# Lab 4: Reinforcement Learning with Linear Function Approximation - Mountain Car

Reinforcement Learning Course

## Introduction and Motivation

Reinforcement Learning problems involving continuous state spaces require methods that can generalize across infinitely many states. In this lab, we study **policy evaluation** in the *Mountain Car* environment using **linear function approximation**. This provides an instructive setting in which to understand how the choice of feature representation affects learning, why Monte Carlo (MC) and Temporal-Difference (TD) methods behave differently, and how linear methods can be extended to nonlinear approximations through feature engineering.

## The Mountain Car Environment

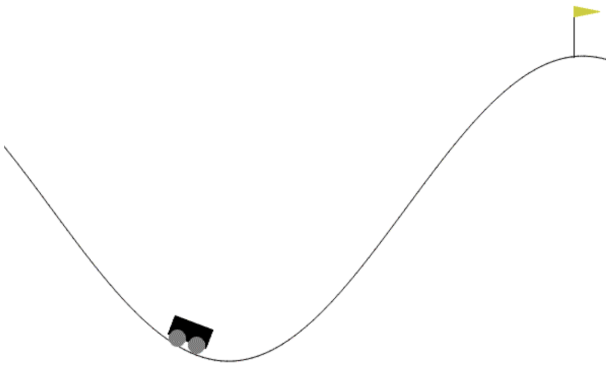

Figure 1: An illustration of the Mountain Car environment. The agent starts in the valley and must build momentum by driving left before reaching the goal on the right hill.

The Mountain Car environment is a classic RL benchmark provided in the Gym library. The state is two-dimensional:

$$s = (x, v),$$

where $x$ is the **horizontal position** of the car along a sinusoidal track, and $v$ is its velocity. The position is bounded by $x \in [-1.2, 0.6]$ and the goal is located at $x = 0.5$ on the right hill. The velocity is bounded by $v \in [-0.07, 0.07]$. The action space is discrete:

$$a \in \{0 \text{ (push left)},\ 1 \text{ (no push)},\ 2 \text{ (push right)}\}.$$

The dynamics follow:

$$v_{t+1} = v_t + 0.001(a_t - 1) - 0.0025\cos(3x_t), \qquad x_{t+1} = x_t + v_{t+1},$$

and the episode terminates once $x \geq 0.5$ or after 200 steps. The agent receives a fixed reward of $-1$ at each time step until the goal is reached:

$$R_{t+1} = -1.$$

### Why Mountain Car is Challenging

The Mountain Car environment is deceptively simple yet conceptually challenging:

- The car's engine is too weak to drive straight up the right hill. To reach the goal, the agent must first move **away** from the goal by driving up the left hill to build momentum, then swing to the right.
- The reward is **sparse and constant**. There is no intermediate positive feedback to guide learning.
- The state space is **continuous**. Tabular methods are impossible, so we must use function approximation.

Together, these aspects make Mountain Car an ideal testbed for studying approximation, generalization, and the behavior of MC and TD learning.

## Function Approximation in Continuous Spaces

In continuous environments like Mountain Car, it is impossible to maintain a table of values, since the state space is uncountably infinite. Instead, we approximate the value function using a parameterized function:

$$\hat{v}(s; \mathbf{w}) = \mathbf{w}^\top \boldsymbol{\phi}(s),$$

where $\boldsymbol{\phi}(s)$ is a **feature representation** of the state and $\mathbf{w}$ is a weight vector to be learned.

Because the approximation is linear in the parameters, stability and convergence properties are more tractable compared to nonlinear function approximators. Importantly, the approximation need not be linear in the *state variables*; nonlinearity can be encoded in the choice of $\boldsymbol{\phi}(s)$.

## Feature Representations

The design of the feature vector $\boldsymbol{\phi}(s)$ is crucial. Better features yield better approximations and significantly affect the performance of MC and TD-learning.

In this lab, we consider three feature representations of increasing complexity.

### 1. Naïve Raw Features

A simple baseline uses the raw (normalized) state variables:

$$\boldsymbol{\phi}_{\text{raw}}(s) = [1,\ x,\ v]^\top.$$

This yields a hyperplane approximation of $v_\pi(s)$, which is almost always too simple to capture the nonlinear value landscape of Mountain Car.

### 2. Polynomial Features

A richer approximation can be constructed using second-order polynomial terms:

$$\boldsymbol{\phi}_{\text{poly}}(s) = [1,\ x,\ v,\ x^2,\ v^2,\ xv]^\top.$$

Polynomial features introduce curvature into the approximation, allowing for a smoother and more flexible value function estimate. However, global polynomials can still struggle to capture localized structure.

### 3. Tile Coding (Coarse Coding)

Tile coding is the most powerful of the representations considered here. The idea is to overlay the state space with several grids (called *tilings*), each slightly offset from the others. For each tiling, exactly one tile is active for any state, producing a sparse binary feature vector.

With $N$ tilings and $k$ tiles per tiling, tile coding produces $N$ active features for each state. Tile coding provides:

- **Local generalization**: nearby states activate similar tiles;
- **Nonlinear expressiveness**: the combination of overlapping grids captures complex shapes;
- **Compatibility with linear function approximation**: updates are simple, stable, and efficient.

Tile coding is widely used in classical RL problems and is particularly effective for Mountain Car.

## Policy Evaluation with Linear Function Approximation

Given a fixed policy $\pi$, we wish to estimate its value function $v_\pi(s)$. We use two classical algorithms.

### Monte Carlo (MC) Evaluation

MC evaluation uses full return estimates:

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots .$$

The weight update is:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha(G_t - \hat{v}(S_t; \mathbf{w}))\, \boldsymbol{\phi}(S_t).$$

MC methods are unbiased but typically have high variance. They do not bootstrap and only update at the end of an episode.

### Temporal-Difference (TD(0)) Evaluation

TD(0) uses a one-step bootstrap estimate:

$$\delta_t = R_{t+1} + \gamma \hat{v}(S_{t+1}; \mathbf{w}) - \hat{v}(S_t; \mathbf{w}),$$

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha\, \delta_t\, \boldsymbol{\phi}(S_t).$$

TD(0) has lower variance than MC and updates online at every step. However, because of its bootstrapping nature, TD(0) introduces bias and can be sensitive to poor feature representations.

## Control with Linear Function Approximation

Although the main focus of this lab is policy evaluation, linear function approximation can also be applied to control via action-value methods. Using a feature representation for state–action pairs, $\boldsymbol{\phi}(s, a)$, one can estimate the action-value function:

$$\hat{q}(s, a; \mathbf{w}) = \mathbf{w}^\top \boldsymbol{\phi}(s, a).$$

A common approach is **semi-gradient Sarsa(0)**, with updates:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha\big(R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}; \mathbf{w}) \;-\; \hat{q}(S_t, A_t; \mathbf{w})\big)\, \boldsymbol{\phi}(S_t, A_t).$$

When used with tile coding, such methods can successfully learn to solve Mountain Car by discovering the correct rocking behavior needed to build momentum and reach the goal.

Control with function approximation is more complex than prediction and introduces additional stability concerns, but it illustrates how the same ideas from prediction extend to learning a good policy.

## Part 1: Exploring the Mountain Car Environment

In this first part of the lab, your task is simply to **run the provided Colab notebook cells** and carefully observe the behavior of the Mountain Car environment. You should make sure you can clearly **explain the key concepts**.

- Inspect the printed state and action spaces.

- Run the example policy and examine the resulting plots of position, velocity, and the phase trajectory.
- Observe why the naive policy fails to reach the goal.

After completing this part, you should be able to briefly explain:

- what the state variables represent,
- what actions the agent can take,
- why a simple greedy strategy does not solve the task,
- how to interpret the plotted trajectory (including the phase plot),
- **and why function approximation is needed in environments with continuous state spaces**.

# Part 2: Linear and Polynomial Function Approximation

In this part, run the provided Colab code that evaluates a fixed policy using *linear* and *second-order polynomial* feature representations. You should be prepared to explain the results.

## What to do

- Run the code that computes the *reference value function* on a grid. Here, we evaluate the policy by systematically resetting the simulator to many different states $(x, v)$ and running multiple Monte Carlo rollouts from each state. This produces a high-accuracy estimate of the value function across the entire state space. **This approach is only possible because Mountain Car is a very small two-dimensional problem and the simulator allows us to manually set the state. In general RL problems, the state space is far too large to sample in this way, and environments do not typically allow arbitrary resets.**
- Run the Monte Carlo (MC) and TD(0) prediction methods with:
  - raw linear features, and
  - second-order polynomial features.
- Inspect the plots comparing the learned value function to the reference along the position axis (for several fixed velocities).
- Look at the RMSE values printed for each method and feature set.

### What to observe

- The **linear feature** representation produces a value estimate that is essentially a plane; it captures only a very crude trend and cannot represent the curvature of the true value function.
- The **polynomial features** introduce some curvature and generally fit the reference function better, but still miss important nonlinear structure.
- Both MC and TD(0) produce value functions consistent with the expressiveness of their feature representations: linear features lead to linear-looking approximations, polynomial features lead to smoother quadratic shapes.
- Neither representation provides an accurate approximation across the whole state space; this motivates the use of richer, more flexible features in Part 3.

### Be prepared to explain

- how the raw and polynomial feature vectors are constructed,
- why a linear function of these features imposes strong restrictions on the shape of the estimated value function,
- why the resulting approximations retain the expected "linear" or "quadratic" structure,
- and why these simple feature sets do not capture the true value function well in Mountain Car.

## Part 3: Tile Coding

In this part, you will evaluate the fixed policy using *tile coding*, which provides a much richer feature representation than the linear or polynomial features used in Part 2. Tile coding allows a linear value function to approximate nonlinear structures by activating one tile per tiling, with multiple overlapping tilings giving local generalization.

### What to do

- Run the Colab code that implements tile coding and applies MC and TD(0) value prediction.
- Inspect the RMSE values comparing the tile-coded estimates to the reference.
- Plot the value function along the position axis for several choices of fixed velocity (the code allows you to select $v_0$).
- Study the state-visitation heatmap provided in the notebook.

### What to observe

- Tile coding generally provides a much better approximation than the raw or polynomial features from Part 2.
- However, the accuracy of the approximation varies across the state space: in some regions the estimate matches the reference extremely well, while in others it may deviate noticeably.
- Pay particular attention to slices at, for example, $v_0 = 0.5$ and $v_0 = 0$: along these slices the approximation is very good in some intervals of $x$ and much worse in others.

### Be prepared to explain

- why tile coding improves the approximation compared to the features in Part 2,
- why the approximation is *very accurate* in some parts of the state space and *poor* in others,
- how the accuracy along specific slices (*e.g.*, at $v_0 = 0.5$ or $v_0 = 0$) correlates with the visitation heatmap provided in the notebook,
- and what this tells you about using function approximation in reinforcement learning.

## Part 4: Control with Sarsa and Tile Coding

In this part, you move from policy evaluation to *control*. You will use Sarsa(0) with tile-coded action-value features to learn a policy that solves the Mountain Car task, and then visualize how the learned policy behaves.

explain the main ideas.

### What to do

- Run the code that trains a Sarsa(0) agent with tile coding and plots the number of steps per episode (and a moving average) over time.
- Run the code that records a short video of the learned greedy policy after training.

### What to observe

- The number of steps per episode is initially close to the maximum (the agent fails or reaches the goal very slowly), but the moving average should decrease over time as the policy improves.
- Even after learning, there will be noticeable variance in episode lengths due to random initial states and remaining exploration; focus on the overall trend rather than individual episodes.

- The video of the learned policy should show the car *rocking back and forth* to build up momentum and then reaching the goal significantly faster than the naive policies from earlier parts.

## Be prepared to explain

- how Sarsa(0) with tile-coded action-value features differs from the value prediction methods used in Parts 2 and 3,
- why the learning curve (steps per episode) is a reasonable measure of control performance in this task,
- and how the qualitative behaviour in the video reflects the policy learned by Sarsa.