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.

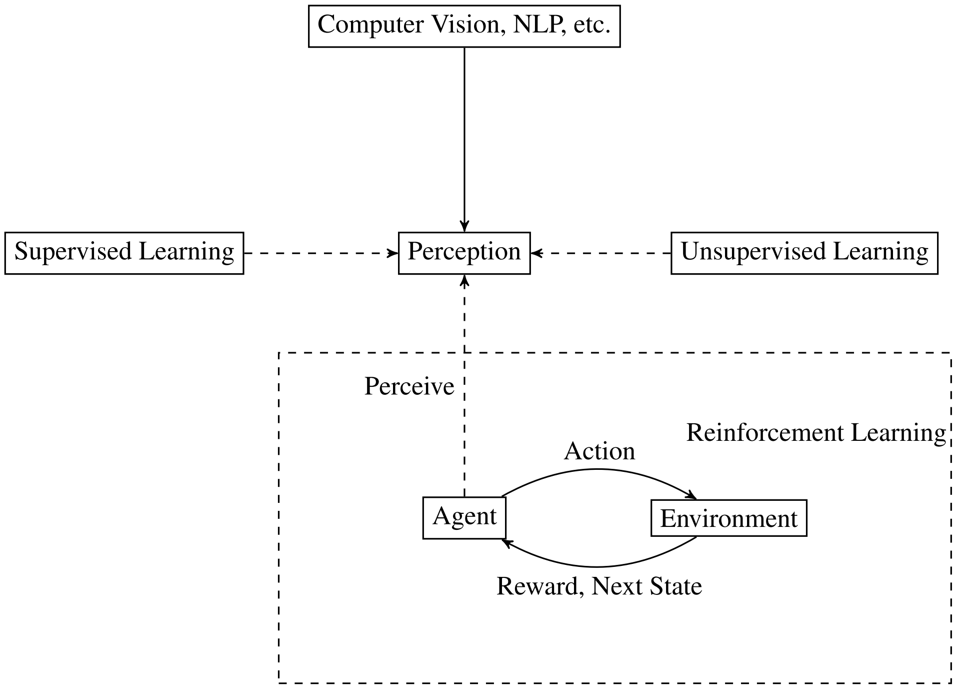

Figure 1: Reinforcement Learning Scheme with Perception

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).

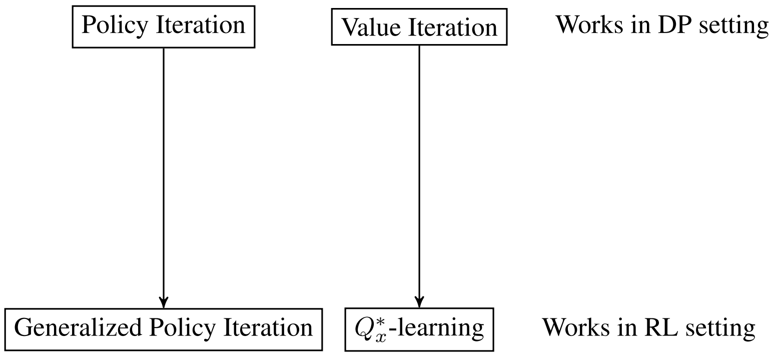

Figure 2: Major RL Schemes To Find Optimal Policy

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:

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

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

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:

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

We move the gradient inside the expectation:

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

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 $\nabla_{\mathbf{w}}\hat{v}(s;\mathbf{w}) = \mathbf{f}(s)$:

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

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 Citations and References