $\mathbb{P}$ onto the column space of F under the D_{p^*}-weighted norm is

$$\boxed{\mathbb{P} \;=\; F\,\big(F^T\, D_{p^*}\, F\big)^{-1}\, F^T\, D_{p^*}.}$$

In other words, for any vector v, $\mathbb{P}(v)$ is given by the above formula.

7.5 Deriving w^* to Minimize the Projected Bellman Error

Next, consider a *linear* approximate value function:

$$\hat{v}(w) \;=\; F\,w.$$

The *Projected Bellman Error* (PBE) can be defined (in squared norm) as

$$e_{\mathrm{PB}}(w) \;=\; \big\|\hat{v}(w) \;-\; \mathbb{P}\big[\mathbb{B}\,\hat{v}(w)\big]\big\|^2_{D_{p^*}},$$

where $\mathbb{B}$ is the Bellman operator, e.g.,

$$\mathbb{B}\,\hat{v}(w) \;=\; r \;-\; g\,\mathbf{1} \;+\; P'\,\big(F\,w\big).$$

We want:

$$w^* \;=\; \arg\min_{w}\; e_{\mathrm{PB}}(w).$$

Outline of the Solution:

- **Apply $\mathbb{P}$:**
$$\mathbb{P}\big[\mathbb{B}\,\hat{v}(w)\big] \;=\; F\,(F^T D_{p^*} F)^{-1} F^T\,D_{p^*}\,\big(r - g\,\mathbf{1} + P'\,F\,w\big).$$
- **Compute the difference:**
$$\hat{v}(w) \;-\; \mathbb{P}\big[\mathbb{B}\,\hat{v}(w)\big] \;=\; F\,w \;-\; F\,(F^T D_{p^*} F)^{-1} F^T\,D_{p^*}\,\big(r - g\,\mathbf{1} + P'\,F\,w\big).$$
- **Square under D_{p^*}:**
$$e_{\mathrm{PB}}(w) \;=\; \big\|F\,w \;-\; F\,(F^T D_{p^*} F)^{-1} F^T\,D_{p^*}\,\big(r - g\,\mathbf{1} + P'\,F\,w\big)\big\|^2_{D_{p^*}}.$$
One can then take the gradient with respect to w and set it to zero. Under suitable assumptions (like invertibility of $(F^T D_{p^*}(I - P')F)$), the resulting w^* often matches the well-known LSTD solution:
$$w^* \;=\; \big(F^T\,D_{p^*}\,\big(I - P'\big)\,F\big)^{-1} F^T\,D_{p^*}\,\big(r - g\,\mathbf{1}\big).$$

Interpretation.

- *Projection*: We only need to operate in the subspace spanned by F. Instead of trying to match the Bellman operator $\mathbb{B}\hat{v}(w)$ exactly, we *project* it back into $\mathrm{Col}(F)$.
- *Weighted Norm*: The diagonal matrix D_{p^*} imposes a weighting, typically corresponding to a stationary distribution over states. States with higher probability mass get higher influence in the error measure.
- *Minimizing PBE*: Solving $\arg\min_w e_{\mathrm{PB}}(w)$ yields w^* that best satisfies the Bellman equation in the projected sense. This is precisely the principle behind LSTD and related least-squares policy/value evaluation methods.

Final Summary. The key formulas are:

$$\boxed{\mathbb{P}(v) = F\,\big(F^T\,D_{p^*}\,F\big)^{-1} F^T\,D_{p^*}\,v, \quad w^* = \arg\min_{w} \big\|\,Fw \;-\; \mathbb{P}\big[\mathbb{B}(Fw)\big]\,\big\|^2_{D_{p^*}}.}$$

They show how the projection operator arises from a simple least-squares argument and how, in turn, one can derive the optimal parameter vector w^* by minimizing the Projected Bellman Error.

7.6 Taking the Gradient and Finding w^*

When we want to find a parameter vector w^* that minimizes a function $e_{\mathrm{PB}}(w)$ (such as the Projected Bellman Error in a linear approximation setting), we typically follow these steps:

1. **Compute the gradient:**
$$\nabla_w\, e_{\mathrm{PB}}(w).$$
2. **Set the gradient to zero:**
$$\nabla_w\, e_{\mathrm{PB}}(w) \;=\; 0.$$
3. **Solve for w:** The solution $\widehat{w}$ that satisfies $\nabla_w\, e_{\mathrm{PB}}(\widehat{w}) = 0$ is a *critical point*. Under suitable convexity conditions (or in least-squares problems), this critical point is a global minimum, which we denote by w^*.

Intuitively, at the minimum of a differentiable function, the slope (or gradient) is zero. This is visualized in a simple 1D schematic in Figure 4, where the horizontal axis represents the parameter w, and the vertical axis represents the error function e.

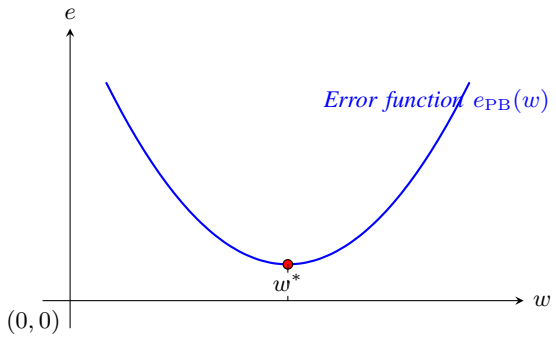

Figure 4: A conceptual 1D illustration of minimizing an error function e by setting its gradient to zero. At the minimum w^*, the slope (or derivative) is zero.

Key Takeaways:

- Setting $\nabla_w e_{\mathrm{PB}}(w) = 0$ is the standard procedure to find a minimum in differentiable optimization.
- In the context of LSTD or other least-squares methods, solving this equation often yields a *closed-form* expression for w^*.
- Graphically, a zero gradient corresponds to the point where the error curve has zero slope (the bottom of a parabola in a 1D schematic).

8 Derivation of the Matrix X in Dynamic Programming Settings

In many Dynamic Programming (DP) and Reinforcement Learning contexts, we encounter a matrix X that helps us solve linear systems of the form

$$X\, w = y,$$

leading to the solution

$$w^* = X^{-1}\, y.$$

This section explains in detail how X can be derived, why it takes its particular form, and how it relates to the DP framework.

8.1 Definitions and Notation

- $p^*(s)$ denotes a stationary distribution (or weighting) over the states s.
- $p(s' \mid s)$ denotes the transition probability from state s to state s'.
- $f(s)$ is a feature vector associated with state s.
- We aim to define a matrix X that captures the transitions and features, which will be used in evaluating or approximating a value function in DP.

8.2 Direct Definition of X

One way to define X is by summing over all states s and possible next states s':

$$X = \sum_s p^*(s) \sum_{s'} p(s' \mid s) \Big[f(s)\, f(s) - f(s)\, f(s') \Big]^T.$$

Here, the notation $\Big[\dots \Big]^T$ indicates that the expression inside is transposed to match the desired dimensions of X.

8.3 Summation Simplification

We can rewrite the difference inside the sum as:

$$f(s)\, f(s) - f(s)\, f(s') = f(s) \big[f(s) - f(s') \big].$$

Thus, the matrix X becomes:

$$X = \sum_s p^*(s) \sum_{s'} p(s' \mid s) f(s) \left[f(s) - f(s') \right]^T.$$

Notice that $\sum_{s'} p(s' \mid s) f(s')$ represents the expected next feature vector given the current state s.

8.4 Matrix Representation

If we collect the features $f(s)$ for all states into a matrix $\mathbf{F}$, and define:

- $\mathbf{D}_{p^*}$ as a diagonal matrix with $p^*(s)$ on its diagonal,
- $\mathbf{P}$ as the transition matrix with entries $\mathbf{P}[s, s'] = p(s' \mid s)$,
- $\mathbf{I}$ as the identity matrix,

then we can express X succinctly as:

$$X = \mathbf{F}^T \mathbf{D}_{p^*} (\mathbf{I} - \mathbf{P}) \mathbf{F}.$$

This representation reflects:

- The weighting of states by $\mathbf{D}_{p^*}$,
- The difference between the feature vector $f(s)$ and the expected next feature $\sum_{s'} p(s' \mid s) f(s')$, captured by $(\mathbf{I} - \mathbf{P})\mathbf{F}$,
- The transformation into the final matrix form by $\mathbf{F}^T$.

8.5 Implication for w^*

In many DP or policy evaluation scenarios, we have a linear equation:

$$X\, w = y,$$

where y is a vector derived from rewards or temporal-difference terms. If X is non-singular, the solution is given by:

$$w^* = X^{-1}\, y.$$

This weight vector w^* typically represents the parameters of an approximate value function in reinforcement learning.

8.6 Conclusion

To summarize, the matrix X is derived as follows:

$$\boxed{X = \sum_s p^*(s) \sum_{s'} p(s' \mid s) \Big[f(s) f(s) - f(s) f(s') \Big]^T = \mathbf{F}^T \mathbf{D}_{p^*} (\mathbf{I} - \mathbf{P}) \mathbf{F}.}$$

This formulation is instrumental in both theoretical analyses and practical algorithms in approximate dynamic programming.

9 Sample-Based Approximation of X and the LSTD Estimator

In practice, instead of computing

$$X = \sum_s p^*(s) \sum_{s'} p(s' \mid s) \big[f(s)\, f(s) - f(s)\, f(s') \big]^T$$

directly (which may be infeasible when the state space is large), we often approximate X using a finite sample of n transitions $\{(s_i, s_i')\}_{i=1}^n$. The sample-based approximation, denoted by $\widehat{X}$, is typically given by:

$$\widehat{X} \;=\; \frac{1}{n} \sum_{i=1}^{n} f(s_i) \left[f(s_i) - f(s_i') \right]^T,$$

where:

- n is the number of sampled transitions.
- $f(s_i)$ is the feature vector corresponding to the i-th state s_i.
- s_i' is the next state observed after s_i.

Similarly, we can form an unbiased sample-based estimate of the vector y, denoted by $\widehat{y}$. For instance, if y represents expected returns or temporal-difference terms, we might write:

$$\widehat{y} \;=\; \frac{1}{n}\sum_{i=1}^{n}\big[r(s_i) - g\big]\; f(s_i),$$

where $r(s_i)$ is the (immediate) reward observed at s_i, and g could be a discount factor term or baseline depending on the exact definition. The main idea is that both $\widehat{X}$ and $\widehat{y}$ are constructed as simple averages over the samples, making them unbiased estimators of X and y, respectively, under suitable assumptions.

9.1 LSTD Solution

Once we have $\widehat{X}$ and $\widehat{y}$, the Least-Squares Temporal Difference (LSTD) method provides an estimate of the parameter vector w via:

$$\widehat{w} \;=\; \widehat{X}^{-1}\,\widehat{y}.$$

This mirrors the exact solution $w^* = X^{-1}y$ but uses sample-based estimates instead of population quantities. In practice, to avoid numerical instability when n is small or $\widehat{X}$ is close to singular, one often adds a small regularization term $\xi\mathbf{I}$ to the matrix, yielding:

$$\widehat{w} \;=\; \big(\widehat{X} + \xi\,\mathbf{I}\big)^{-1}\,\widehat{y}.$$

This technique is sometimes referred to as L2-regularization or ridge regression in the context of LSTD.

9.2 Discussion and Unbiasedness

By construction, if the samples (s_i, s_i') are drawn from the stationary distribution $p^*(s)$ and transition model $p(s' \mid s)$, the expectation of $\widehat{X}$ coincides with X. Similarly, $\widehat{y}$ is unbiased for y. Hence, as $n \to \infty$, we expect $\widehat{w}$ to converge to the true parameter w^*, provided that X is nonsingular and other regularity conditions hold.

9.3 Defining y and the Practical Motivation for LSTD

Recall from the previous discussion that the matrix X in the DP setting can be written as:

$$X = \mathbf{F}^T\,\mathbf{D}_{p^*}\,\big(\mathbf{I} - \mathbf{P}\big)\,\mathbf{F}.$$

An analogous expression for the vector y is often given by:

$$y = \sum_s p^*(s)\,\big[r(s) - g\big]\; f(s) = \mathbf{F}^T\,\mathbf{D}_{p^*}\,\big(r - g\,\mathbf{1}\big),$$

where:

- $r(s)$ is the (immediate) reward function for state s,
- g is a constant (e.g., a baseline or a term related to discounting),
- $\mathbf{1}$ is a vector of all ones,
- $\mathbf{F}$ is the matrix of feature vectors, and
- $\mathbf{D}_{p^*}$ is a diagonal matrix with entries $p^*(s)$.

Why not directly compute $X^{-1}y$?
In practice, especially in RL, we do *not* know:

- The exact stationary distribution $p^*(s)$,
- The full transition model $p(s' \mid s)$,
- The exact reward function $r(s)$ for all states (it might be known locally but not necessarily globally).

Moreover, even if we did know them, computing large matrices X and y might be intractable for high-dimensional state spaces. Thus, we cannot simply form X and y and invert X to solve

$$X\, w^* = y.$$

Instead, we rely on *sample-based* methods, which is where **Least-Squares Temporal Difference** (**LSTD**) enters.

9.4 LSTD Recap

The LSTD approach addresses these limitations by replacing X and y with sample-based estimates $\widehat{X}$ and $\widehat{y}$:

$$\widehat{X} = \frac{1}{n}\sum_{i=1}^{n} f(s_i)\left[f(s_i) - f(s_i')\right]^T, \quad \widehat{y} = \frac{1}{n}\sum_{i=1}^{n}\left[r(s_i) - g\right]\, f(s_i).$$

Here:

- (s_i, s_i') are sampled transitions from an environment or simulator,
- $r(s_i)$ is the observed reward at state s_i,
- g is a baseline or discount-related term, and
- n is the total number of samples.

Then, the LSTD solution approximates w^* by:

$$\widehat{w} = \widehat{X}^{-1}\,\widehat{y},$$

assuming $\widehat{X}$ is non-singular (or using a regularized inverse such as $(\widehat{X} + \xi I)^{-1}$). In this way, LSTD sidesteps the need for full knowledge of $p^*(s)$ and $p(s' \mid s)$ by leveraging empirical data. As $n \to \infty$, the law of large numbers implies that $\widehat{X}$ and $\widehat{y}$ converge (in expectation) to X and y, respectively, making $\widehat{w}$ converge to w^* under appropriate conditions.

10 Practical Considerations in Reinforcement Learning

No Clear Separation Between Training and Testing. In many Reinforcement Learning (RL) scenarios, the agent continues to interact with the *same* environment while it is learning. Consequently, there is no strict boundary between a "training phase" and a "testing phase" as one might see in supervised learning. Instead, the agent collects data (state transitions, rewards) *while* it is acting. If we do want to distinguish training from testing in RL, we may do so by:

- **Stopping Learning:** The agent may freeze its policy or its value function parameters after a certain point and then evaluate performance.
- **Using a Separate Environment:** If possible, the agent can copy or simulate the environment for training and then switch to a real (or different) environment for testing. However, this is not always feasible or realistic.

Challenges in Non-Stationary Environments. A more challenging scenario arises when the environment itself changes over time (i.e., it is *non-stationary*). In such cases:

- The transition probabilities $p(s' \mid s, a)$ may shift or drift over time, invalidating assumptions that were true at the beginning of training.
- The policy or value function must adapt continuously, often requiring methods that can track these changes (e.g., by placing higher weight on more recent samples).
- Simple sample-based methods, like LSTD, may need modifications to account for non-stationary data, such as resetting estimates, using a sliding window of samples, or incorporating explicit forgetting factors.

Different but Similar Environments. Even when the environment changes, it might still share certain similarities (e.g., common dynamics, partial overlap in states, or similar reward structures). In these cases, transfer learning or meta-RL techniques can help the agent quickly adapt to the modified environment using knowledge acquired previously.

Tabular Setting as a Special Case of Function Approximation. In a *tabular* RL setting, each state s has a corresponding entry in a table for its value $V(s)$ or action-value $Q(s,a)$. This can be viewed as a special case of linear function approximation where:

$$f(s) \;=\; e_s,$$

and e_s is a *one-hot* (or indicator) vector with 1 in the position corresponding to state s and 0 elsewhere. Therefore:

- The dimension of $f(s)$ equals the total number of states in the environment.
- The matrix X and vector y reduce to their tabular counterparts, often seen in standard DP or TD methods.
- LSTD in the tabular case is essentially the same as solving a system of linear equations to find exact state-value estimates (assuming we sample enough transitions to build these equations).

Summary.

- LSTD (and related algorithms) aim to solve the TD fixed-point equation using sample-based estimates of X and y.
- In RL, we typically do not have a separate, static training set vs. testing set. Instead, the agent continuously learns in the environment.
- Non-stationary or evolving environments introduce additional challenges that require adaptive or robust methods.
- The tabular setting can be seen as a simple, exact form of function approximation where the feature vector is an indicator for each state.

11 Tabular Representation as One-Hot Encoding

As noted earlier, the tabular setting can be interpreted as a special case of linear function approximation where the feature vector $f(s)$ is *one-hot* (or an indicator vector). Suppose the state space is finite, with $|\mathcal{S}|$ distinct states. Then we can define:

$$f(s) \;=\; e_s \;\in\; \mathbb{R}^{|\mathcal{S}|},$$

where e_s is a vector of length $|\mathcal{S}|$ with:

$$(e_s)_i = \begin{cases} 1, & \text{if } i = s, \\ 0, & \text{otherwise.} \end{cases}$$

This representation implies the following:

- The dimensionality of $f(s)$ is exactly $|\mathcal{S}|$.
- Consequently, the parameter vector w also has dimension $|\mathcal{S}|$. Hence, each state s corresponds to one component w_s in the parameter vector.
- The approximate value function (for a state-value function) is then:

  $$\hat{V}(s) = f(s)^T w = e_s^T w = w_s,$$

  which matches the classic *tabular* representation where we store a single scalar value $V(s)$ for each state s.

Example with Small State Space. Consider a small environment with five states $\{s^0, s^1, s^2, s^3, s^4\}$. The one-hot feature vectors $f(s^0), f(s^1), \ldots, f(s^4)$ might look like:

$$f(s^0) = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad f(s^1) = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \quad \ldots, \quad f(s^4) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}.$$

Then the parameter vector w is

$$w = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix},$$

so that $\hat{V}(s^i) = w_i$ for $i = 0, 1, \ldots, 4$.

Implication for LSTD and Other Methods. In this tabular context:

- The matrix X and vector y (or their sample-based approximations $\widehat{X}$, $\widehat{y}$) become $|\mathcal{S}| \times |\mathcal{S}|$ and $|\mathcal{S}| \times 1$, respectively.
- Solving the linear system $X\, w = y$ is equivalent to solving for the exact value of each state (given enough samples and under appropriate conditions).
- Methods like LSTD or TD(0) reduce to their well-known tabular forms, since each state is treated independently in the feature space (though transitions couple them together through the Bellman equation).

Thus, the one-hot encoding neatly bridges the gap between the tabular approach and general linear function approximation, showing that the tabular method is simply the special case where each state is its own feature dimension.

12 Citations and References