Feature Construction for Linear Methods in Reinforcement Learning

When using **linear function approximation** in reinforcement learning, the value function is estimated as:

$$\hat{v}(s,w) = w^T x(s)$$

Here, $x(s)$is a **feature vector** representing the state. The effectiveness of a linear method **depends entirely on how you construct these features**, because linear methods can't represent non-linear relationships directly.

Goals of Feature Construction

A good feature set should:

- Capture the important structure of the environment.
- Balance **generalization** (learn from limited data) and **discrimination** (distinguish critical differences between states).
- Be compact and computationally efficient.

Common Feature Construction Techniques

1. Tabular Features (One-hot Encoding)

- Each state is uniquely represented.
- Feature vector: all zeros except a 1 at the index for state sss

Pros: No interference between states.
Cons: Doesn't generalize; impractical for large/continuous spaces.

2. Tile Coding (CMAC)

- Overlay multiple overlapping grids (tilings) on the state space.
- Each tiling provides a binary feature (1 if active).
- Final feature vector: binary vector from all tilings.

Pros: Local generalization; scalable.
Cons: Hand-designed; requires tuning tile widths and number of tilings.

3. Radial Basis Functions (RBFs)

- Feature: $x_i(s) = \exp\left(-\frac{\|s-c_i\|^2}{2\sigma^2}\right)$, where c_i is the center of the i^{th} basis function

Pros: Smooth generalization.
Cons: Still hand-designed; can become computationally expensive in high dimensions.

4. Polynomial Features

- Construct features like $x_1 = s, x_2 = s^2, x_3 = s_1 \cdot s_2$, etc.

Pros: Captures nonlinearities.
Cons: Can explode in size and be hard to interpret.

5. Fourier Basis

- Features: $\cos(\pi c^T s)$ for different frequency vectors ccc
- Inspired by Fourier series expansion.

Pros: Works well in continuous domains; effective in practice.
Cons: Choosing good frequency terms ccc is non-trivial.

6. Hand-Crafted Features

- Based on domain knowledge (e.g., distances, angles, obstacles).
- Combines discrete and continuous info.

Pros: Efficient and informative if designed well.
Cons: Doesn't scale or generalize easily to other tasks.

Neural Networks in Reinforcement Learning1. What Is a Neural Network?

A **neural network** is a **parameterized nonlinear function** composed of layers of simple computational units (neurons). Each neuron applies a **linear transformation** followed by a **nonlinear activation function**.

A simple 2-layer neural network approximates a function $f(s)$ as:

$$f(s) = \sigma_2(W_2 \cdot \sigma_1(W_1 \cdot s + b_1) + b_2)$$

Where:

- s: input (e.g. state)
- W_1, W_2: weight matrices
- b_1, b_2: biases
- σ_1, σ_2: nonlinear activation functions (e.g. ReLU, tanh)

This transforms raw input into complex, **nonlinear representations**.

2. Nonlinear Function Approximation

In RL, we often want to approximate functions like:

- **Value functions** $v_\pi(s)$
- **Action-value functions** $q_\pi(s, a)$
- **Policies** $\pi(a|s)$

A **linear approximator** assumes:

$$\hat{v}(s) = w^T x(s)$$

But this can't capture **nonlinear patterns** or **complex relationships** in the state space.

So we replace it with a **nonlinear function approximator** (e.g., a neural net):

$$\hat{v}(s, \theta) = NN(s, \theta)$$

Where:

- NN is a neural network
- θ are its parameters (weights and biases)

This allows us to **learn features automatically** instead of hand-crafting them.

---

3. Deep Neural Networks (DNNs)

A **deep neural network** has **multiple hidden layers**, enabling it to:

- Learn **hierarchical** representations
- Extract **features at different levels of abstraction**
- Handle **high-dimensional and structured inputs** (e.g. images, sensor data)

Example:

A DNN might learn:

- Layer 1: edges or colors
- Layer 2: shapes or object parts
- Layer 3: full objects or patterns

In RL:

- DNNs are used in **Deep Q-Networks (DQN)** to approximate $Q(s, a)$
- In **Actor-Critic** methods to learn both the **policy** and the **value function**
- In **model-based RL**, to learn transition models

Training Neural Networks: Core Idea

Neural networks are trained to **minimize a loss function** using **gradient descent**. In RL, this loss is typically based on **prediction error** or **control objectives**.

1. Prediction Objective (Value Function Approximation)

We want to approximate $v_\pi(s) \approx \hat{v}(s, \theta)$, where θ are neural network parameters.

A common loss: **Mean Squared TD Error**

$$L(\theta) = [R_{t+1} + \gamma \hat{v}(S_{t+1}, \theta^-) - \hat{v}(S_t, \theta)]^2$$

- θ^- optional **target network** parameters (kept fixed for stability)
- The **TD target** is semi-bootstrapped, introducing training instability

Training uses **stochastic gradient descent (SGD)**:

$$\theta \leftarrow \theta + \alpha \cdot \delta_t \cdot \nabla_\theta \hat{v}(S_t, \theta)$$

2. Control Objective (Action-Value Approximation)

In control, we want to approximate optimal $q_*(s, a)$ using $\hat{q}(s, a; \theta)$

Example: **Deep Q-Learning (DQN) Loss**

$$L(\theta) = \left[R_{t+1} + \gamma \max_{a'}(S_{t+1}, a'; \theta^-) - \hat{v}(S_t, A_t; \theta)\right]^2$$

Same training idea:

- Use experience replay to decorrelate data
- Use target networks to stabilize learning

---

3. Policy Gradient (for Stochastic Policies)

If using neural networks to represent policies $\pi(a \mid s;\theta)$, the **policy gradient** is used to train the network:

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi\left[\nabla_\theta \log \pi(A_t|S_t;\theta) \cdot \widehat{A}_t\right]$$

Where $\widehat{A}_t$ is an estimate of **advantage** (e.g., TD error, or Monte Carlo return minus baseline).

Used in:

- **REINFORCE**
- **Actor-Critic**
- **PPO**, **A3C**, etc.