Function Approximation in Reinforcement Learning

Santhosh Kumar G

October 22, 2025

Abstract

Function approximation is a fundamental concept in Reinforcement Learning (RL) used to estimate complex functions, such as value functions or policies, when the state space is too large for tabular methods. Function approximation is an instance of *supervised learning*, so it is easy to apply any such machine learning methods to RL. Here we summarize the key methods including value prediction, gradient descent, linear function approximation, control algorithms, fitted iterative methods, and policy gradient techniques.

1 Function Approximation: Value Prediction

In simple RL environments that we have seen so far, an agent uses a tabular method. This means agent stores the estimated value for every single state in a *lookup table*. This will be troublesome when the state-space is too large or continuous; in many practical cases it is so. If the environment is *continuous* (like the position of a robot arm: infinite states) or extremely large (like a complex video game: trillions of states), a table is impossible to store. Also, the tabular methods treat every state independently. If the agent visits a new state that is very similar to one it has seen before, it has to learn the value from scratch. Function approximation solves this by replacing the massive lookup table with a parameterized function, typically a neural network (though linear models or decision trees can also be used). In general, instead of storing $V(s)$ for every state s, the agent learns a set of parameters (weights) $\mathbf{w}$ that can approximate the true value function: $\hat{V}(s, w) \approx V(s)$

In RL, function approximation estimates the **Value Function** ($V(s)$ or $Q(s, a)$) using a parameterized function, $\hat{v}(s, \mathbf{w})$ or $\hat{q}(s, a, \mathbf{w})$, instead of a large lookup table. For example, $\hat{v}$ might be the function computed by an artificial neural network, with $\mathbf{w}$ the vector of connection weights. By adjusting the weights, any of a wide range of different functions $\hat{v}$ can be implemented by the network. Typically, the number of parameters n (the number of components of $\mathbf{w}$) is much less than the number of states, and changing one parameter changes the estimated value of many states. Consequently, when a single state is backed up, the change generalizes from that state to affect the values of many other states.

1.1 Goal and Error Measure

The objective is to find the parameter vector $\mathbf{w}$ that minimizes the error between the estimated value and a target value (e.g., Monte Carlo return or TD target). The standard error measure is the **Mean Squared Error (MSE)**:

$$L(\mathbf{w}) = E[(Target - \hat{v}(s, \mathbf{w}))^2]$$

2 Gradient Descent Methods

Gradient Descent (GD) is the primary optimization technique, iteratively adjusting the parameters $\mathbf{w}$ in the direction opposite to the gradient of the loss function $L(\mathbf{w})$.

2.1 Stochastic Gradient Descent (SGD) for RL

Since the full expectation of the error is unknown, updates are performed using a sample (the TD error) at time t:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \nabla L(\mathbf{w}_t)$$

For TD learning, the update simplifies to:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \underbrace{(Target_t - \hat{v}(S_t, \mathbf{w}_t))}_{\text{TD error}} \nabla \hat{v}(S_t, \mathbf{w}_t)$$

where α is the learning rate.

3 Linear Function Approximation

The simplest and most robust form of function approximation, where the estimated value is a linear combination of a feature vector $\mathbf{x}(s)$ and the weight vector $\mathbf{w}$.

3.1 Formulation and Gradient

The value estimate is:

$$\hat{v}(s, \mathbf{w}) = \mathbf{w}^T \mathbf{x}(s) = \sum_i w_i x_i(s)$$

The gradient with respect to $\mathbf{w}$ is simply the feature vector:

$$\nabla \hat{v}(s, \mathbf{w}) = \mathbf{x}(s)$$

3.2 TD(0) Update Rule

The linear TD(0) update rule is:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t) \right] \mathbf{x}(S_t)$$

4 Control Algorithms

Control algorithms aim to find an optimal policy π^* by approximating the action-value function, $\hat{q}(s, a, \mathbf{w})$.

4.1 Sarsa with Function Approximation (On-Policy)

Uses the value of the next action A_{t+1} chosen by the policy π:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t) \right] \nabla \hat{q}(S_t, A_t, \mathbf{w}_t)$$

4.2 Q-Learning with Function Approximation (Off-Policy)

Uses the maximum value (greedy action) at the next state:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[R_{t+1} + \gamma \max_{a'} \hat{q}(S_{t+1}, a', \mathbf{w}_t) - \hat{q}(S_t, A_t, \mathbf{w}_t) \right] \nabla \hat{q}(S_t, A_t, \mathbf{w}_t)$$

Note: The combination of off-policy learning, bootstrapping, and non-linear function approximation is known as the **Deadly Triad** and can lead to divergence.

5 Fitted Iterative Methods (Batch Methods)

These methods reframe the problem as supervised learning, using a fixed dataset $\mathcal{D}$ (e.g., from a replay buffer) to update the approximator.

5.1 Fitted Q-Iteration (FQI)

1. Collect a dataset of transitions $\mathcal{D} = \{(s_i, a_i, r_i, s'_i)\}$. 2. Iteratively calculate target values y_i using the current Q-function estimate:

$$y_i = r_i + \gamma \max_{a'} \hat{q}(s'_i, a', \mathbf{w}_k)$$

3. Train a new Q-function $\hat{q}(\cdot, \cdot, \mathbf{w}_{k+1})$ to regress onto the targets y_i over the entire dataset $\mathcal{D}$. **Example:** Deep Q-Networks (DQN) use this batch-style approach with experience replay and a target network.

6 Policy Gradient Methods

Policy Gradient (PG) methods directly parameterize the policy $\pi(a|s, \theta)$ and use gradient ascent to find the parameters θ that maximize the expected return $J(\theta)$.

6.1 Performance Objective

The goal is to maximize the expected return:

$$J(\theta) = E[\text{Return}|\pi_\theta]$$

6.2 Policy Gradient Theorem

The gradient used for the update is proportional to:

$$\nabla J(\theta) \propto \sum_s d^\pi(s) \sum_a Q^\pi(s, a) \nabla \pi(a|s, \theta)$$

6.3 Key Algorithms

- **REINFORCE:** Uses the empirical return G_t (from Monte Carlo) as an estimate for $Q^\pi(s, a)$.
- **Actor-Critic Methods:** Use two separate approximators: an **Actor** (the policy $\pi(a|s, \theta)$) and a **Critic** (a value function $\hat{v}(s, \mathbf{w})$) to estimate the advantage $A(s, a)$ or action-value $Q(s, a)$, providing lower variance updates for the Actor.