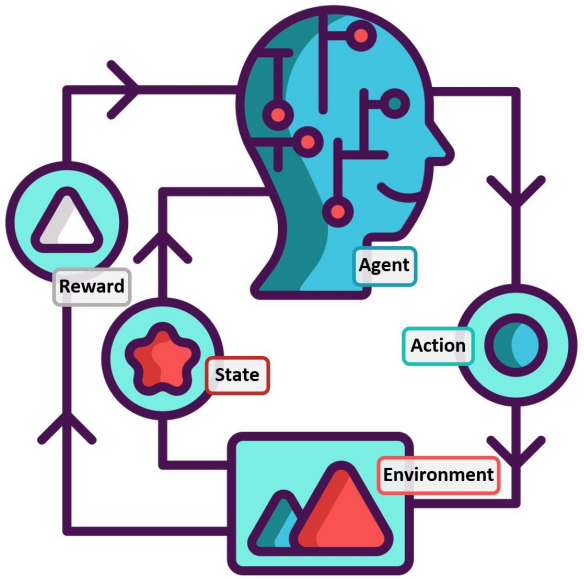
AMCO
ARTIFICIAL INTELLIGENCE, MACHINE LEARNING AND CONTROL RESEARCH GROUP

Reinforcement Learning
2025/2026

RL2
Reinforcement Learning Research Lab @ UniPD

Lecture #14
Value Function Approximation

Gian Antonio Susto



Announcements before starting: 34 Importance Sampling Algorithm submissions...

An example of good solution

Algorithm 1 Monte Carlo Off-Policy Prediction for $V_\pi(s)$ (Weighted Importance Sampling - Average Implementation)

Input: target policy π, behaviour policy b
Initialize $V(s) \in \mathbb{R}$ arbitrarily for all $s \in \mathcal{S}$
Initialize $W_{\text{sum}}(s) \leftarrow 0$ for all $s \in \mathcal{S}$ ▷ Cumulative Sum of wefts
Initialize $G_{\text{sum}}(s) \leftarrow 0$ for all $s \in \mathcal{S}$ ▷ Cumulative sum of Returns
loop ▷ For each episode
    Generate an episode following b: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    $W \leftarrow 1$ ▷ Importance sampling weight
    for $t = T-1, T-2, \ldots, 0$ **do**
        $G \leftarrow \gamma G + R_{t+1}$ ▷ Policy Evaluation
        $W \leftarrow W \cdot \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$ ▷ Importance Sampling ratio
        if $W > 0$ **then** ▷ Only update if weight is non-zero
            $G_{\text{sum}}(S_t) \leftarrow G_{\text{sum}}(S_t) + W \cdot G$
            $W_{\text{sum}}(S_t) \leftarrow W_{\text{sum}}(S_t) + W$
            $V(S_t) \leftarrow \frac{G_{\text{sum}}(S_t)}{W_{\text{sum}}(S_t)}$ ▷ Weighted average
        end if
    end for
end loop

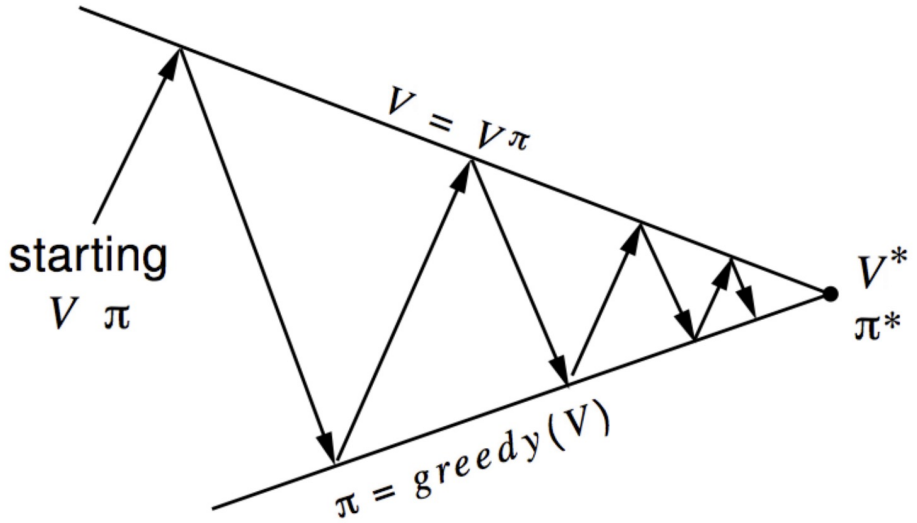
Recap: from Tabular RL…

- Up until now, all our approaches for prediction and control are based on the availability of good estimations of v and q
- So far we have represented value function by a lookup table

-> Every state s has an entry V (s)

-> Or every state-action pair s, a has an entry Q(s, a)

- We are therefore dealing with the so-called tabular RL: we have look-up tables that we reply on



… to Value Function Approximation

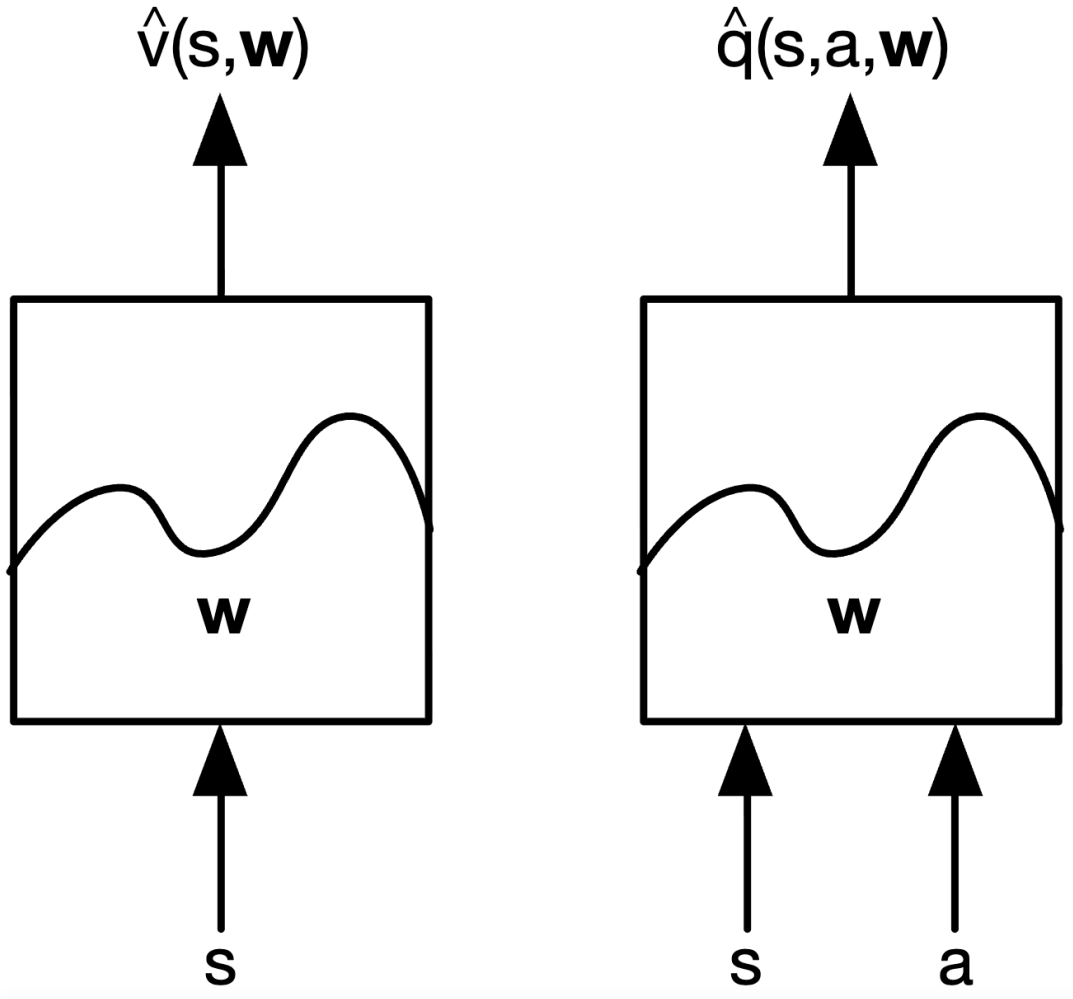
Solution for large MDPs:

- Estimate value function with function approximation

$$\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$$

$$\hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a)$$

- Generalise from seen states to unseen states



Recap: Limits of Tabular RL

Unfortunately, in the naïve version, the tabular RL algorithms seen so far have some strong limitations: if the scale of problem become bigger, tables can become enormous

-> Backgammon: 10^{20} states

-> Go: 10^{170} states

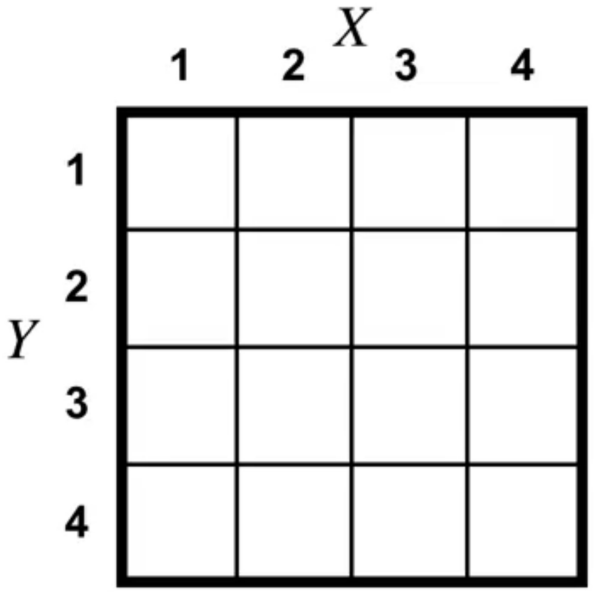
-> Autonomous driving (continuous problem): continuous state space

Problems

- Memory: tables impossible to store
- Data/Time: to complete all entries on the tables with enough data to have good estimations, we need huge amount of data
- No generalization: a single update only impact an element of the table

Generalization with VFA

$$\hat{v}(s, \mathbf{w}) \doteq w_1 X + w_2 Y$$



$w_1 = 1$
$w_2 = 1$

Y \ X	1	2	3	4
1	2	3	4	5
2	3	4	5	6
3	4	5	6	7
4	5	6	7	8

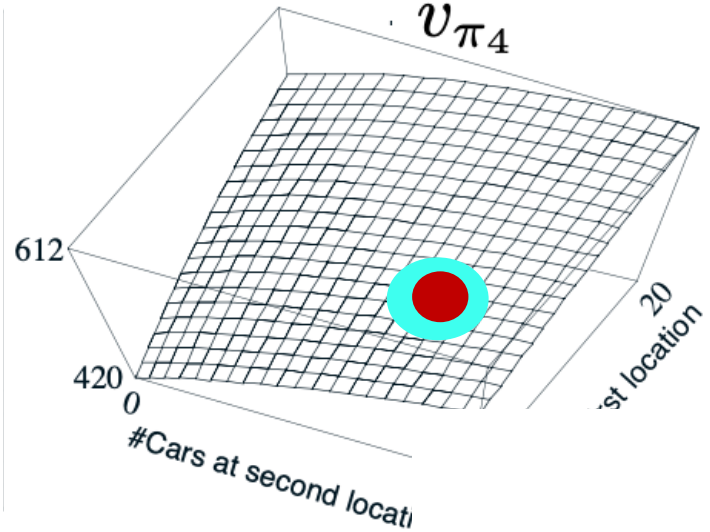
$w_1 = 4$
$w_2 = 1$

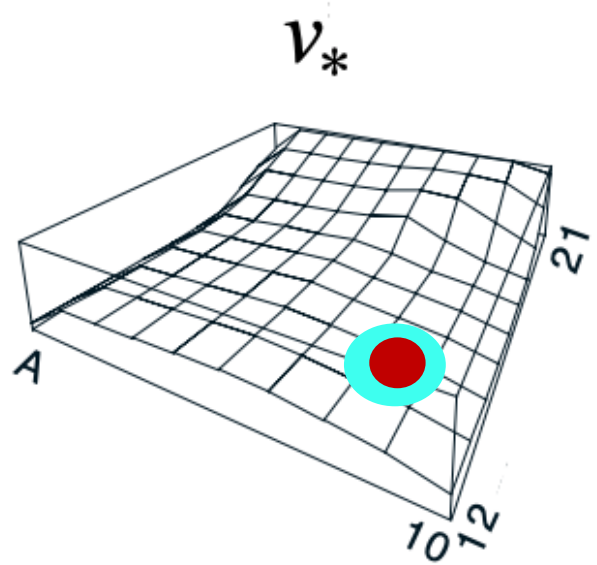
Y \ X	1	2	3	4
1	5	9	13	17
2	6	10	14	18
3	7	11	15	19
4	8	12	16	20

One change of parameters (1 step, 1 episode), affect potentially the estimations for all states!

Generalization with VFA

Moreover, this is typically what we do in supervised learning! We don't have data for all possible situations; instead, we aim to find a function capable of generalizing—i.e., one that can describe the underlying phenomena for unseen data.





seen

unseen

Recap: From true Q and V to approximation - example

Small grid-world: 16 states

Can we find an alternative, more ‘compact’, description for v?

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-5
-3	-4	-5	-6

V_7

For example, we can introduce descriptors (features) of the state (x,y) = ([0, …, 3], [0, …, 3]) we can see V is

$$V(x, y) = -(x + y)$$

A function approximator can, for example, have the form:

$$\hat{V}(s; \theta) = \theta_0 + \theta_1 x + \theta_2 y$$

Recap: From true Q and V to approximation - example

Small grid-world: 16 states

Can we find an alternative, more ‘compact’, description for v?

For example, we can introduce descriptors (features) of the state (x,y) = ([0, ..., 3], [0, ..., 3]) we can see V is

$$V(x, y) = -(x + y)$$

A function approximator can, for example, have the form:

$$\hat{V}(s; \theta) = \theta_0 + \theta_1 x + \theta_2 y$$

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-5
-3	-4	-5	-6

V_7

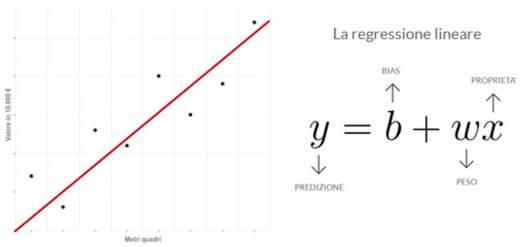
Depending on the problem we can come out with different type of features/compact representations of the problem

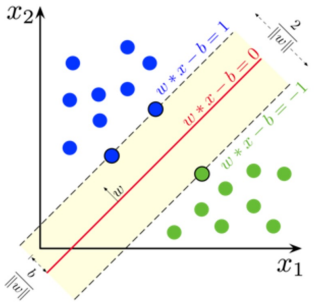
Recap: Which function approximator?

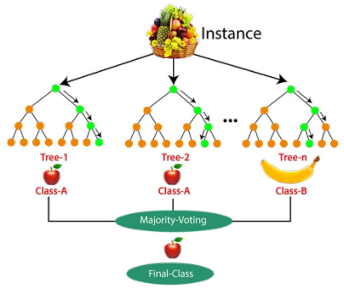
<u>In principle all supervised learning approaches are fine!</u>

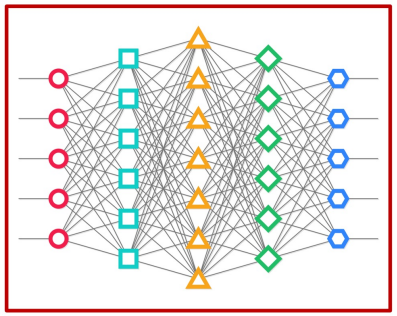
We can use:

- Linear regression
- Neural networks / Deep Learning
- SVM
- Decision trees
- Ensemble approaches (Random Forest, XGBoost, …)









Neural Networks (Deep Learning)

However, we have a strong preference for differentiable approaches: some approach that I can learn along the way (ie. while I'm collecting experience interacting with the real world)

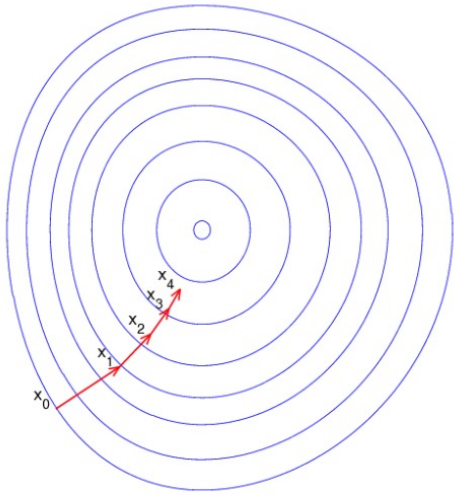
Differentiable Supervised Learning approaches

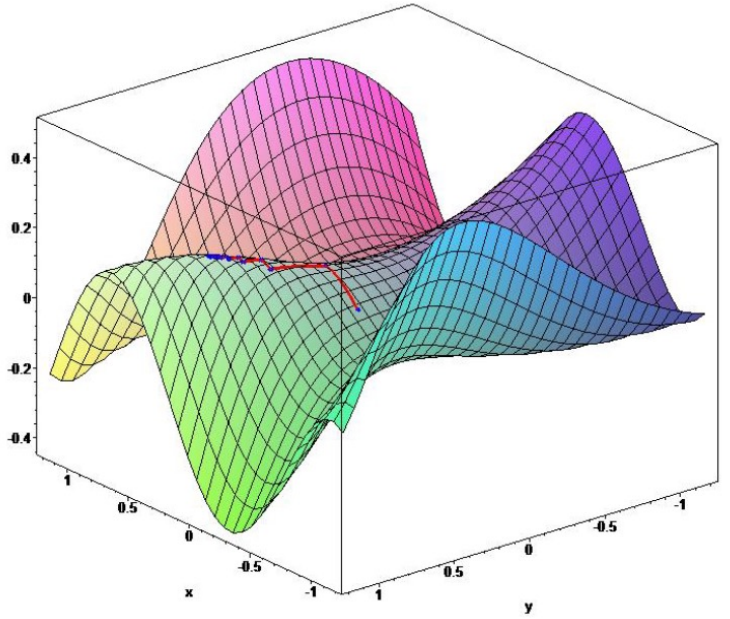
With:

- Linear approaches
- Neural networks / Deep Learning

we can use stochastics gradient descent and update (for example with new data)

The procedure is incremental: we can see this as ‘learning along the way’





Differentiable Supervised Learning approaches

With:

- Linear approaches
- Neural networks / Deep Learning

we can use stochastics gradient descent and update (for example with new data)

The procedure is incremental: we can see this as 'learning along the way'

Linear approaches:

- Easy intuitions and guarantees
- Require the definition of features

Neural networks / Deep Learning approaches:

- Self-learning of features
- More complicated function approximators
- No guarantees and more complexity

Next week, lecture 15-16 will be dedicated to basics of NN/DL!

We need a Prediction Objective!

- In the tabular case we could aim for perfect matching of the true value function v_π. But when using function approximation, this is no longer feasible: changes in one state's estimate affect other states & therefore not possible to get the values of all states exactly correct!
- Moreover, making one state's estimate more accurate might make others'less accurate!
- We therefore need a formal objective that defines what "good" approximation means globally — over all states.
- Define the approximate value function $\widehat{v_\pi}(s, \boldsymbol{w})$, parameterized by weight vector $\boldsymbol{w}$
- Define a state-distribution $\mu(s) \geq 0$, with $\sum \mu(s) = 1$ (sum over all the states), which captures how much we care about each state.
- Then the objective is the so-called Mean-Squared Value Error (VE):

$$\overline{VE}(\mathbf{w}) \doteq \sum_{s \in \mathcal{S}} \mu(s) \left[v_\pi(s) - \hat{v}(s, \mathbf{w}) \right]^2$$

We need a Prediction Objective!

- In on-policy continuous tasks μ(s) is the stationary distribution under π
- For simplicity we will consider this case onwards

$$\overline{\mathrm{VE}}(\mathbf{w}) \doteq \sum_{s \in \mathcal{S}} \mu(s) \left[v_\pi(s) - \hat{v}(s, \mathbf{w}) \right]^2$$

$$J(\mathbf{w}) = \mathbb{E}_\pi \left[\left(v_\pi(s) - \hat{v}(s, \mathbf{w}) \right)^2 \right]$$

Recap on Stochastic Gradient Descent (SGD)

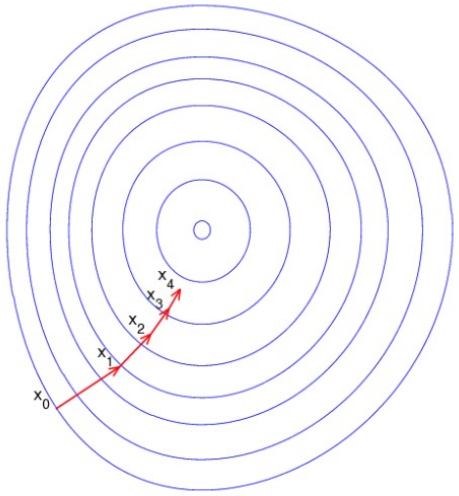
- Let $J(\mathbf{w})$ be a differentiable function of parameter vector $\mathbf{w}$
- Define the *gradient* of $J(\mathbf{w})$ to be

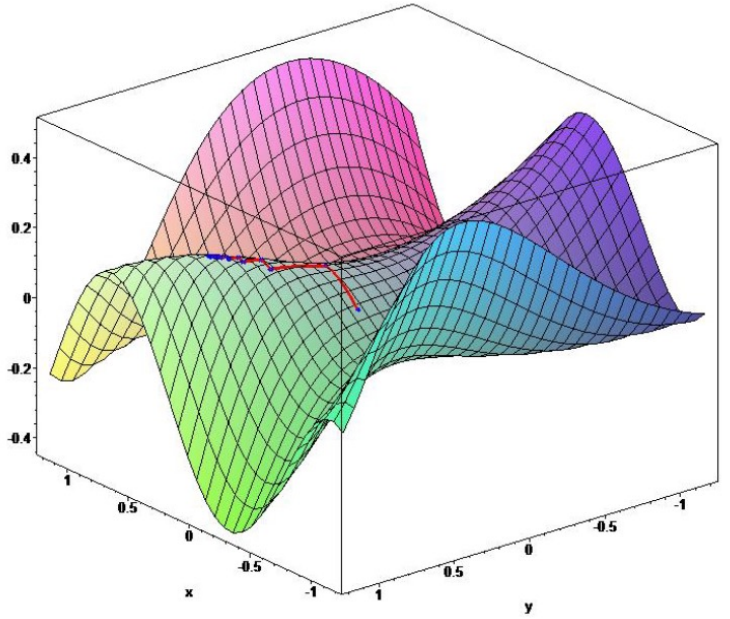
$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \begin{pmatrix} \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_1} \\ \vdots \\ \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_n} \end{pmatrix}$$

- To find a local minimum of $J(\mathbf{w})$
- Adjust $\mathbf{w}$ in direction of -ve gradient

$$\Delta \mathbf{w} = -\frac{1}{2}\alpha \nabla_{\mathbf{w}} J(\mathbf{w})$$

where α is a step-size parameter





Stochastic Gradient Descent (SGD)

- Goal: find parameter vector $\mathbf{w}$ minimising mean-squared error between approximate value fn $\hat{v}(s, \mathbf{w})$ and true value fn $v_\pi(s)$

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

Only computable if the oracle is available!

- Gradient descent finds a local minimum

$$\begin{aligned} \Delta \mathbf{w} &= -\frac{1}{2} \alpha \nabla_{\mathbf{w}} J(\mathbf{w}) \\ &= \alpha \mathbb{E}_\pi \left[(v_\pi(S) - \hat{v}(S, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w}) \right] \end{aligned}$$

A positive step-size parameter, following this condition to obtain convergence (more later)

$$\sum_{t=1}^{\infty} \alpha_t = \infty \qquad \sum_{t=1}^{\infty} \alpha_t^2 < \infty$$

- Stochastic gradient descent *samples* the gradient

$$\Delta \mathbf{w} = \alpha (v_\pi(S) - \hat{v}(S, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w})$$

- Expected update is equal to full gradient update $\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \nabla \mathbf{w}_t$

$$= \mathbf{w}_t + \alpha \left[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t) \right] \nabla \hat{v}(S_t, \mathbf{w}_t)$$

In the linear case

- Represent value function by a linear combination of features

$$\hat{v}(S, \mathbf{w}) = \mathbf{x}(S)^\top \mathbf{w} = \sum_{j=1}^{n} \mathbf{x}_j(S) \mathbf{w}_j$$

- Objective function is quadratic in parameters $\mathbf{w}$

$$J(\mathbf{w}) = \mathbb{E}_\pi \left[(v_\pi(S) - \mathbf{x}(S)^\top \mathbf{w})^2 \right]$$

- Stochastic gradient descent converges on *global* optimum
- Update rule is particularly simple

$$\nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w}) = \mathbf{x}(S)$$

$$\Delta \mathbf{w} = \alpha (v_\pi(S) - \hat{v}(S, \mathbf{w})) \mathbf{x}(S)$$

Update = *step-size* $\times$ *prediction error* $\times$ *feature value*

In the linear case

- Represent value function by a linear combination of features

$$\hat{v}(S, \mathbf{w}) = \mathbf{x}(S)^\top \mathbf{w} = \sum_{j=1}^{n} \mathbf{x}_j(S) \mathbf{w}_j$$

- Objective function is quadratic in parameters $\mathbf{w}$

$$J(\mathbf{w}) = \mathbb{E}_\pi \left[(v_\pi(S) - \mathbf{x}(S)^\top \mathbf{w})^2 \right]$$

- Stochastic gradient descent converges on *global* optimum
- Update rule is particularly simple

$$\nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w}) = \mathbf{x}(S)$$

$$\Delta \mathbf{w} = \alpha (v_\pi(S) - \hat{v}(S, \mathbf{w})) \mathbf{x}(S)$$

Update = *step-size* × *prediction error* × *feature value*

Tabular RL is a special case of linear VFA

$$\mathbf{x}^{table}(S) = \begin{pmatrix} \mathbf{1}(S = s_1) \\ \vdots \\ \mathbf{1}(S = s_n) \end{pmatrix}$$

$$\hat{v}(S, \mathbf{w}) = \begin{pmatrix} \mathbf{1}(S = s_1) \\ \vdots \\ \mathbf{1}(S = s_n) \end{pmatrix} \cdot \begin{pmatrix} \mathbf{w}_1 \\ \vdots \\ \mathbf{w}_n \end{pmatrix}$$

Recap: Supervised Learning

- Supervised learning approaches requires tagged examples! We will need a dataset of examples

$$(s_1, v(s_1)), (s_2, v(s_2)), \dots, (s_D, v(s_D))$$

$$(s_1, a_1, q(s_1, a_1)), (s_2, a_2, q(s_2, a_2)), \dots, (s_D, a_D, q(s_D, a_D))$$

Core idea of Value Function Approximation

To create a supervised setting, instead of the true v and q, we use a target

- Monte Carlo target: the return

$$(s_1, G_t^1), (s_2, G_t^2), \dots, (s_D, G_t^D)$$

- TD(0) target: the TD target

$$(s_1, R_{t+1}^1 + \gamma \hat{v}(S_{t+1}^1, \boldsymbol{w})), (s_2, R_{t+1}^2 + \gamma \hat{v}(S_{t+1}^2, \boldsymbol{w})), \dots, (s_D, R_{t+1}^D + \gamma \hat{v}(S_{t+1}^D, \boldsymbol{w}))$$

- TD(λ) target: the λ-return

$$(s_1, G_1^\lambda), (s_2, G_2^\lambda), \dots, (s_D, G_D^\lambda)$$

Value Function Approximation: Monte Carlo

- Return G_t is an unbiased, noisy sample of true value $v_\pi(S_t)$
- Can therefore apply supervised learning to "training data":

$$\langle S_1, G_1 \rangle, \langle S_2, G_2 \rangle, ..., \langle S_T, G_T \rangle$$

- For example, using *linear Monte-Carlo policy evaluation*

$$\begin{aligned} \Delta \mathbf{w} &= \alpha(G_t - \hat{v}(S_t, \mathbf{w}))\nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w}) \\ &= \alpha(G_t - \hat{v}(S_t, \mathbf{w}))\mathbf{x}(S_t) \end{aligned}$$

- Monte-Carlo evaluation converges to a local optimum
- Even when using non-linear value function approximation

Prediction - Value Function Approximation: Monte Carlo

Gradient Monte Carlo Algorithm for Estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, S_1, A_1, \ldots, R_T, S_T$ using π
  Loop for each step of episode, $t = 0, 1, \ldots, T-1$:
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[G_t - \hat{v}(S_t,\mathbf{w})\big]\nabla\hat{v}(S_t,\mathbf{w})$

Being the MC target (the return), an unbiased estimator, the gradient-descent version of MC prediction converges to a:
- [linear case] globally optimal approximation of v_π
- [non-linear case] locally optimal approximation of v_π

Prediction - Value Function Approximation: Monte Carlo

Gradient Monte Carlo Algorithm for Estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, S_1, A_1, \ldots, R_T, S_T$ using π
  Loop for each step of episode, $t = 0, 1, \ldots, T-1$:
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[G_t - \hat{v}(S_t,\mathbf{w})\big]\nabla\hat{v}(S_t,\mathbf{w})$

We don't have the same guarantees with TD(0)* or TD(lambda)

* But we are 'close'

Being the MC target (the return), an unbiased estimator, the gradient-descent version of MC prediction converges to a:
- [linear case] globally optimal approximation of v_π
- [non-linear case] locally optimal approximation of v_π

Value Function Approximation: TD(0)

- The TD-target $R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$ is a *biased* sample of true value $v_\pi(S_t)$
- Can still apply supervised learning to "training data":

$$\langle S_1, R_2 + \gamma \hat{v}(S_2, \mathbf{w}) \rangle, \langle S_2, R_3 + \gamma \hat{v}(S_3, \mathbf{w}) \rangle, ..., \langle S_{T-1}, R_T \rangle$$

- For example, using *linear TD(0)*

$$\begin{aligned} \Delta \mathbf{w} &= \alpha(R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w}) \\ &= \alpha \delta \mathbf{x}(S) \end{aligned}$$

- Linear TD(0) converges (close) to global optimum

Prediction - Value Function Approximation: TD(0)

Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
    Initialize S
    Loop for each step of episode:
        Choose $A \sim \pi(\cdot|S)$
        Take action A, observe R, S'
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})\big]\nabla\hat{v}(S,\mathbf{w})$
        $S \leftarrow S'$
    until S is terminal

Why ‘semi-gradient’?

An important observation:

$$\Delta \mathbf{w} = \alpha(v_\pi(S) - \hat{v}(S, \mathbf{w}))\nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w})$$

Target does not depend on w

$$\Delta \mathbf{w} = \alpha(R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w}))\nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w})$$

It depends on w (a typical property of all bootstrapping methods)

They are not true gradient descent method: *they take into account the effect of changing the weight vector w_t on the estimate but ignore its effect on the target*

Value Function Approximation: TD(lambda)

- The λ-return G_t^λ is also a biased sample of true value $v_\pi(s)$
- Can again apply supervised learning to "training data":

$$\left\langle S_1, G_1^\lambda \right\rangle, \left\langle S_2, G_2^\lambda \right\rangle, ..., \left\langle S_{T-1}, G_{T-1}^\lambda \right\rangle$$

- Forward view linear TD(λ)

$$\begin{aligned}\Delta \mathbf{w} &= \alpha(G_t^\lambda - \hat{v}(S_t, \mathbf{w}))\nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w}) \\ &= \alpha(G_t^\lambda - \hat{v}(S_t, \mathbf{w}))\mathbf{x}(S_t)\end{aligned}$$

- Backward view linear TD(λ)

$$\begin{aligned}\delta_t &= R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w}) \\ E_t &= \gamma \lambda E_{t-1} + \mathbf{x}(S_t) \\ \Delta \mathbf{w} &= \alpha \delta_t E_t\end{aligned}$$

Value Function Approximation: TD(lambda)

- The λ-return G_t^λ is also a biased sample of true value $v_\pi(s)$
- Can again apply supervised learning to "training data":

$$\left\langle S_1, G_1^\lambda \right\rangle, \left\langle S_2, G_2^\lambda \right\rangle, ..., \left\langle S_{T-1}, G_{T-1}^\lambda \right\rangle$$

Each 'couple' has info for many states

- Forward view linear TD(λ)

$$\begin{aligned}\Delta \mathbf{w} &= \alpha(G_t^\lambda - \hat{v}(S_t, \mathbf{w}))\nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w}) \\ &= \alpha(G_t^\lambda - \hat{v}(S_t, \mathbf{w}))\mathbf{x}(S_t)\end{aligned}$$

- Backward view linear TD(λ)

$$\begin{aligned}\delta_t &= R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w}) \\ E_t &= \gamma \lambda E_{t-1} + \mathbf{x}(S_t) \\ \Delta \mathbf{w} &= \alpha \delta_t E_t\end{aligned}$$

Prediction - Value Function Approximation: n-step TD

n-step semi-gradient TD for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Algorithm parameters: step size $\alpha > 0$, a positive integer n
Initialize value-function weights $\mathbf{w}$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
All store and access operations (S_t and R_t) can take their index mod $n+1$

Loop for each episode:
  Initialize and store $S_0 \neq$ terminal
  $T \leftarrow \infty$
  Loop for $t = 0, 1, 2, \ldots$:
  | If $t < T$, then:
  | Take an action according to $\pi(\cdot|S_t)$
  | Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
  | If S_{t+1} is terminal, then $T \leftarrow t+1$
  | $\tau \leftarrow t - n + 1$ (τ is the time whose state's estimate is being updated)
  | If $\tau \geq 0$:
  | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
  | If $\tau + n < T$, then: $G \leftarrow G + \gamma^n \hat{v}(S_{\tau+n},\mathbf{w})$ $(G_{\tau:\tau+n})$
  | $\mathbf{w} \leftarrow \mathbf{w} + \alpha\,[G - \hat{v}(S_\tau,\mathbf{w})]\,\nabla\hat{v}(S_\tau,\mathbf{w})$
  Until $\tau = T - 1$

Prediction - Value Function Approximation: convergence

Method	Linear FA	Nonlinear FA
TD(0)	✓ converges	✘ no guarantee, can diverge
TD(λ)	✓ converges for all λ	✘ no guarantee
n-step TD	✓ converges	✘ no guarantee
MC	✓ globally	✓ *locally* (unbiased target avoids bootstrapping issues)

What about features? How to build them?

- Linear learning methods are of great interest:

• convergence properties

• very efficient in terms of both data and computation

...but the overall goodness depends on how the states are represented in terms of the features!

- Adding prior domain knowledge to RL systems is important for feature design (it is usually important, also in other phases of an RL and ML project)

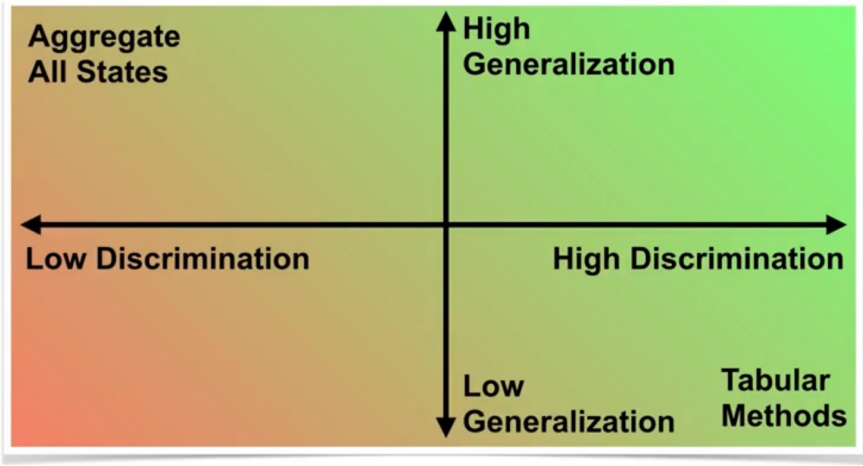
Ie. HVAC system -> room occupancy, hour of the day, month, current temperature...

Ie. Mobile robot -> locations, remaining battery, recent sensor readings...

What about features? How to build them?

In the search of ‘good’ features, we need a good trade-off between:

- Generalization, ie. how features allow the value function approximator to treat similar states as having similar values.
- Discrimination, ie. how features allow the approximator to distinguish states that have different values.



Generalization is important because:

- In large or continuous state spaces, we cannot learn a separate value for every state.

- Instead, the agent must generalize information learned from one state to nearby or similar states.

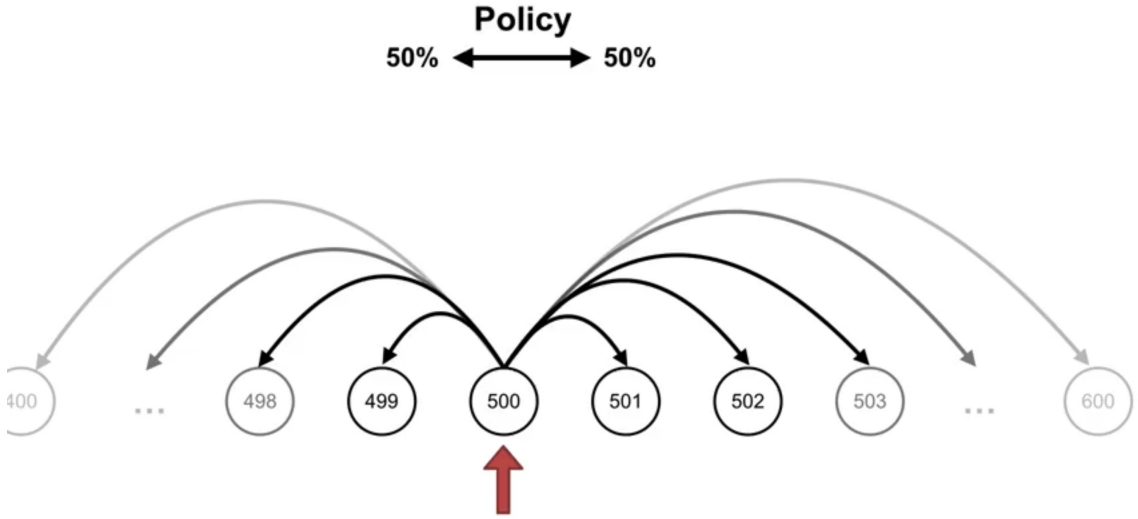
Discrimination is important because:

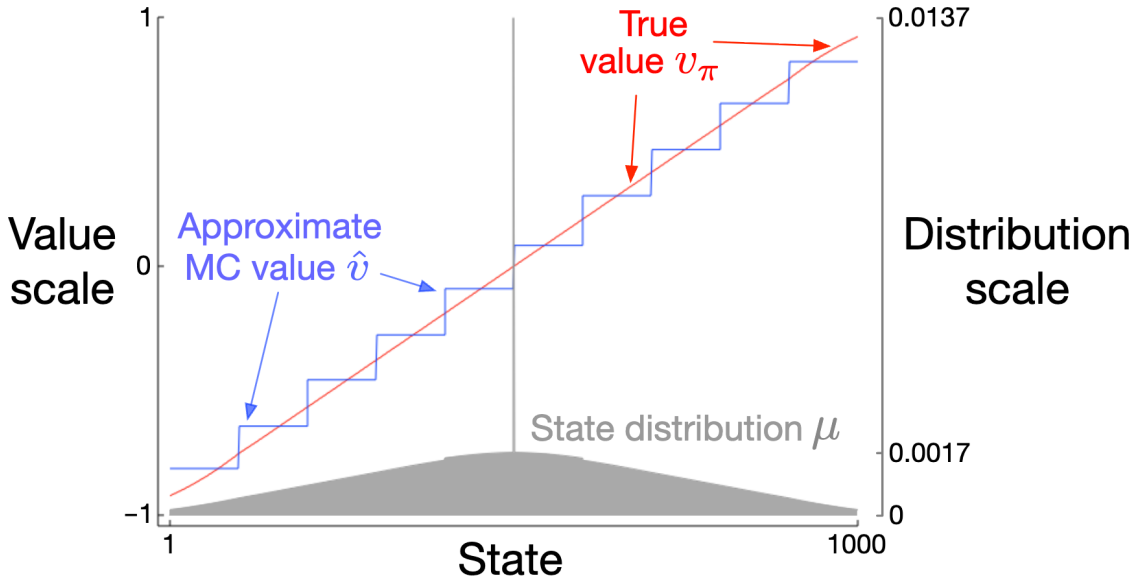
- If two states must be valued differently for optimal behavior, the function approximator must have the capacity to separate them.

- Without discrimination, value estimates will blur together, causing suboptimal policy learning.

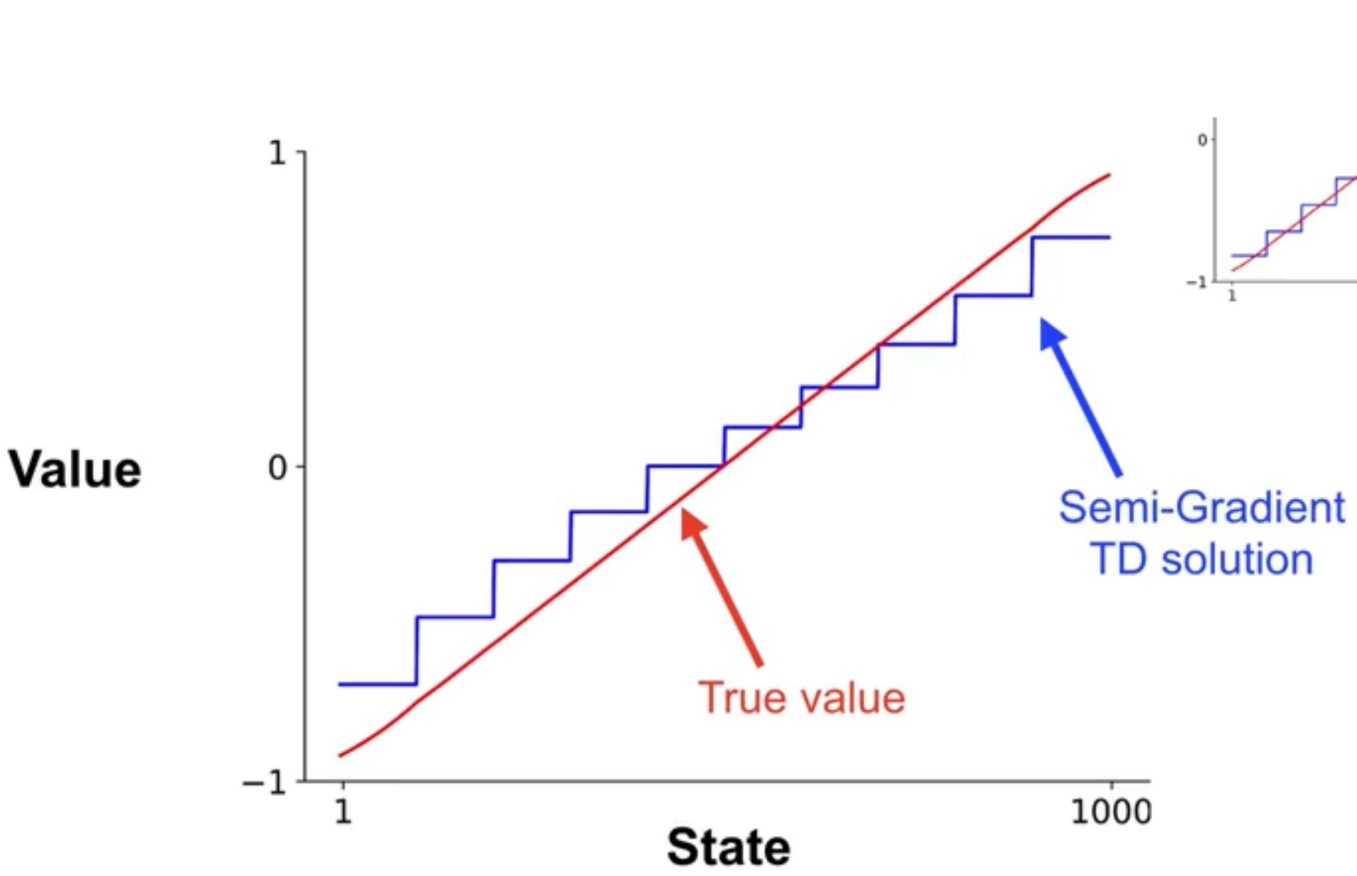
Feature Design #01: state aggregation

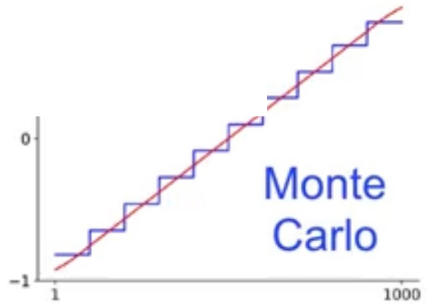
- A simple form of generalizing function approximation in which states are grouped together, with one estimated value (one component of the weight vector **w**) for each group
- Large scale random walk example in the book

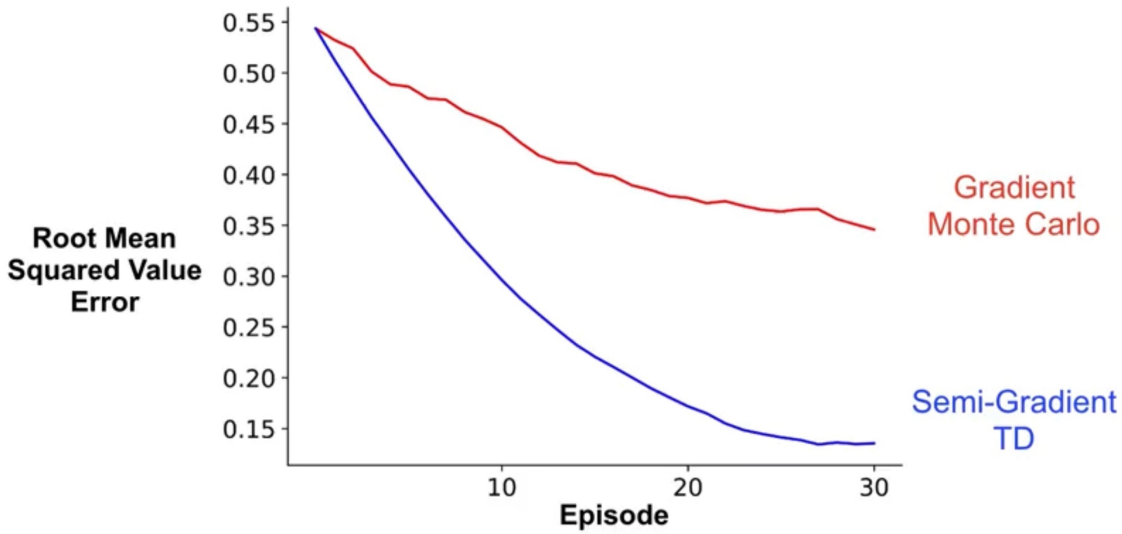




Feature Design #01: state aggregation, large random walk

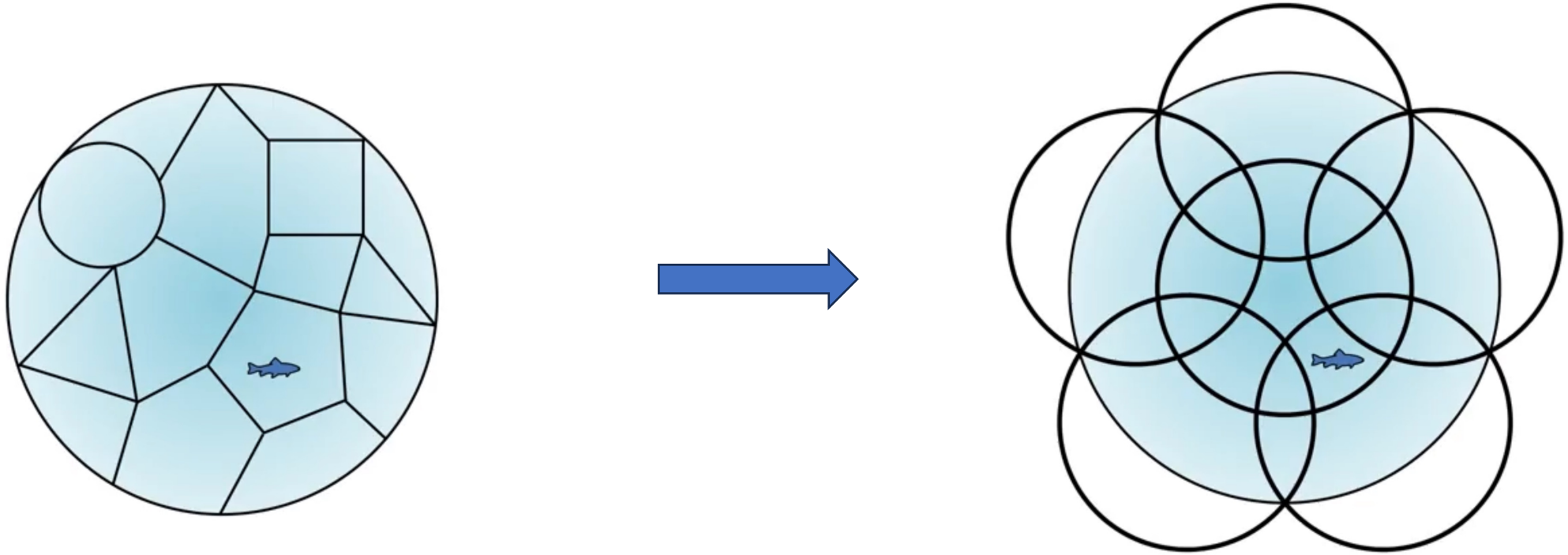






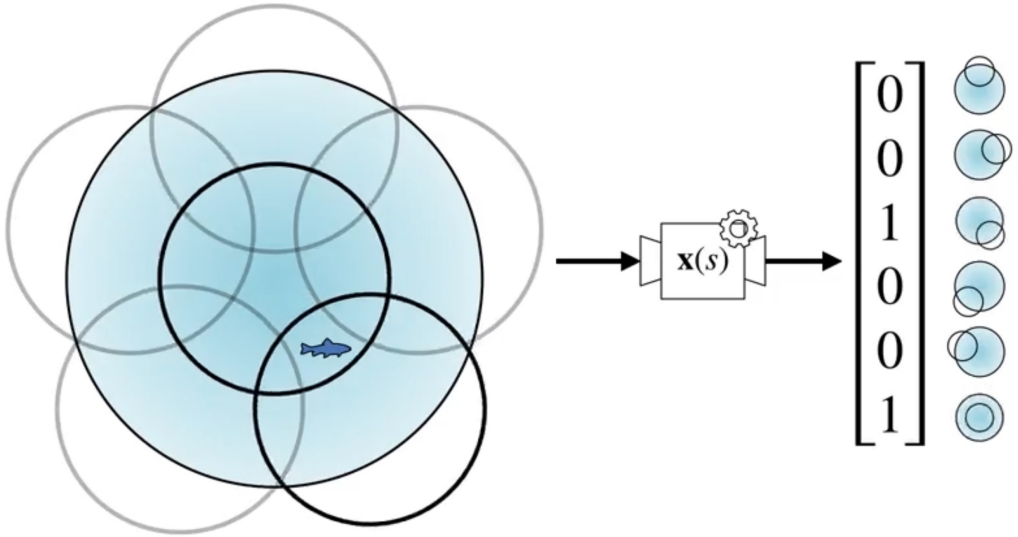
Feature Design #02: coarse coding

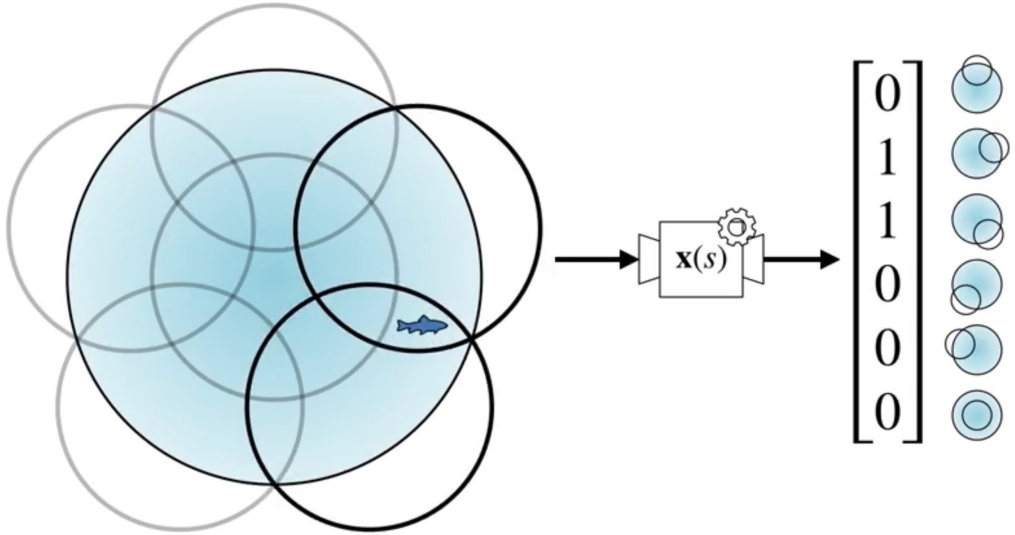
- Coarse coding is a feature-representation method where each feature responds to a broad region of the state space, and many features overlap.



Feature Design #02: coarse coding

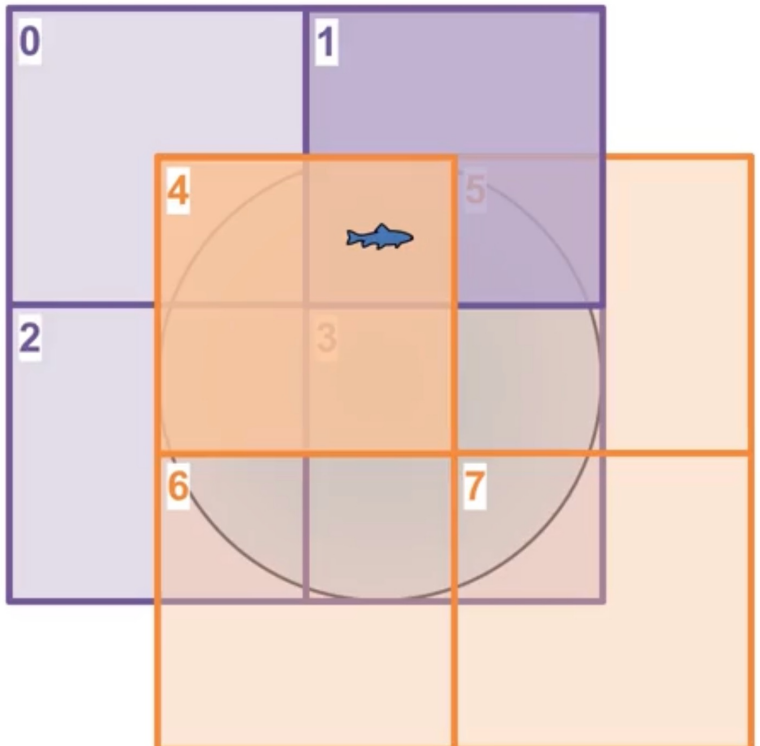
- With coarse coding we have a set of features that each respond to a coarse (wide) region of the state space; any given state activates several of these overlapping features, and the unique pattern of activations represents the state
- Coarse coding uses fewer, broader features, making the representation compact, but because states activate different overlapping subsets of these features, it can still capture fine-grained distinctions.
- Aggregated states behave like one-hot features, while coarse coding is more efficient because each state activates multiple overlapping features instead of just one.





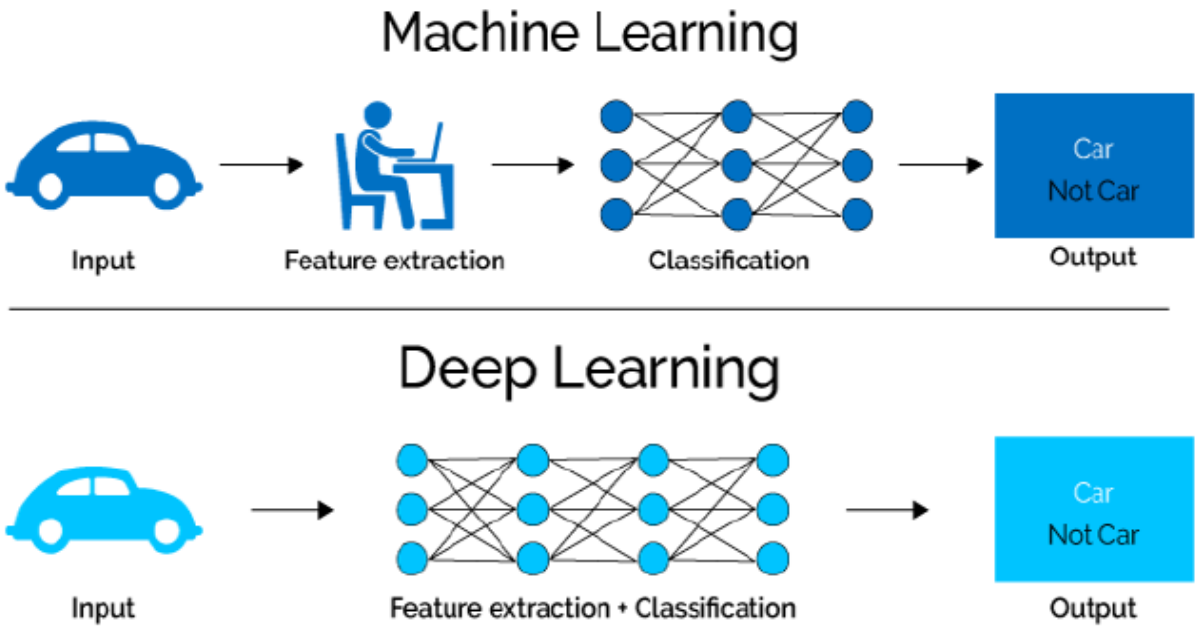
Feature Design #03: tile coding

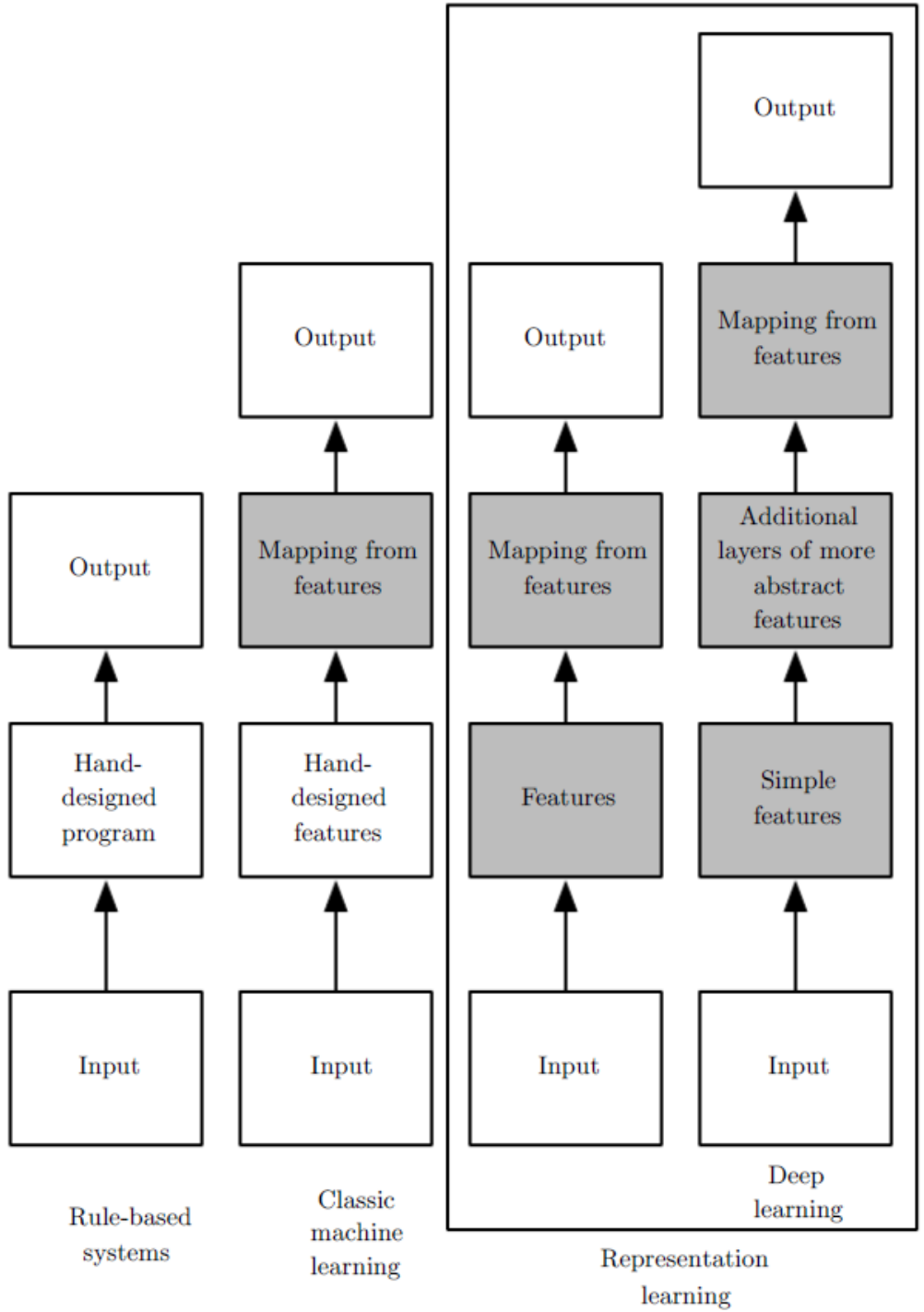
- A special, structured case of coarse coding is tile coding
- Tile coding: rectangular regions arranged in multiple overlapping grids (tilings)
- Very efficient, sparse and easy to use in RL
- Overlap is binary (inside tile = 1, outside = 0)
- Fixed costs at each iteration (we are always updating the same amount of weight)



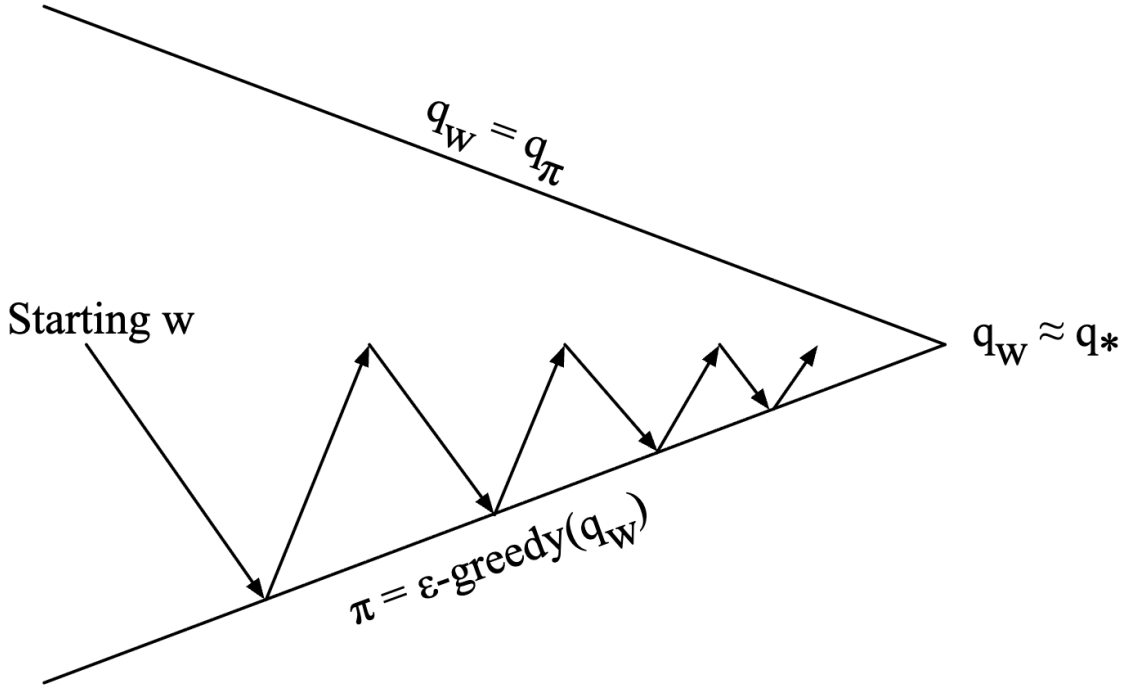
$$\mathbf{w} = \begin{bmatrix} 0.0 \\ 1.5 \\ 1.1 \\ 0.2 \\ 0.5 \\ -0.3 \\ 1.3 \\ -0.7 \end{bmatrix} \quad \mathbf{x}(s) = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{matrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{matrix}$$

With Deep Learning, it is easier!





Control - Even if we are dealing with a new 'framework', we leverage on the algorithms we have already seen!



Policy evaluation Approximate policy evaluation, $\hat{q}(\cdot, \cdot, \mathbf{w}) \approx q_\pi$

Policy improvement ϵ-greedy policy improvement

Control - VFA

- Approximate the action-value function

$$\hat{q}(S, A, \mathbf{w}) \approx q_\pi(S, A)$$

- Minimise mean-squared error between approximate action-value fn $\hat{q}(S, A, \mathbf{w})$ and true action-value fn $q_\pi(S, A)$

$$J(\mathbf{w}) = \mathbb{E}_\pi \left[(q_\pi(S, A) - \hat{q}(S, A, \mathbf{w}))^2 \right]$$

- Use stochastic gradient descent to find a local minimum

$$-\frac{1}{2} \nabla_{\mathbf{w}} J(\mathbf{w}) = (q_\pi(S, A) - \hat{q}(S, A, \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(S, A, \mathbf{w})$$

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

Control - VFA

- Represent state *and* action by a *feature vector*

$$\mathbf{x}(S, A) = \begin{pmatrix} \mathbf{x}_1(S, A) \\ \vdots \\ \mathbf{x}_n(S, A) \end{pmatrix}$$

- Represent action-value fn by linear combination of features

$$\hat{q}(S, A, \mathbf{w}) = \mathbf{x}(S, A)^\top \mathbf{w} = \sum_{j=1}^{n} \mathbf{x}_j(S, A)\mathbf{w}_j$$

- Stochastic gradient descent update

$$\nabla_{\mathbf{w}} \hat{q}(S, A, \mathbf{w}) = \mathbf{x}(S, A)$$

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

Control - VFA

- Like prediction, we must substitute a *target* for $q_\pi(S, A)$
  - For MC, the target is the return G_t

$$\Delta \mathbf{w} = \alpha(G_t - \hat{q}(S_t, A_t, \mathbf{w}))\nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w})$$

  - For TD(0), the target is the TD target $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$

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

  - For forward-view TD(λ), target is the action-value λ-return

$$\Delta \mathbf{w} = \alpha(q_t^\lambda - \hat{q}(S_t, A_t, \mathbf{w}))\nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w})$$

  - For backward-view TD(λ), equivalent update is

$$\delta_t = R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w})$$
$$E_t = \gamma \lambda E_{t-1} + \nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w})$$
$$\Delta \mathbf{w} = \alpha \delta_t E_t$$

Episodic Semi-gradient Sarsa for Estimating $\hat{q} \approx q_*$

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
    $S, A \leftarrow$ initial state and action of episode (e.g., ε-greedy)
    Loop for each step of episode:
        Take action A, observe R, S'
        If S' is terminal:
            $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
            Go to next episode
        Choose A' as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., ε-greedy)
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
        $S \leftarrow S'$
        $A \leftarrow A'$

Episodic Semi-gradient Sarsa for Estimating $\hat{q} \approx q_*$

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
    $S, A \leftarrow$ initial state and action of episode (e.g., ε-greedy)
    Loop for each step of episode:
        Take action A, observe R, S'
        If S' is terminal:
            $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
            Go to next episode
        Choose A' as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., ε-greedy)
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
        $S \leftarrow S'$
        $A \leftarrow A'$

We can also have optimistic value initialization strategies to foster exploration at the early stages of the episode

Episodic Semi-gradient Sarsa for Estimating $\hat{q} \approx q_*$

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  $S, A \leftarrow$ initial state and action of episode (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    If S' is terminal:
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
      Go to next episode
    Choose A' as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., ε-greedy)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big|R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\big|\nabla\hat{q}(S, A, \mathbf{w})$
    $S \leftarrow S'$
    $A \leftarrow A'$

The update of the parameters of q is in fact the improvement step!

Episodic Semi-gradient Sarsa for Estimating $\hat{q} \approx q_*$

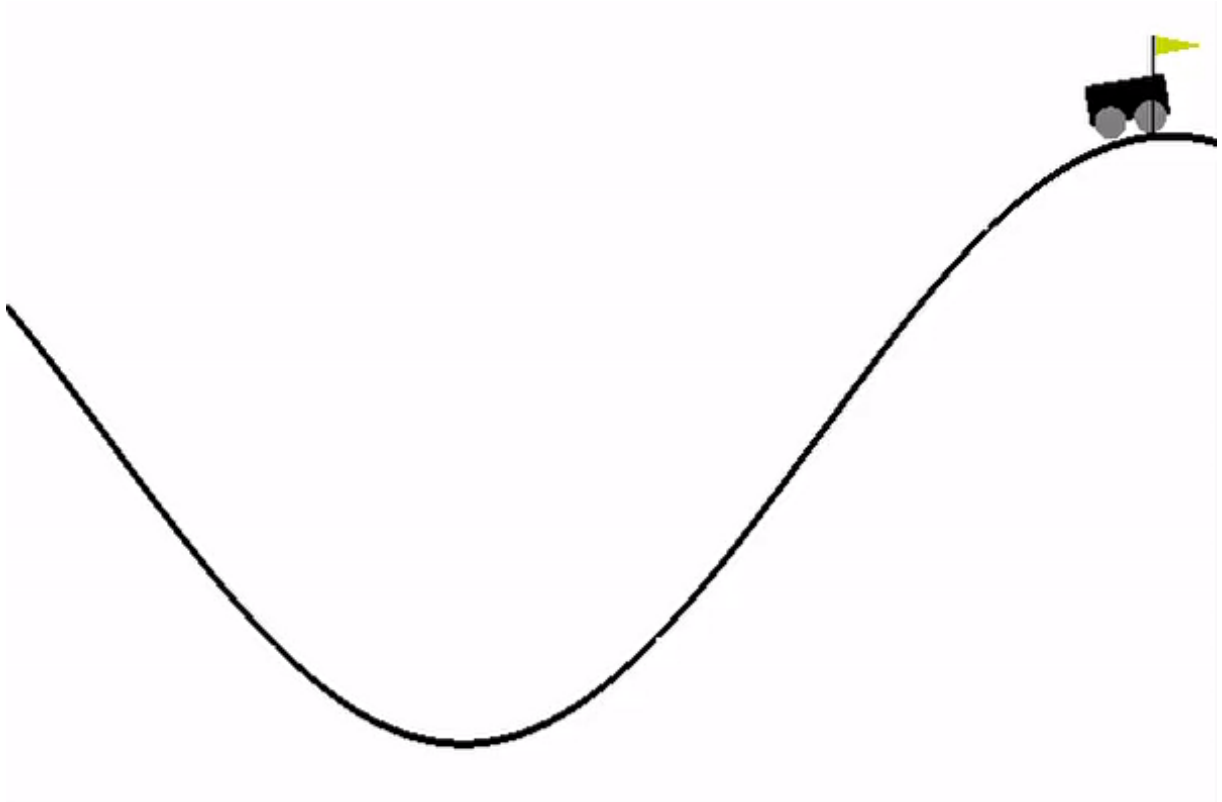
Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  $S, A \leftarrow$ initial state and action of episode (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    If S' is terminal:
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
      Go to next episode
    Choose A' as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., ε-greedy)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
    $S \leftarrow S'$
    $A \leftarrow A'$

On policy

Example: Mountain Car Problem

Example: Mountain Car problem (Example 10.1 of the book)



D. Unzueta https://towardsdatascience.com/reinforcement-learning-applied-to-the-mountain-car-problem-1c4fb16729ba

Example: Mountain Car Problem

Example: Mountain Car problem (Example 10.1 of the book)

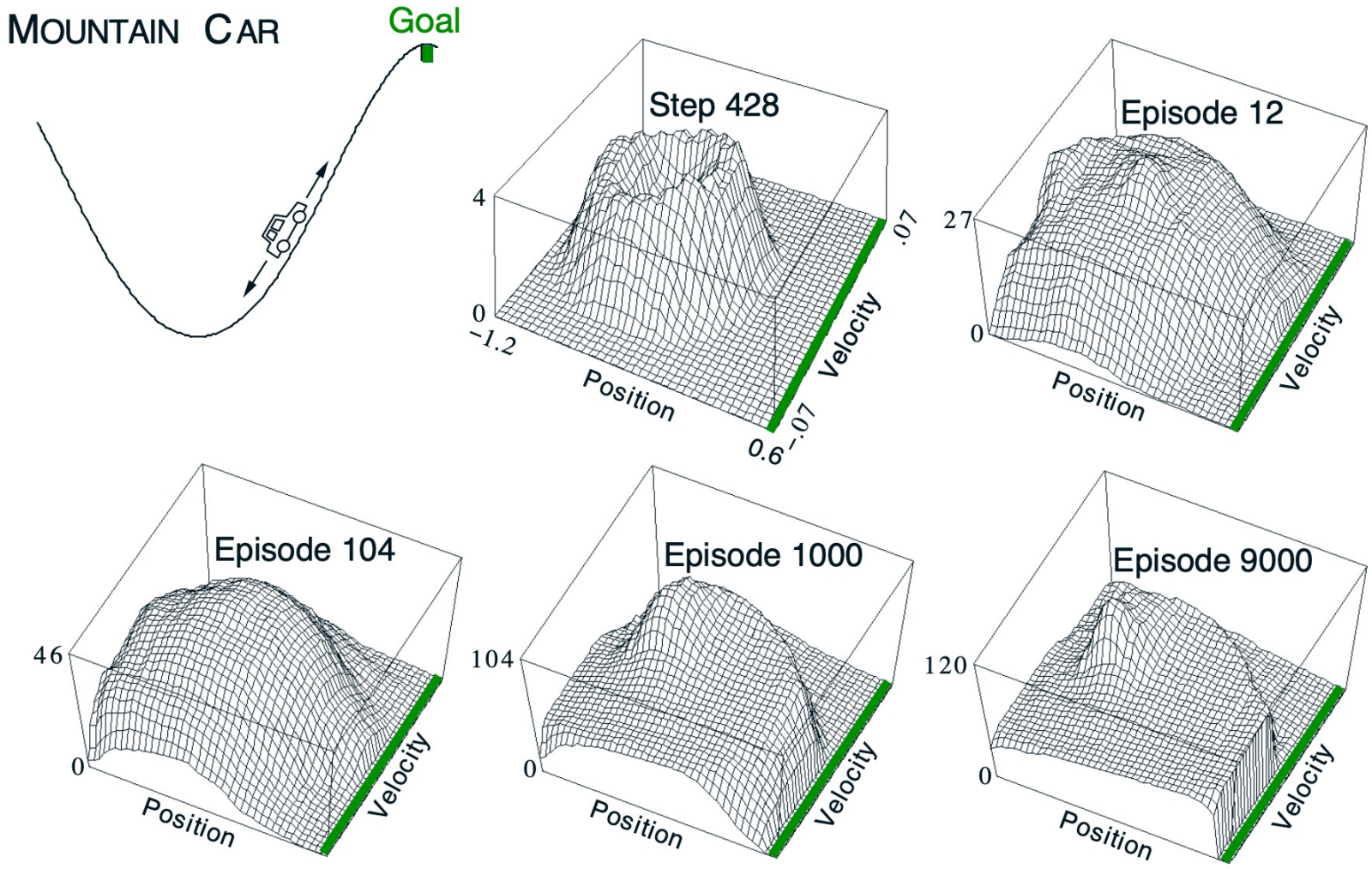


Figure 10.1: The Mountain Car task (upper left panel) and the cost-to-go function $(-\max_a \hat{q}(s, a, \mathbf{w}))$ learned during one run.

Credits

- David Silver ‘Lecture on RL’, UCL 2015
- Image of the course is taken from C. Mahoney ‘Reinforcement Learning’ https://towardsdatascience.com/reinforcement-learning-fda8ff535bb6

AMCO
ARTIFICIAL INTELLIGENCE, MACHINE LEARNING AND CONTROL RESEARCH GROUP

Reinforcement Learning 2025/2026

RL2
Reinforcement Learning Research Lab @ UniPD

Thank you!
Questions?

Lecture #14: Value Function Approximation

Gian Antonio Susto