Model Predictive Control and Reinforcement Learning
– On-Policy Control with Function Approximation –

Joschka Boedecker and Moritz Diehl

University Freiburg

July 29, 2021

Lecture Overview

Acknowledgement

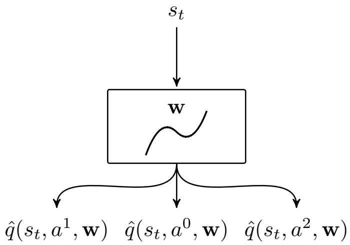
Slide contents are partially based on *Reinforcement Learning: An Introduction* by Sutton and Barto and the Reinforcement Learning lecture by David Silver.

Function Approximation in Reinforcement Learning

- Up to this point, we represented all elements of our RL systems by tables (value functions, models and policies)
- If the state and action spaces are very large or infinite, this is not a feasible solution
- We can apply function approximation to find a more compact representation of RL components and to generalize over states and actions
- Reinforcement Learning with function approximation comes with new issues that do not arise in Supervised Learning – such as non-stationarity, bootstrapping and delayed targets

Function Approximation in Reinforcement Learning

- Here: we estimate value-functions $v_\pi(\cdot)$ and $q_\pi(\cdot,\cdot)$ by function approximators $\hat{v}(\cdot,\mathbf{w})$ and $\hat{q}(\cdot,\cdot,\mathbf{w})$, parameterized by weights $\mathbf{w}$



- But we can also represent models or policies

Function Approximation in Reinforcement Learning

We can use different types of function approximators:

- Linear combinations of features
- Neural networks
- Decision trees
- Gaussian processes
- Nearest neighbor methods
- ...

Here: We focus on differentiable FAs and update the weights via gradient descent.

Function Approximation in Reinforcement Learning

We want to update our weights w.r.t. the *Mean Squared Value Error* of our prediction:

$$\begin{aligned}\mathbf{w}_{t+1} &= \mathbf{w}_t - \frac{1}{2}\alpha\nabla[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)]^2 \\ &= \mathbf{w}_t + \alpha[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)]\nabla\hat{v}(S_t, \mathbf{w}_t)\end{aligned}$$

However, we don't have $v_\pi(S_t)$.

Gradient MC

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[G_t - \hat{v}(S_t, \mathbf{w})]\nabla\hat{v}(S_t, \mathbf{w})$$

Semi-gradient TD(0)

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w})]\nabla\hat{v}(S_t, \mathbf{w})$$

Why are bootstrapping methods, defined this way, called *semi-gradient methods*?

Function Approximation in Reinforcement Learning

Gradient MC

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[G_t - \hat{v}(S_t, \mathbf{w})]\nabla\hat{v}(S_t, \mathbf{w})$$

Semi-gradient TD(0)

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w})]\nabla\hat{v}(S_t, \mathbf{w})$$

Why are bootstrapping methods, defined this way, called *semi-gradient methods*?
They take into account the effects of changing $\mathbf{w}$ w.r.t. the prediction, but not w.r.t. the target!

Linear Methods

- Represent state s by feature vector $\mathbf{x}(s) = (x^1(s), x^2(s), \dots, x^d(s))^\top$
- These features can also be non-linear functions/combinations of state dimensions
- Linear methods approximate the value function by a linear combination of these features

$$\hat{v}(s, \mathbf{w}) = \mathbf{w}^\top \mathbf{x}(s) = \sum_{i=1}^{d} w^i x^i(s)$$

- Therefore, $\nabla_{\mathbf{w}} \hat{v}(s, \mathbf{w}) = \mathbf{x}(s)$
- Gradient MC prediction converges under linear FA
- On-policy linear semi-gradient TD(0) is stable
- Unfortunately, this does not hold for non-linear FA

Fixed point of on-policy linear semi-gradient TD

- The update at each time step t is:

$$\begin{aligned}\mathbf{w}_{t+1} &= \mathbf{w}_t + \alpha\left(R_{t+1} + \gamma \mathbf{w}_t^\top \mathbf{x}_{t+1} - \mathbf{w}_t^\top \mathbf{x}_t\right)\mathbf{x}_t \\ &= \mathbf{w}_t + \alpha\left(R_{t+1}\mathbf{x}_t - \mathbf{x}_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^\top \mathbf{w}_t\right)\end{aligned}$$

- The expected next weight vector can thus be written:

$$\mathbb{E}[\mathbf{w}_{t+1}|\mathbf{w}_t] = \mathbf{w}_t + \alpha(\mathbf{b} - \mathbf{A}\mathbf{w}_t),$$

  where $\mathbf{b} = \mathbb{E}[R_{t+1}\mathbf{x}_t]$ and $\mathbf{A} = \mathbb{E}[\mathbf{x}_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^\top]$

- If the system converges, it has to converge to the *fixed point*:

$$\mathbf{w}_{\text{TD}} = \mathbf{A}^{-1}\mathbf{b}$$

Least Squares TD

- Recall the *fixed point*: $\mathbf{w}_{\text{TD}} = \mathbf{A}^{-1}\mathbf{b}$
- Why don't we calculate $\mathbf{A}$ and $\mathbf{b}$ directly?
- LSTD does exactly that:

$$\hat{\mathbf{A}}_t = \sum_{k=0}^{t-1} \mathbf{x}_k(\mathbf{x}_k - \gamma\mathbf{x}_{k+1})^\top + \varepsilon\mathbf{I} \text{ and } \hat{\mathbf{b}}_t = \sum_{k=0}^{t-1} R_{k+1}\mathbf{x}_k$$

- LSTD is more data-efficient, but also has quadratic runtime (compared to semi-gradient TD(0) – which is linear)

Least Squares TD

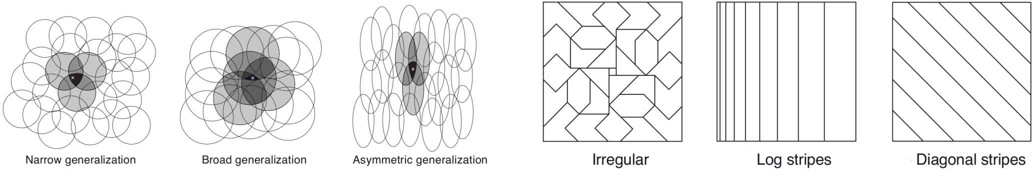
LSTD for estimating $\hat{v} = \mathbf{w}^\top \mathbf{x}(\cdot) \approx v_\pi$ ($O(d^2)$ version)

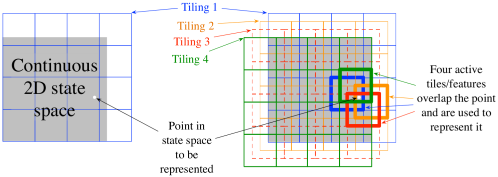
Input: feature representation $\mathbf{x} : \mathcal{S}^+ \rightarrow \mathbb{R}^d$ such that $\mathbf{x}(terminal) = \mathbf{0}$
Algorithm parameter: small $\varepsilon > 0$

$\widehat{\mathbf{A}^{-1}} \leftarrow \varepsilon^{-1}\mathbf{I}$ — A $d \times d$ matrix
$\widehat{\mathbf{b}} \leftarrow \mathbf{0}$ — A d-dimensional vector
Loop for each episode:
  Initialize S; $\mathbf{x} \leftarrow \mathbf{x}(S)$
  Loop for each step of episode:
    Choose and take action $A \sim \pi(\cdot|S)$, observe R, S'; $\mathbf{x}' \leftarrow \mathbf{x}(S')$
    $\mathbf{v} \leftarrow \widehat{\mathbf{A}^{-1}}^\top (\mathbf{x} - \gamma \mathbf{x}')$
    $\widehat{\mathbf{A}^{-1}} \leftarrow \widehat{\mathbf{A}^{-1}} - (\widehat{\mathbf{A}^{-1}}\mathbf{x})\mathbf{v}^\top / (1 + \mathbf{v}^\top \mathbf{x})$
    $\widehat{\mathbf{b}} \leftarrow \widehat{\mathbf{b}} + R\mathbf{x}$
    $\mathbf{w} \leftarrow \widehat{\mathbf{A}^{-1}}\widehat{\mathbf{b}}$
    $S \leftarrow S'$; $\mathbf{x} \leftarrow \mathbf{x}'$
  until S' is terminal

Coarse Coding

Divide the state space in circles/tiles/shapes and check in which some state is inside. This is a binary representation of the location of a state and leads to generalization.





Memory-based Function Approximation

- So far, we discussed the parametric approach to represent value functions
- Memory-based methods simply store collected examples and their values in memory and retrieve samples in order to estimate the value for a query state
- The simplest examples are the nearest neighbor method or the weighted average method over a subset of nearest neighbors
- Similarity between states can be defined by a *kernel* $k(s, s')$
- The value of a query state then is

$$\hat{v}(s, \mathcal{D}) = \sum_{s' \in \mathcal{D}} k(s, s') g(s'),$$

where $g(s')$ is the stored value of s'

On-policy Control with Function Approximation

- Again, up to this point we discussed Policy Evaluation based on state value functions
- In order to apply FA in control, we parameterize the action-value function

Semi-gradient SARSA

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

Semi-gradient SARSA

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)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
    $S \leftarrow S'$
    $A \leftarrow A'$

Semi-gradient SARSA

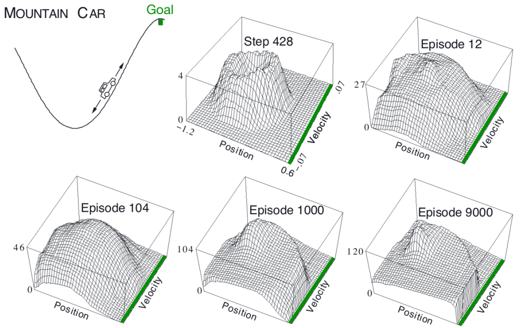


Figure 10.1: The Mountain Car task (upper left panel) and the cost-to-go function ($-\max_a \hat{q}(s, a, \mathbf{w})$) learned during one run.

Semi-gradient SARSA

Mountain Car

Steps per episode

log scale

averaged over 100 runs