Function Approximation

Ayça Özçelikkale

Dept. of Electrical Engineering, Uppsala University

New Teacher

Ayça Özçelikkale (How to pronounce my name? Read “I-cha” in English).

- Senior Lecturer at the Division of Signals and Systems, Dept. of Electrical Engineering
- **Key research themes:**
  - statistical estimation, online estimation and optimization, machine learning including reinforcement learning
  - applications of signal processing and machine learning in communications, sensor networks and production logistics
- **Professional activities:**
  - Technical committee member of various academic conferences
  - Past editor of IEEE Wireless Communications Letters

Course Activities with Ayça

I will hold the lectures for the following subjects:

- Function Approximation
- Model-based RL
- Policy-gradient Approaches

These subjects will be covered in the lectures L6-L9.

The corresponding tinkering notebooks are Notebook 4-6.

Basic Assignment 3, Basic Assignment 4 and a part of the Advanced Assignment will primarily focus on these subjects.

Motivation: Why do we study function approximation?

We would like to be able to apply RL to the following problems:

- Problems with large state space
  - Games such as Go (10^{170} states)
- Problems with continuous state-space
  - Robotic control tasks
  - Many real-life decision problems

Why can't we use the methods in the previous lectures?

- Previously: Every state s (or state-action) pair has an individual entry $v(s)$ or $q(s, a)$
- It is difficult to learn values of states and/or actions individually
- No generalization: Learning the values individually ignores the fact that observed stated can provide information about unseen states
  - Consider how the previous methods will learn the value functions for a very large maze!

Function Approximation

IDEA: We will approximate $v(s)$ or $q(s, a)$ using function approximation. The number of free parameters in the approximation will be typically smaller than the number of states/state-action pairs.

There exist various function approximation methods:

- Linear Approximation
- Neural Networks
- Decision Trees
- Nearest Neighbor

In this course, we primarily focus on differentiable function approximators. In particular, we work on the following:

- Linear Approximation (Today's class)
- Neural Networks (Project for 7.5hp course)

Feature Vectors

Idea: Associate the state s with a feature vector $\boldsymbol{x}(s) \in \mathbb{R}^d$:

$$\boldsymbol{x}(s) = \begin{bmatrix} x_1(s) \\ x_2(s) \\ \vdots \\ x_d(s) \end{bmatrix}$$

- Each $x_i(.)$ is a function from the state space to $\mathbb{R}$.
- The entire function $x_i(.)$ is called a feature.
- Features are also referred as basis functions for linear approximation.

Example: Edges in an Image (This is an example of contextually easily interpretable features)

Example: Polynomials

Example: Fourier Basis

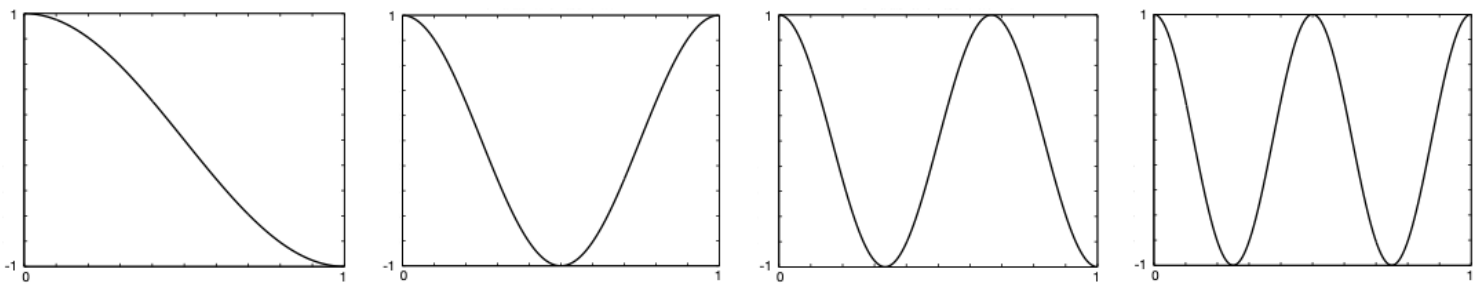
Example: Fourier Cosine Basis

Why Fourier Basis?: Fourier basis is easy to use and has shown to perform well in a range of RL problems.

Let $s \in [0,1]$. We consider one-dimensional order-n Fourier cosine basis with $n+1$ features: $x_i(s) = \cos(i\pi s)$, $s \in [0,1]$ for $i = 0, \ldots, n$.

Let $n = 4$. Feature vector can be written as follows:

$$\begin{bmatrix} x_0(s) \\ x_1(s) \\ \vdots \\ x_4(s) \end{bmatrix} = \begin{bmatrix} \cos(0\pi s) \\ \cos(1\pi s) \\ \vdots \\ \cos(4\pi s) \end{bmatrix}$$



x_i, $i = 1, \ldots, 4$ for approximating functions over $s \in [0,1]$

Self-study Exercise: Plot x_0.

Linear Value Function Approximation

We represent the value function by a linear combination of the features

$$\hat{v}(s, \boldsymbol{w}) = \sum_{i=1}^{d} w_i x_i(s) = \boldsymbol{w}^T \boldsymbol{x}(s) = \boldsymbol{x}(s)^T \boldsymbol{w}$$

where

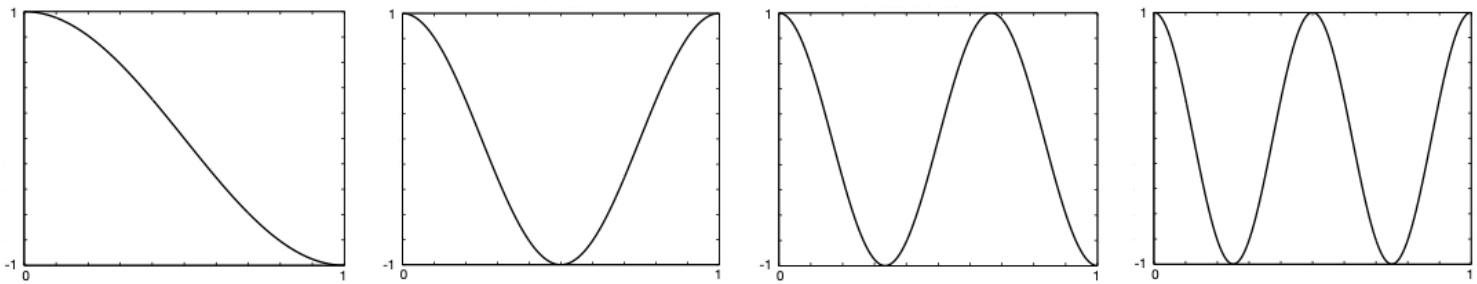
$$\boldsymbol{x}(s) = \begin{bmatrix} x_1(s) \\ x_2(s) \\ \vdots \\ x_d(s) \end{bmatrix}, \qquad \boldsymbol{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_d \end{bmatrix}$$

The approximation is linear in the weights.

Example: How do we form the approximation with order n Fourier cosine features?

Self-study Exercise: Suppose that you know $v(s)$ for $s_1, \ldots, s_m$. Suggest a method to find the weights $\boldsymbol{w}$ so that $v(s) \approx \hat{v}(s, \boldsymbol{w})$.

$$\hat{v}(s, \boldsymbol{w}) = \sum_{i=0}^{n+1} w_i x_i(s) = \boldsymbol{w}^T \boldsymbol{x}(s) = \boldsymbol{x}(s)^T \boldsymbol{w}$$



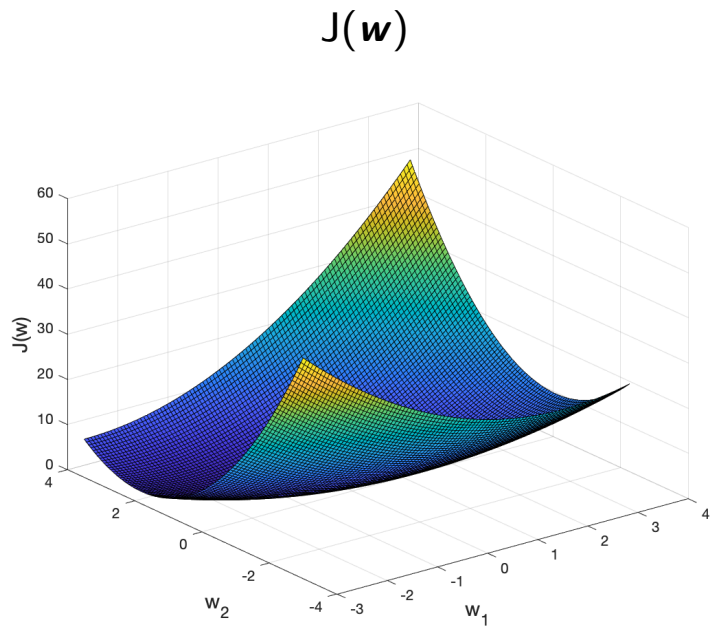
x_i, $i = 1, \ldots, 4$ for approximating functions over $s \in [0, 1]$

Note: If you think that you need to know that how much time we spend at each s_i, assume that time spent for all states are equal.

Gradient Descent: A general idea for minimizing functions

Why do we study gradient descent?

- To motivate stochastic gradient descent!
- Using stochastic gradient descent (SGD), we will motivate the RL methods for approximating v and q.



What is gradient descent? Gradient descent is an iterative procedure to find minimum of functions.

Problem statement: Let $J(\boldsymbol{w}) : \mathbb{R}^d \to \mathbb{R}$ be a scalar valued function with a vector input. We would like to find a $\boldsymbol{w}$ that minimizes $J(\boldsymbol{w})$ iteratively.

Notation- Gradient

Let $J(\boldsymbol{w}) : \mathbb{R}^d \rightarrow \mathbb{R}$ be a scalar valued function with a vector input. Gradient of $J(\boldsymbol{w})$ is given by

$$\nabla J \triangleq \frac{\partial J}{\partial \boldsymbol{w}} \triangleq \begin{bmatrix} \frac{\partial J}{\partial w_1} \\ \frac{\partial J}{\partial w_2} \\ \vdots \\ \frac{\partial J}{\partial w_d} \end{bmatrix} \in R^{d \times 1}$$

The gradient evaluated at a particular value $\bar{\boldsymbol{w}}$ is denoted by the following:

$$\nabla J(\bar{\boldsymbol{w}}) = \left. \frac{\partial J}{\partial \boldsymbol{w}} \right|_{\boldsymbol{w}=\bar{\boldsymbol{w}}}$$

The gradient $\nabla J(\bar{\boldsymbol{w}})$ gives the direction of maximum ascent around $\bar{\boldsymbol{w}}$.

Self-study Exercise: Let $J(\boldsymbol{w}) = w_1^2 + 3w_2$. Find $\nabla J(\bar{\boldsymbol{w}})$ for $\bar{\boldsymbol{w}} = \begin{bmatrix} 5 \\ 2 \end{bmatrix}$.

Gradient Descent Idea

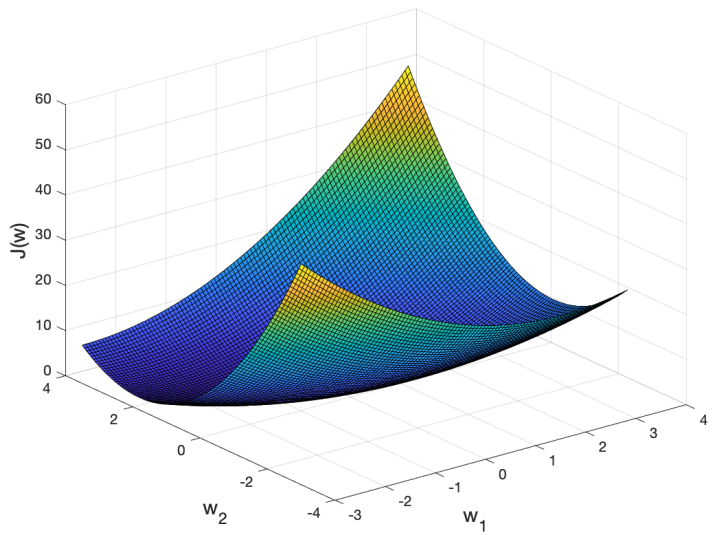
Gradient Descent Idea: If you want to minimize a function $J(\boldsymbol{w})$ iteratively, start from an initial point $\boldsymbol{w}_i$ (i is the iteration index!), and move in the direction of maximum "descent":

$$\begin{aligned}\boldsymbol{w}_{i+1} &= \boldsymbol{w}_i + \Delta \boldsymbol{w}_i \\ &= \boldsymbol{w}_i - \frac{1}{2}\alpha \nabla J(\boldsymbol{w}_i)\end{aligned}$$

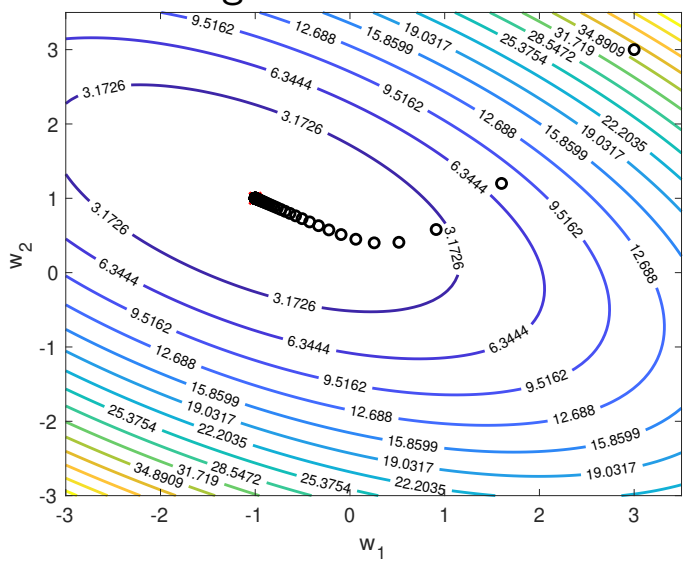
where $\alpha > 0$ is the step size. (Here, $\frac{1}{2}$ is just to make our notation neater since it will cancel out later.)

Example: Find $\boldsymbol{w}$ that minimizes $J(\boldsymbol{w})$ plotted below.

J($\boldsymbol{w}$)



Contour Plot of $J(\boldsymbol{w})$ with example trajectory of gradient descent



Value Function Approximation with Gradient Descent

- Let J($\boldsymbol{w}$) denote the mean-square error between the true value function $v_\pi(s)$ and our approximation $\hat{v}(s, \boldsymbol{w})$:

$$J(\boldsymbol{w}) = \mathbb{E}_\pi[(v_\pi(S) - \hat{v}(S, \boldsymbol{w}))^2]$$

- We would like to find the optimal $\boldsymbol{w}$ that minimizes $J(\boldsymbol{w})$ iteratively using gradient descent:

$$\begin{aligned}\boldsymbol{w}_{i+1} &= \boldsymbol{w}_i - \frac{1}{2}\alpha\nabla J(\boldsymbol{w}_i) \\ &= \boldsymbol{w}_i + \alpha\mathbb{E}_\pi[v_\pi(S) - \hat{v}(S, \boldsymbol{w}_i)]\nabla\hat{v}(S, \boldsymbol{w}_i)\end{aligned}$$

- It is difficult to know $\mathbb{E}_\pi[v_\pi(S) - \hat{v}(S, \boldsymbol{w})]$!
- Stochastic Gradient Descent Idea: Use a sample instead of the expectation

$$\boldsymbol{w}_{i+1} = \boldsymbol{w}_i + \alpha(v_\pi(S) - \hat{v}(S, \boldsymbol{w}_i))\nabla\hat{v}(S, \boldsymbol{w}_i)$$

- Hence, the update $\Delta\boldsymbol{w}_i$ is given by: $\Delta\boldsymbol{w}_i = \alpha(v_\pi(S) - \hat{v}(S, \boldsymbol{w}_i))\nabla\hat{v}(S, \boldsymbol{w}_i)$

$$\Delta\boldsymbol{w}_i = \text{step size} \times \text{prediction error} \times \text{gradient of value function approximation}$$

Linear Value Function Approximation with Gradient Descent

- Let J($\boldsymbol{w}$) denote the mean-square error between the true value function $v_\pi(s)$ and our approximation $\hat{v}(s, \boldsymbol{w})$:

$$J(\boldsymbol{w}) = \mathbb{E}_\pi[(v_\pi(S) - \boldsymbol{x}(S)^T \boldsymbol{w})^2]$$

- We would like to find the optimal $\boldsymbol{w}$ that minimizes $J(\boldsymbol{w})$ iteratively using gradient descent:

$$\begin{aligned} \boldsymbol{w}_{i+1} &= \boldsymbol{w}_i - \frac{1}{2}\alpha \nabla J(\boldsymbol{w}_i) \\ &= \boldsymbol{w}_i + \alpha \mathbb{E}_\pi[v_\pi(s) - \boldsymbol{x}(S)^T \boldsymbol{w}_i]\boldsymbol{x}(S) \end{aligned}$$

- It is difficult to know $\mathbb{E}_\pi[v_\pi(S) - \boldsymbol{x}(S)^T \boldsymbol{w}_i]$!
- Stochastic Gradient Descent Idea: Use a sample instead of the expectation

$$\boldsymbol{w}_{i+1} = \boldsymbol{w}_i + \alpha((v_\pi(S) - \hat{v}(S, \boldsymbol{w}_i))\boldsymbol{x}(S)$$

- Hence, the update $\Delta \boldsymbol{w}_i$ is given by $\Delta \boldsymbol{w}_i = \alpha(v_\pi(S) - \hat{v}(S, \boldsymbol{w}_i))\boldsymbol{x}(S)$

$$\Delta \boldsymbol{w}_i = \text{step size} \times \text{prediction error} \times \text{gradient of value function approximation}$$

Linear Value Function Approximation with Gradient Descent Without Knowing $v_\pi(S)$

- We obtained
$$\Delta \boldsymbol{w} = \alpha((v_\pi(s) - \hat{v}(s, \boldsymbol{w}))\boldsymbol{x}(s)$$
i.e., we assumed that true value function is known, but we do not know $v_\pi(s)$!
- We substitute a target for $v_\pi(s)$:
  - For Monte-Carlo, the target is the return G_t
$$\Delta \boldsymbol{w} = \alpha(G_t - \hat{v}(S_t, \boldsymbol{w}))\boldsymbol{x}(S_t)$$
  - For TD(0), the target is the TD-target $R_{t+1} + \gamma \hat{v}(S_{t+1}, \boldsymbol{w})$
$$\Delta \boldsymbol{w} = \alpha(R_{t+1} + \gamma \hat{v}(S_{t+1}, \boldsymbol{w}) - \hat{v}(S_t, \boldsymbol{w}))\boldsymbol{x}(S_t)$$

Monte-Carlo with Linear Value Function Approximation

- Return G_t is an unbiased, noisy estimate of the true value $v_\pi(s)$
- Below Monte-Carlo algorithm converges to a global optimum (if α is reduced over time appropriately).

Input: Inputs: Policy π, features $\boldsymbol{x}(\cdot)$.

Initialization: Set step size $\alpha > 0$. Initialize weights $\boldsymbol{w} \in \mathbb{R}^d$.

repeat (for each episode)

Generate an episode using π: $S_0, A_0, R_1, S_1, A_1, \ldots, R_T, S_T$.

repeat : (for each step of the episode $t = 0, 1, \ldots T-1$)

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha(G_t - \hat{v}(S_t, \boldsymbol{w}))\boldsymbol{x}(S_t)$$

Desired Output $\hat{v}(s, \boldsymbol{w}) \approx v_\pi(s)$

Example: Function Approximation for 1000-state random walk task

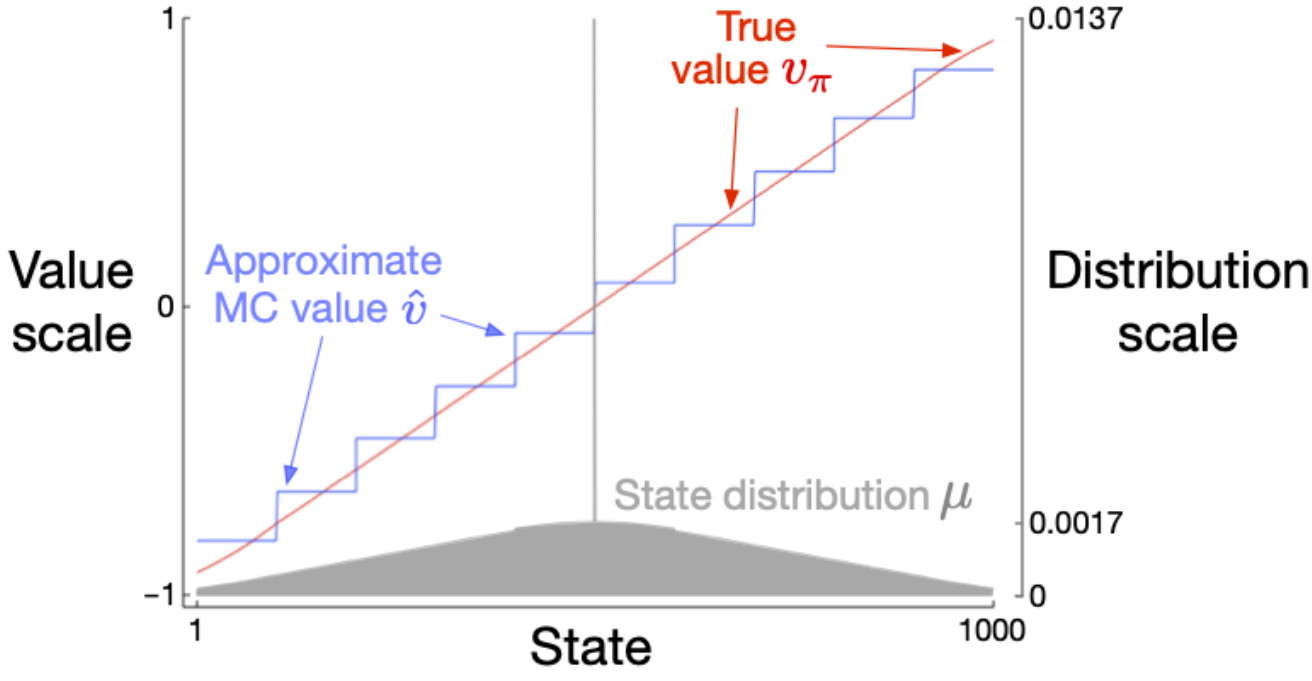
Task:

- A corridor of non-terminal 1000-states and one terminal state at each end
- State transitions are to one of the 100 neighboring states to its left, or to one of the 100 neighboring states to its right, all with equal probability.
  - Around the edges, all the probability due to fewer neighbours go the terminating on the edge. Example: State 950 has a 0.25 chance of terminating on the right terminating state within one step
- All episodes start at state 500
- All transitions have the reward of 0, except to the left terminating state with -1 and to the right terminating state with $+1$.

Function Approximation by state-aggregation: States are grouped together, with one approximated value (one component of the weight vector $\boldsymbol{w}$) for each group. Here, the 1000 states were partitioned into 10 groups of 100 states each.

Self-study Exercise: What are the basis functions for the above state-aggregation setting?

Example: Function Approximation for 1000-state random walk task



Value approximation and state distribution

State distribution $\mu(s)$: fraction of time spent on s under policy π. By definition, $\mu(s) \geq 0$ $\sum_{s \in \mathcal{S}} \mu(s) = 1$. We have $\sum_{s \in \mathcal{S}} \mu(s)(v_\pi(s) - \hat{v}(s, \boldsymbol{w}))^2 = \mathbb{E}_\pi[(v_\pi(S) - \hat{v}(S, \boldsymbol{w}))^2]$.

TD(0) with Linear Value Function Approximation

- The TD-target $R_{t+1} + \gamma \hat{v}(S_{t+1}, \boldsymbol{w})$ is a biased estimate of the true value $v_\pi(s)$
- In contrast to Monte-Carlo methods, we do not need to wait for the end of episode.
- Below TD(0) algorithm converges to a local optimum.

Input: Inputs: Policy π, features $\boldsymbol{x}(\cdot)$.
Initialization: Set step size $\alpha > 0$. Initialize weights $\boldsymbol{w} \in \mathbb{R}^d$.
repeat (for each episode)

Initialize S.

repeat : (for each step of the episode)

Choose $A \sim \pi(.|S)$

Take action A, observe R, S'

$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha(R + \gamma \hat{v}(S', \boldsymbol{w}) - \hat{v}(S, \boldsymbol{w}))\boldsymbol{x}(S)$

$S \leftarrow S'$

Desired Output $\hat{v}(s, \boldsymbol{w}) \approx v_\pi(s)$

Linear Action-Value Function Approximation

Feature Vectors: Associate the state-action pair (s, a) with a feature vector $\boldsymbol{x}(s, a) \in \mathbb{R}^d$:

$$x(s,a) = \begin{bmatrix} x_1(s,a) \\ x_2(s,a) \\ \vdots \\ x_d(s,a) \end{bmatrix}$$

We represent the action-value function by a linear combination of the features

$$\hat{q}(s, a, \boldsymbol{w}) = \sum_{i=1}^{d} w_i x_i(s,a) = \boldsymbol{w}^T \boldsymbol{x}(s,a) = \boldsymbol{x}(s,a)^T \boldsymbol{w}$$

Action-Value Function Approximation

- Aim: Approximate the action value function $\hat{q}(S, A, \boldsymbol{w}) \approx q_\pi(S, A)$
- We're now interested in minimizing $J(\boldsymbol{w}) = \mathbb{E}_\pi[((q_\pi(S, A) - \hat{q}(S, A, \boldsymbol{w}))^2]$
- The update: step size $\times$ prediction error $\times$ gradient of action-value function approx.

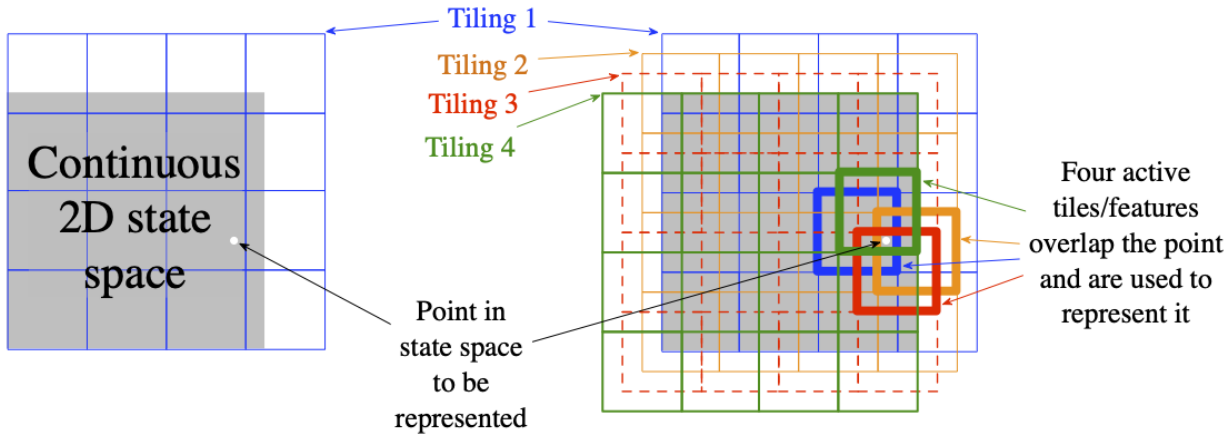
$$\Delta \boldsymbol{w} = \alpha((q_\pi(S, A) - \hat{q}(S, A, \boldsymbol{w}))\nabla \hat{q}(S, A, \boldsymbol{w})$$

- We substitute a target for $q_\pi(S, A)$. For Sarsa, we have

$$\Delta \boldsymbol{w} = \alpha(R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \boldsymbol{w}) - \hat{q}(S_t, A_t, \boldsymbol{w}))\nabla \hat{q}(S_t, A_t, \boldsymbol{w})$$

- Actions from continuous spaces or from large discrete sets are more difficult to handle than such state spaces.

Tile Coding



Multiple, over-lapping grid-tilings, each off-set by a fraction of a tile-width on a bounded two-dimensional space.

- The number of features that are active at one time is the same for any state. Setting the step size α is easier.
- Since the features are binary vectors, calculation of approximate value function is easy.

Mountain Car

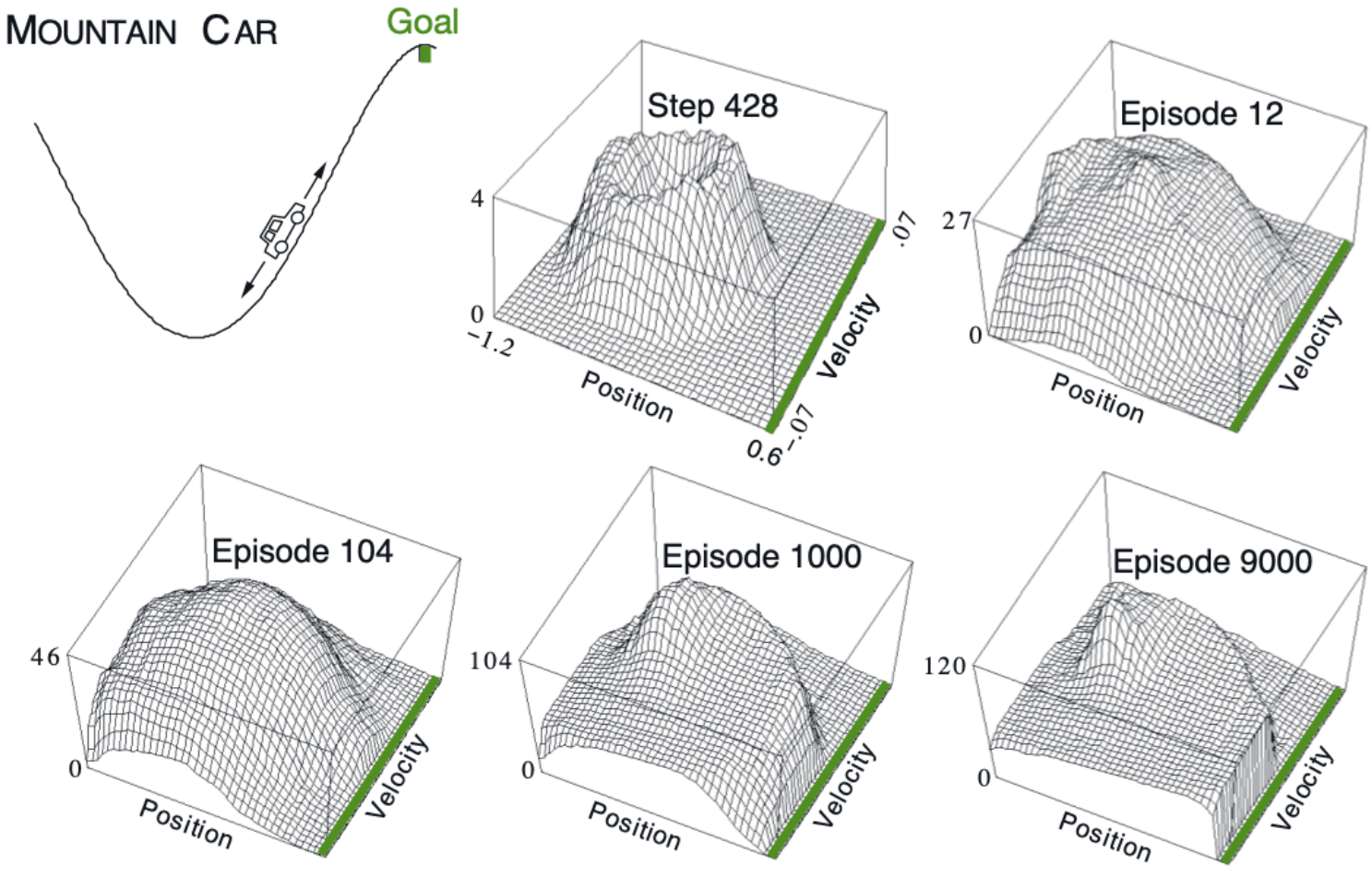
Mountain Car Task:

- States: Position $x_t \in [-1.2, 0.6]$, velocity $\dot{x}_t \in [-0.07, 0.07]$
- Actions: + full forward (+1), -full reverse (-1), zero throttle (0).
- Reward: -1 for all transitions.
- Car starts randomly $x_t \in [-0.6, -0.4)$.
- Episode terminates when the car passes the goal.
- Caveat: Car cannot climb the hill the goal resides without backing on the hill on the left.

Approach: Sarsa with linear approximation with tile coding

- Initialization: Initial state-action values are set to an optimistic 0 (all true values are negative in this task).
- ϵ-greedy with $\epsilon = 0$

Mountain Car with Sarsa with linear approximation



Cost-to-go function ($-\max_a \hat{q}(s, a, \boldsymbol{w})$) for Mountain Car task

Concluding Remarks I

Motivation: With function approximation, we would like to be able to

- deal with large/continuous state/action-spaces
- use less computation/memory/experience
- obtain better generalization properties

Linear Approximation

- Value Function: $\hat{v}(s, \boldsymbol{w}) = \sum_{i=1}^{d} w_i x_i(s)$
- Action-Value Function: $\hat{q}(s, a, \boldsymbol{w}) = \sum_{i=1}^{d} w_i x_i(s, a)$

Concluding Remarks II

How to find the "best" approximation? We combine two ideas:

- Stochastic Gradient Descent
  - Update: step size $\times$ prediction error $\times$ gradient of (action-)value function approx.

$$\Delta \boldsymbol{w} = \alpha((v_\pi(S) - \hat{v}(S, \boldsymbol{w}))\nabla \hat{v}(S, A, \boldsymbol{w})$$

$$\Delta \boldsymbol{w} = \alpha((q_\pi(S, A) - \hat{q}(S, A, \boldsymbol{w}))\nabla \hat{q}(S, A, \boldsymbol{w})$$

- Replace $v_\pi(s, \boldsymbol{w})$ or $q(s, a, \boldsymbol{w})$ with an appropriate update target:
  - MC for predicting $v_\pi(s, \boldsymbol{w})$: G_t
  - TD(0) for predicting $v_\pi(s, \boldsymbol{w})$: $R_{t+1} + \gamma \hat{v}(S_{t+1}, \boldsymbol{w})$
  - Sarsa for predicting $q_\pi(s, a, \boldsymbol{w})$: $R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \boldsymbol{w})$

More?

- Important Non-linear Approximation Approach: Artificial Neural Networks.

References

- Sutton, Barto, Reinforcement Learning: An Introduction, 2nd edition
- Bertsekas, Reinforcement Learning and Optimal Control, 1st edition
- Tsitsiklis and Van Roy. An Analysis of Temporal-Difference Learning with Function Approximation. 1997