# Reinforcement Learning
# Lecture 11: Recap Session [1]

Lecturer: Prof. Dr. Mathias Niepert

Institute for Parallel and Distributed Systems
Machine Learning and Simulation Lab

University of Stuttgart
Germany

imprs-is

July 13, 2023

[1] Many slides adapted from R. Sutton's course, D. Silver's course as well as previous RL courses given at U. of Stuttgart by J. Mainprice, D. Hennes, M. Toussaint, H. Ngo, and V. Ngo.

# Outline

# Evaluation

https://evasysw.uni-stuttgart.de/evasys/online.php?p=HP7FT

https://evasysw.uni-stuttgart.de/evasys/online.php?p=QSEE5

# Study Suggestions

# What you need to study

The exam will cover topics in lectures 1-9

1. Multi-armed bandits; chapters: 2.1, 2.2, and 2.4 (+ softmax policy)
2. MDPs (definition, values function, etc.); chapters: 3.1-3.6
3. Policy improvement with dynamic programming; chapters 4.1-4.4
4. Monte-Carlo methods; chapter 5.1-5.7
5. Temporal difference methods; 6.1, 6.2, and 6.4-6.6
6. Planning and Learning; chapters: 8.1-8.5
7. Function approximation; chapter: 9.1-9.4, 9.5.4
8. n-step bootstrapping (no eligibility traces!); chapters: 7.1-7.3
9. Policy gradient methods; chapters: 13.1-13.6

# General Remarks

- There will be no need to memorize or write pseudo-code (but pseudo-code typically helps understanding a method)
- You should memorize the core formulas corresponding to the various algorithms such as Bellman equation, MC, TD, etc.
- You should memorize backup diagrams of these formulas
- You may only use pen and scratch paper – no other materials (no textbooks, script, calculators, or mobiles) are allowed

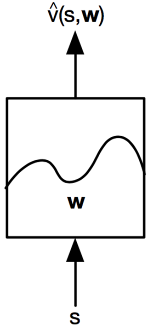
# Value function approximation

# Idea of value function approximation

- Parameterized functional form, with weights $\boldsymbol{w} \in \mathbb{R}^d$:

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

- Generally, much less weights than states $d \ll |\mathcal{S}|$
  - obvious for continuous state spaces
  - changing single weight, changes value estimate of many states
  - when one state is updated, change *generalizes* to many states
- Update $\boldsymbol{w}$ with MC or TD learning



# Function approximators for RL

- Differentiable function approximators, e.g.:
  - Linear combination of features
  - Neural networks
- RL specific problems:
  - non-stationary
  - non-iid data
  - bootstrapping
  - delayed targets

---

Random variables are *independent and identically distributed* (iid) if they each have the same probability distribution and are mutually independent

# Stochastic Gradient Descent (SGD)

- Approximate value function $\hat{v}(s, \boldsymbol{w})$
  - differentiable for all $s \in \mathcal{S}$
- Weight vector $\boldsymbol{w} = (w_1, w_2, \dots, w_d)^\top$
  - $\boldsymbol{w}_t$ weight vector at time $t = 0, 1, 2, \dots$
- *Gradient* of $f(\boldsymbol{w})$: $\boldsymbol{\nabla} f(\boldsymbol{w}) = \left(\frac{\partial f(\boldsymbol{w})}{\partial w_1}, \frac{\partial f(\boldsymbol{w})}{\partial w_2}, \dots, \frac{\partial f(\boldsymbol{w})}{\partial w_d}\right)^\top$
- Do gradient descent by sampling additive parts of the full gradient (i.e., each state consecutively)
- We can compute our update over smaller sets of inputs

# Stochastic Gradient Descent (SGD) (part 2)

- When we **approximate** the gradient
- For example

$$\mathcal{L}(\boldsymbol{w}) = \sum_{n=1}^{N} \mathcal{L}_n(\boldsymbol{w})$$

  where $\boldsymbol{w}$ are weights.
- In machine learning

$$\mathcal{L}(\boldsymbol{w}) = -\sum_{n=1}^{N} \log p(y_n \mid \boldsymbol{x}_n, \boldsymbol{w})$$

  where $\boldsymbol{x}_n \in \mathbb{R}^D$ are training *inputs* and $y_n$ are the training *targets*

- The corresponding update

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t - \alpha_t \sum_{n=1}^{N} (\boldsymbol{\nabla} \mathcal{L}_n)(\boldsymbol{w}_t)$$

- Often the gradient is too difficult to compute (CPU/GPU expensive)
- **Mini-batch**: random subset
  a Large: accurate but costly
  b Small: noisy but cheep

# Stochastic Gradient Descent (SGD) (part 3)

- *Mean squared Value Error*: $\mathcal{L}(\boldsymbol{w}) = \sum_{s \in S} \mu(s) \left[v_\pi(s) - \hat{v}(s, \boldsymbol{w})\right]^2$
- Adjust $\boldsymbol{w}$ to reduce the error on sample $S_t \mapsto v_\pi(S_t)$:

$$\begin{aligned}
\boldsymbol{w}_{t+1} &= \boldsymbol{w}_t - \frac{1}{2}\alpha_t(\boldsymbol{\nabla}\mathcal{L}_t)(\boldsymbol{w}_t) \\
&= \boldsymbol{w}_t - \frac{1}{2}\alpha_t \boldsymbol{\nabla} \underbrace{\left[v_\pi(S_t) - \hat{v}(S_t, \boldsymbol{w}_t)\right]^2}_{\textbf{squared sample error}} \\
&= \boldsymbol{w}_t + \alpha_t[v_\pi(S_t) - \hat{v}(S_t, \boldsymbol{w}_t)]\nabla\hat{v}(S_t, \boldsymbol{w})
\end{aligned}$$

- $\alpha_t$ is a step size parameter
- Why not use $\alpha = 1$, thus eliminating full error on sample?

# Linear methods

- Special case where $\hat{v}(\cdot, \boldsymbol{w})$ is *linear* in the weights
- **Feature vector** $\boldsymbol{x}(s)$ represents state $s$:

$$\boldsymbol{x}(s) = \left[x_1(s), \quad x_2(s), \quad \dots, \quad x_d(s)\right]^\top$$

- Each component of $\boldsymbol{x}$ is a feature, examples:
  - distance of robot to landmarks
  - piece on a specific location on a chess board
- Value function is represented as a linear combination of features $\boldsymbol{x}(s)$:

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

- Gradient is simply $\nabla \hat{v}(s, \boldsymbol{w}) = \boldsymbol{x}(s)$

# Prediction

# Prediction with function approximation

- We assumed the true value function $v_\pi(S_t)$ is known
- Substitute target $U_t$ for $v_\pi(s)$

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha[U_t - \hat{v}(S_t, \boldsymbol{w}_t)]\nabla\hat{v}(S_t, \boldsymbol{w}_t)$$

- $U_t$ might be a noisy or bootstrapped approximation of the true value
- **Monte Carlo:** $U_t = G_t$
- **TD(0):** $U_t = R_{t+1} + \gamma\hat{v}(S_{t+1}, \boldsymbol{w}_t)$
- **TD($\lambda$):** $U_t = G_t^\lambda$

# Monte–Carlo with function approximation

- Target is unbiased by definition:

$$\mathbb{E}[U_t|S_t = s] = \mathbb{E}[G_t|S_t = s] = v_\pi(S_t)$$

- Training data:

$$\mathcal{D} = \{(S_1, G_1),\ (S_2, G_2),\ \ldots,\ (S_{T-1}, G_{T-1}),\ (S_T, 0)\}$$

- Using SGD, $\boldsymbol{w}$ is guaranteed to converge to a *local optimum*
- MC prediction exhibits local convergence with linear and non-linear function approximation
- SGD update for sample $S_t \mapsto G_t$:

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

# Gradient Monte Carlo Algorithm

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

Input: the policy $\pi$ 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 $\pi$
  Loop for each step of episode, $t = 0, 1, \ldots, T-1$:
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[G_t - \hat{v}(S_t,\mathbf{w})\big]\nabla\hat{v}(S_t,\mathbf{w})$

# TD with function approximation

- TD-target $U_t = R_{t+1} + \gamma \hat{v}(S_{t+1}, \boldsymbol{w})$ is *biased* sample of the true value $v_\pi(S_t)$
- Training data:

$$\mathcal{D} = \{(S_1, R_2 + \gamma \hat{v}(S_2, \boldsymbol{w})),\ (S_2, R_3 + \gamma \hat{v}(S_3, \boldsymbol{w})),\ \ldots, (S_{T-1}, R_T)\}$$

- SGD update for sample $S_t \mapsto G_t$:

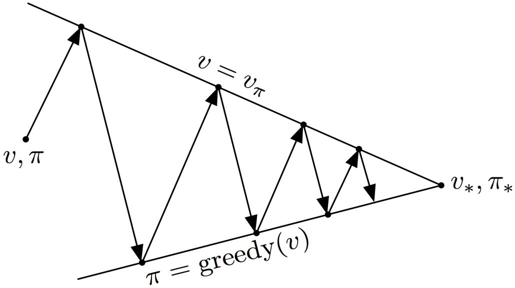
$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha[R_{t+1} + \gamma \hat{v}(S_{t+1}, \boldsymbol{w}) - \hat{v}(S_t, \boldsymbol{w})]\nabla \hat{v}(S_t, \boldsymbol{w})$$

- Linear TD(0):

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha[\underbrace{R_{t+1} + \gamma \boldsymbol{w}^T \boldsymbol{x}(S_{t+1})}_{U_t:\text{TD-Target}} - \underbrace{\boldsymbol{w}^T \boldsymbol{x}(S_t)}_{\hat{v}:\text{value function}}]\boldsymbol{x}(S_t)$$

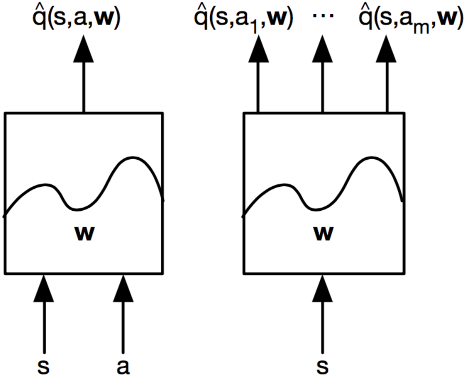
# Control

# Control with function approximation




- Control via generalized policy iteration (GPI):
  - *policy evaluation:* **approximate** policy evaluation: $\hat{q}(\cdot, \cdot, \boldsymbol{w}) \approx q_\pi$
  - *policy improvement:* $\epsilon$-greedy policy improvement

# Types of action–value function approximation




- Action as input: $\hat{q}(s, a, \boldsymbol{w}) \approx q_\pi(s, a)$
- Multiple action–value outputs: $\hat{\boldsymbol{q}}_a(s, \boldsymbol{w}) \approx q_\pi(s, a)$

# Action-value function approximation

- Approximate action-value function $\hat{q}(s, a, \boldsymbol{w}) \approx q_\pi(s, a)$
- Linear case:

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

$$\nabla \hat{q}(s, a, \boldsymbol{w}) = \boldsymbol{x}(s, a)$$

- Minimize squared error on samples $S_t, A_t \mapsto q_\pi$: $\left[q_\pi - \hat{q}(S_t, A_t, \boldsymbol{w})\right]^2$
- Use SGD to find local minimum:

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

# Control with function approximation

- Again, we must substitute target $U_t$ for true action-value $q_\pi(s, a)$:

$$\boldsymbol{w}_{t+1} = \boldsymbol{w}_t + \alpha[U_t - \hat{q}(S_t, A_t, \boldsymbol{w}_t)]\nabla\hat{q}(S_t, A_t, \boldsymbol{w})$$

- **Monte Carlo:** $U_t = G_t$
- **One-step Sarsa:** $U_t = R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, \boldsymbol{w})$

# Policy-based Reinforcement Learning

# Policy–based reinforcement learning

- So far we approximated the action–value function and generated a policy from it
- Approximation of the action–value function

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

- Generation of policy by, e.g., $\epsilon$–greedy

$$\hat{q}(s, a, \boldsymbol{w}) \xrightarrow{\epsilon\text{-greedy}} \pi$$

- Now we directly *parameterize* the policy $\pi$

# Policy optimization

- Policy optimization:

$$\pi_* = \pi(a \mid s, \boldsymbol{\theta}_*)$$

with

$$\boldsymbol{\theta}_* = \arg\max_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

where $J$ is some *performance measure*

- Discounted return: $G_0 = \sum_{k=0}^{T-1} \gamma^k R_{k+1}$

$$\pi_* = \arg\max_{\pi} \mathbb{E}_\pi[G_0] = \arg\max_{\pi} \mathbb{E}_\pi[v_\pi(s_0) \mid S_0 = s_0]$$

- Undiscounted return: $G_0 = \sum_{k=0}^{T-1} R_{k+1}$, i.e., $\gamma = 1$

$$\pi_* = \arg\max_{\pi} \mathbb{E}_\pi[R_0 + R_1 + \ldots + R_{T-1} \mid S_0 = s_0]$$

# Policy optimization

- In continuing environments, we could use the **average** state–value:

$$\sum_{s} \mu_\pi(s) \sum_{a} \pi(a \mid s) \sum_{s'} p(s' \mid s, a) r(s, a, s')$$

where $\mu_\pi$ is the steady–state distribution under $\pi$. This is usuful for the function approximation where states are not well defined (i.e., only seen through their features).

# Parameterized policies

- Policies parameterized by parameter $\boldsymbol{\theta} \in \mathbb{R}^d$:
  - deterministic: $a = \pi(s, \boldsymbol{\theta})$
  - stochastic: $a \sim \pi(a \mid s, \boldsymbol{\theta}) = \Pr\{A_t = a \mid S_t = s, \boldsymbol{\theta}\}$
- Objective becomes $\max_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \max_{\boldsymbol{\theta}} \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[G_0]$
- We can parameterize $\pi$ in any way, as long as it is *differentiable* wrt to $\boldsymbol{\theta}$
- In general we require $\pi(a \mid s, \boldsymbol{\theta}) \in [0, 1]$ for all $s, a$
- If the action space is discrete (and not too large): **softmax policy**

$$\pi(a \mid s, \boldsymbol{\theta}) = \frac{e^{h(s,a,\boldsymbol{\theta})}}{\sum_b e^{h(s,b,\boldsymbol{\theta})}}$$

  where $h(\cdot)$ is the *action preference* function
- Can this approach the deterministic policy?

# Policy gradient methods

- Problem:

$$\pi_* = \pi(a \mid s, \boldsymbol{\theta}_*)$$

with

$$\boldsymbol{\theta}_* = \arg\max_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[G_0]$$

Intuition: collect a bunch of trajectories, and ...
1. Make the good trajectories more probable
2. Make the good actions more probable
3. Push the policy towards generating good actions

- Policy gradient methods:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \widehat{\nabla J(\boldsymbol{\theta})}$$

  - where $\nabla J(\boldsymbol{\theta})$ is the *policy gradient*:

$$\nabla J(\boldsymbol{\theta}) = \left( \frac{\partial J(\boldsymbol{\theta})}{\partial \theta_0}, \ldots, \frac{\partial J(\boldsymbol{\theta})}{\partial \theta_d} \right)^T$$

# Score function

- We now compute the policy gradient analytically
- Assume policy $\pi_{\boldsymbol{\theta}}$ is differentiable whenever it is non-zero
- and we know the gradient $\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(a \mid s)$
- We have the following useful identity

$$\begin{aligned}\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(a \mid s) &= \pi_{\boldsymbol{\theta}}(a \mid s)\frac{\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(a \mid s)}{\pi_{\boldsymbol{\theta}}(a \mid s)} \\ &= \pi_{\boldsymbol{\theta}}(a \mid s)\nabla_{\boldsymbol{\theta}}\log\pi_{\boldsymbol{\theta}}(a \mid s)\end{aligned}$$

- The **score function** is $\nabla_{\boldsymbol{\theta}}\log\pi_{\boldsymbol{\theta}}(a \mid s)$

# Score: Softmax Policy

- Weight actions using linear combination of features $h(s, a) = \phi(s, a)^\top \boldsymbol{\theta}$
- Probability of action is proportional to exponentiated weight

$$\pi_{\boldsymbol{\theta}} \sim \exp\Big(\phi(s, a)^\top \boldsymbol{\theta}\Big)$$

- The score function is

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid s) = \phi(s, a) - \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[\phi(s, \cdot)]$$

# Score: Gaussian Policy

- In continuous action spaces, a Gaussian policy is natural
- Mean is a linear combination of state features $\mu(s) = \phi(s)^\top \boldsymbol{\theta}$
- Variance may be fixed $\sigma^2$, or can also parametrized
- Policy is Gaussian, $a \sim \mathcal{N}(\mu(s), \sigma^2)$
- The score function is

$$\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid s) = \frac{(a - \mu(s))\, \phi(s)}{\sigma^2}$$

# One-Step MDPs

- Consider a simple class of one-step MDPs
  - Starting in state $s \sim \mu(s)$
  - Terminating after one time-step (taking action a) with reward $r = R_{s,a}$
- Use score function to compute the policy gradient

$$\begin{aligned} J(\boldsymbol{\theta}) &= \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[r] \\ &= \sum_{s \in \mathcal{S}} \mu(s) \sum_{a \in \mathcal{A}} \pi_{\boldsymbol{\theta}}(a \mid s) R_{s,a} \\ \boldsymbol{\nabla}_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \sum_{s \in \mathcal{S}} \mu(s) \sum_{a \in \mathcal{A}} \pi_{\boldsymbol{\theta}}(a \mid s) \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid s) R_{s,a} \\ &= \mathbb{E}_{\pi_{\boldsymbol{\theta}}}[r \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a \mid s)] \end{aligned}$$

# Policy Gradient Theorem

- The policy gradient theorem generalises the previous derivation to multi-step MDPs
- Replaces instantaneous reward r with long-term value $q_\pi(s, a)$
- Policy gradient theorem applies to start state objective, average reward and average value objective

**Theorem (Policy-gradient)**

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}\left[q_\pi(s, a) \nabla_{\boldsymbol{\theta}} \log \pi(a \mid s, \boldsymbol{\theta})\right]$$

# Policy Gradient Methods

# Monte–Carlo policy gradient (REINFORCE)

- Update parameters $\boldsymbol{\theta}$ by stochastic gradient ascent
  - using the policy gradient theorem
  - using return $G_t$ as an unbiased sample of $q_\pi(S_t, A_t)$

Initialize a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Initialize $\boldsymbol{\theta} \in \mathbb{R}^{d'}$
**for all** episodes **do**
  Generate an episode $\tau = (S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T, S_T)$ following $\pi(.|., \boldsymbol{\theta})$
  **for** $t = 0, 1, \ldots, T-1$ **do**
    $G_t \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$ // return at time $t$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\gamma^t G_t \nabla_{\boldsymbol{\theta}} \log \pi(A_t \mid S_t, \boldsymbol{\theta})$ // update rule
  **end for**
**end for**

# REINFORCE with baseline

- Monte–Carlo policy gradient still suffers from high variance

**Theorem (Policy-gradient with baseline)**

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\pi_{\boldsymbol{\theta}}}\left[\, (q_\pi(s,a) - b(s))\, \nabla_{\boldsymbol{\theta}} \log \pi(a \mid s, \boldsymbol{\theta})\right]$$

Update rule: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\gamma^t\,(G_t - b(S_t))\,\nabla_{\boldsymbol{\theta}} \log \pi(A_t \mid S_t, \boldsymbol{\theta})$

- Still unbiased with lower variance!

# REINFORCE with baseline

- We use an approximation of the value function $\hat{v}(s, \boldsymbol{w}) \approx v_\pi(s)$ as a *baseline*
- Policy parameters: $\boldsymbol{\theta}$
- Value estimator parameters: $\boldsymbol{w}$
- Update rule:

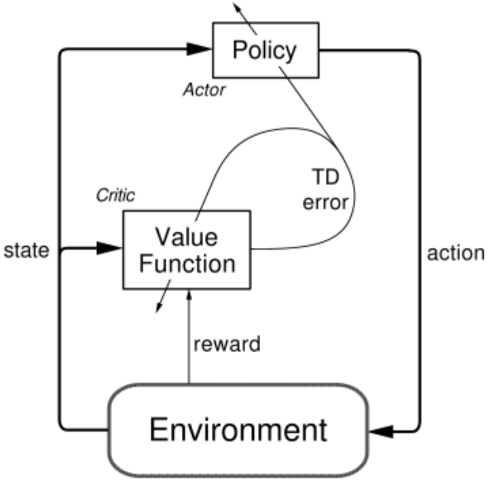
$$\boxed{\begin{aligned} \delta &\leftarrow G - \hat{v}(S_t, \boldsymbol{w}) \\ \boldsymbol{w} &\leftarrow \boldsymbol{w} + \alpha^{\boldsymbol{w}} \gamma^t \delta \nabla_{\boldsymbol{w}} \hat{v}(S_t, \boldsymbol{w}) \\ \boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \gamma^t \delta \nabla_{\boldsymbol{\theta}} \log \pi(A_t \mid S_t, \boldsymbol{\theta}) \end{aligned}}$$

- Interpretation:
  - $\uparrow$ log–prob of action $A_t$ proportionally to how much $G_t$ is better than expected
  - baseline accounts for and removes the effect of past actions

# Estimating the action–value function

- We are solving the prediction problem: policy evaluation
- How good is policy $\pi_{\boldsymbol{\theta}}$ with current parameters $\boldsymbol{\theta}$?
- Familiar toolset for *fitting* the baseline:
  - Monte–Carlo policy evaluation
  - TD-learning
  - TD($\lambda$)
  - LSPI

# Actor-critic concept



# Actor-critic vs. baseline

- Actor-critic methods use the value function as a baseline for policy gradients
- Delivers trade off between *variance reduction* of policy gradients with *bias introduction* from value function methods
- **Critic:** updates value–function parameters $\boldsymbol{w}$
- **Actor:** updates policy parameters $\boldsymbol{\theta}$ using critic
- REINFORCE with baseline uses value–function as *baseline* not as a *critic*
  - not used for *bootstrapping*
- One–step actor-critic update:

$$\delta \leftarrow R_{t+1} + \gamma \hat{v}(S_{t+1}, \boldsymbol{w}) - \hat{v}(S_t, \boldsymbol{w})$$
$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha^{\boldsymbol{w}} \gamma^t \delta \nabla_{\boldsymbol{w}} \hat{v}(S_t, \boldsymbol{w})$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \gamma^t \delta \nabla_{\boldsymbol{\theta}} \log \pi(A_t \mid S_t, \boldsymbol{\theta})$$

# Bias in Actor–Critic algorithms

- Approximating (bootstrapping) the policy gradient introduces bias
- Biased policy gradient might not find the right solution
- But reduces variance and makes learning substantially more efficient
- *Compatible function approximation* avoids this problem: