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

Joschka Boedecker and Moritz Diehl

University Freiburg

October 6, 2023

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.

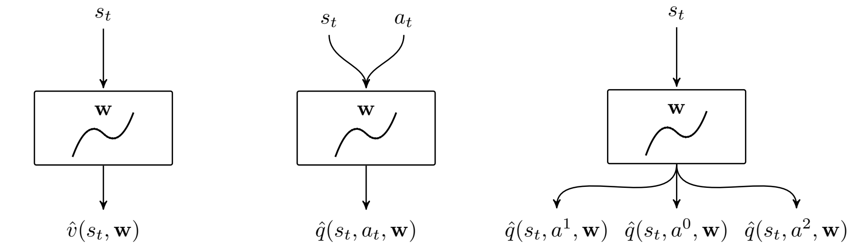
Lecture Overview

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 $v_\pi(S_t)$.

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)

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

Lecture Overview

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

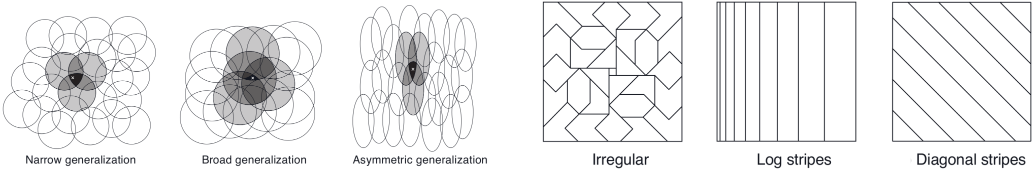
LSTD for estimating $\hat{v} = \mathbf{w}^\top \mathbf{x}(\cdot) \approx v_\pi$ ($O(d^2)$ version)

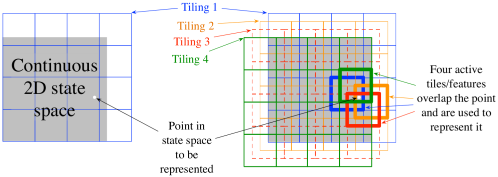
Input: feature representation $\mathbf{x} : \mathcal{S}^+ \to \mathbb{R}^d$ such that $\mathbf{x}(terminal) = \mathbf{0}$
Algorithm parameter: small $\varepsilon > 0$

$\widehat{\mathbf{A}^{-1}} \leftarrow \varepsilon^{-1}\mathbf{I}$ — 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.





Lecture Overview

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

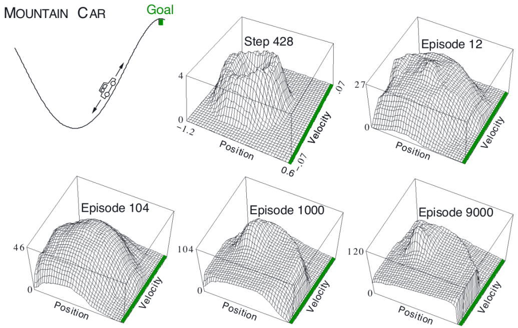
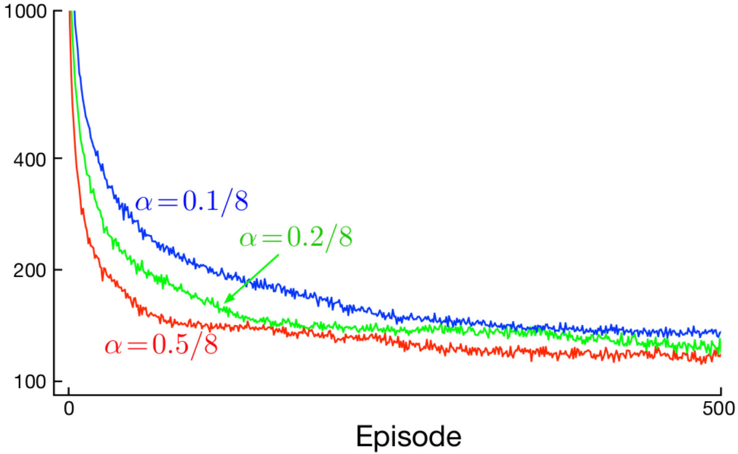


Figure 10.1: The Mountain Car task (upper left panel) and the cost-to-go function ($-\max_a \hat{q}(s, a, \mathbf{w})$) learned during one run.

Semi-gradient SARSA

Mountain Car
Steps per episode
log scale
averaged over 100 runs



Lecture Overview

Off-policy Learning

- We want to learn the optimal policy, but we have to account for the problem of *maintaining exploration*
- We call the (optimal) policy to be learned the *target policy* π and the policy used to generate behaviour the *behaviour policy* b
- We say that learning is from data *off* the target policy – thus, those methods are referred to as *off-policy learning*

Importance Sampling

- Weight returns according to the relative probability of target and behaviour policy
- Define state-transition probabilities $p(s'|s,a)$ as $p(s'|s,a) = \mathsf{Pr}\{S_t = s'|S_{t-1} = s, A_{t-1} = a\} = \sum_{r\in\mathcal{R}} p(s',r|s,a)$
- The probability of the subsequent trajectory under any policy π, starting in S_t, then is:

$$\begin{aligned} &\mathsf{Pr}\{A_t, S_{t+1}, A_{t+1}, \dots S_T|S_t, A_{t:T-1} \sim \pi\} \\ &= \pi(A_t|S_t)p(S_{t+1}|S_t, A_t)\pi(A_{t+1}|S_{t+1}) \cdots p(S_T|S_{T-1}, A_{T-1}) \\ &= \prod_{k=t}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k, A_k) \end{aligned}$$

Importance Sampling

The relative probability therefore is:

Definition: Importance Sampling Ratio

$$\rho_{t:T-1} = \frac{\prod_{k=t}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k, A_k)}{\prod_{k=t}^{T-1} b(A_k|S_k)p(S_{k+1}|S_k, A_k)} = \frac{\prod_{k=t}^{T-1} \pi(A_k|S_k)}{\prod_{k=t}^{T-1} b(A_k|S_k)}$$

The expectation of the returns by b is $\mathbb{E}[G_t|S_t = s] = v_b(s)$. However, we want to estimate the expectation under π. Given the importance sampling ratio, we can transform the MC returns by b to yield the expectation under π:

$$\mathbb{E}[\rho_{t:T-1} G_t|S_t = s] = v_\pi(s).$$

Importance Sampling can come with a vast increase in variance.

Off-policy MC Prediction and Semi-gradient TD(0)

To use importance sampling with function approximation, replace the update to an array to an update to weight vector $\mathbf{w}$, and correct it with the importance sampling weight.

Off-policy MC Prediction

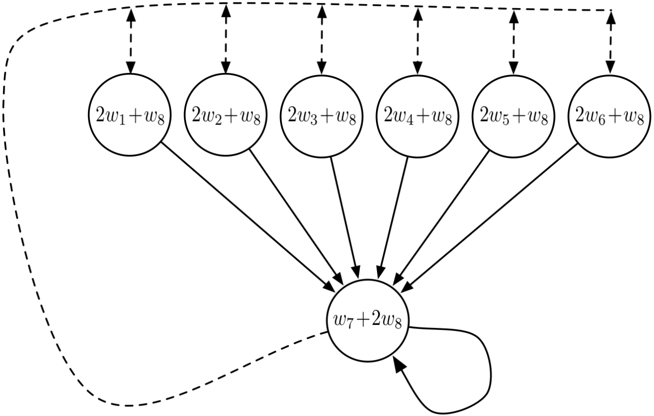
$\mathbf{w} \leftarrow \mathbf{w} + \alpha \rho_{t:T-1} [G_t - \hat{v}(S_t, \mathbf{w})] \nabla \hat{v}(S_t, \mathbf{w})$

Semi-gradient Off-policy TD(0)

$\mathbf{w} \leftarrow \mathbf{w} + \alpha \rho_t \delta_t \nabla \hat{v}(S_t, \mathbf{w})$
where $\delta_t = R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w})$

Lecture Overview

Baird's Counterexample



$\pi(\text{solid}|\cdot) = 1$

$b(\text{dashed}|\cdot) = 6/7$
$b(\text{solid}|\cdot) = 1/7$

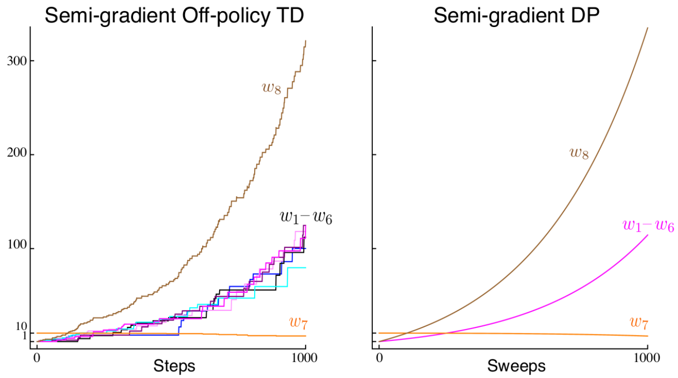
$\gamma = 0.99$

The reward is 0 for all transitions, hence $v_\pi(s) = 0$. This could be exactly approximated by $\mathbf{w} = \mathbf{0}$.

Baird's Counterexample

Semi-gradient DP

$$\mathbf{w} \leftarrow \mathbf{w} + \frac{\alpha}{|\mathcal{S}|} \sum_{s \in \mathcal{S}} \left(\mathbb{E}[R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) | S_t = s] - \hat{v}(s, \mathbf{w}) \right) \nabla \hat{v}(s, \mathbf{w})$$



The Deadly Triad

The combination of

- Function Approximation,
- Bootstrapping and
- Off-policy Learning

is known as the *Deadly Triad*, since it can lead to stability issues and divergence.

Lecture Overview

Neural Fitted-Q Iteration (NFQ) [Riedmiller 2005]

- Model-free off-policy RL algorithm that works on continuous state and discrete action spaces
- Q-function is represented by a multi-layer perceptron
- One of the first approaches that combined RL with ANNs, predecessor of DQN

Neural Fitted-Q Iteration (NFQ) [Riedmiller 2005]

for *iteration* $i = 1, .., N$ **do**

  sample trajectory with ϵ-greedy exploration and add to memory D

  initialize network weights randomly

  generate pattern set $P = \{(x_j, y_j) | j = 1..|D|\}$ with

$$x_j = (s_j, a_j) \text{ and } y_j = \begin{cases} r_j & \text{if } s_j \text{ is terminal} \\ r_j + \gamma \max_{a'} Q(s_{j+1}, a', \mathbf{w}_i) & \text{else} \end{cases}$$

  for *iteration* $k = 1, .., K$ **do**

    Fit weights according to:

$$L(\mathbf{w}_i) = \frac{1}{|D|} \sum_{j=1}^{|D|} (y_j - Q(x_j, \mathbf{w}_i))^2$$

  end

end

Algorithm 1: NFQ

Deep Q-Networks (DQN)

DQN provides a stable solution to deep RL:

- Use experience replay (as in NFQ)
- Sample minibatches (as opposed to Full Batch in NFQ)
- Freeze target Q-networks (no target networks in NFQ)
- Optional: Clip rewards or normalize network adaptively to sensible range

Deep Q-Networks: Experience Replay

To remove correlations, build data set from agent's own experience

- Take action a_t according to ϵ-greedy policy
- Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay memory D
- Sample random mini-batch of transitions (s, a, r, s') from D
- Optimize MSE between Q-network and Q-learning targets, e.g.

$$L(\mathbf{w}) = \mathbb{E}_{s,a,r,s' \sim D}\left[\left(r + \gamma \max_{a'} Q(s', a', \mathbf{w}) - Q(s, a, \mathbf{w})\right)^2\right]$$

Deep Q-Networks: Target Networks

To avoid oscillations, fix parameters used in Q-learning target

- Compute Q-learning targets w.r.t. old, fixed parameters $\mathbf{w}^-$

$$r + \gamma \arg\max_{a'} Q(s', a', \mathbf{w}^-)$$

- Optimize MSE between Q-network and Q-learning targets

$$L(\mathbf{w}) = \mathbb{E}_{s,a,r,s' \sim D}\left[\left(r + \gamma \max_{a'} Q(s', a', \mathbf{w}^-) - Q(s, a, \mathbf{w})\right)^2\right]$$

- Periodically update fixed parameters $\mathbf{w}^- \leftarrow \mathbf{w}$
  - hard update: update target network every N steps
  - slow update: slowly update weights of target network every step by

$$\mathbf{w}^- \leftarrow (1 - \tau)\mathbf{w}^- + \tau \mathbf{w}$$

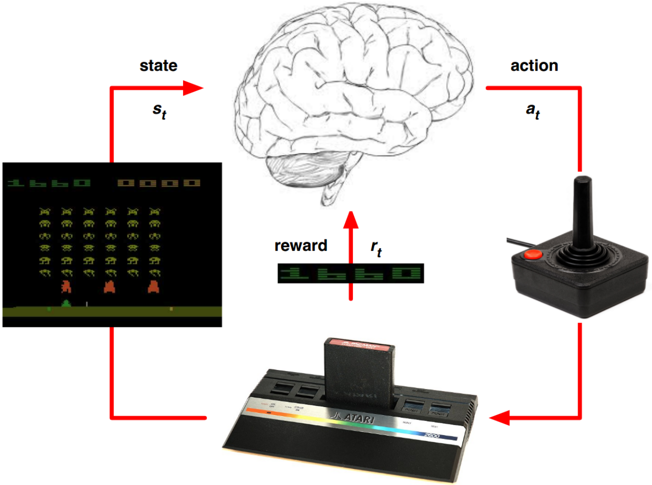
Deep Q-Networks (DQN)

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights
for *episode* $i = 1, .., M$ **do**
    for $t = 1, .., T$ **do**
        select action a_t ϵ-greedily
        Store transition (s_t, a_t, s_{t+1}, r_t) in D
        Sample minibatch of transitions (s_j, a_j, r_j, s_{j+1}) from D
        Set $y_j = \begin{cases} r_j & \text{if } s_{j+1} \text{ is terminal} \\ r_j + \gamma \max_{a'} Q(s_{j+1}, a', \mathbf{w}^-) & \text{else} \end{cases}$
        Update the parameters of Q according to:

$$\nabla \mathbf{w}_i L_i(\mathbf{w}_i) = \mathbb{E}_{s,a,s,r\sim D}[(y_j - Q(s, a, \mathbf{w}_i))\nabla_{\mathbf{w}_i} Q(s, a, \mathbf{w}_i)]$$

        Update target network
    end
end

Deep Q-Networks: Reinforcement Learning in Atari



Deep Q-Networks: Reinforcement Learning in Atari

- End-to-end learning of values $Q(s,a)$ from pixels s
- Input state s is a stack of raw pixels from the last 4 frames
- Output is $Q(s,a)$ for 18 joystick/button positions
- Reward is change in score for that step

How much does DQN help?

	Q-Learning	Q-Learning + Target Q	Q-Learning + Replay	DQN Q-learning + Replay + Target Q
Breakout	3	10	241	**317**
Enduro	29	142	831	**1006**
River Raid	1453	2868	4103	**7447**
Seaquest	276	1003	831	**2894**
Space Invaders	302	373	826	**1089**