Machine Learning for Industry

Jose M. Peña
STIMA, IDA, LiU

Lecture Block 8: Reinforcement Learning: REINFORCE Algorithm

AI COMPETENCE
FOR SWEDEN

LINKÖPING UNIVERSITY

Contents

Literature

- Main source
  - Sutton, R. S. and Barto, A. G. *Reinforcement Learning: An Introduction*. The MIT Press, 2018. Chapters 9, 10, 12, 13 and 16. **Available online**.
- Additional source
  - Russel, S. and Norvig, P. *Artifical Intelligence: A Modern Approach*. Pearson, 2010. Chapters 16, 17 and 21.

Function Approximation

- Representing the state and action value functions as look-up tables is not feasible for large state spaces, due to storage space and time to convergence. Instead, we may want to represent them as **parameterized functions**.
- This may bring advantages such as fewer parameters than table entries and thus easier to learn and reach convergence, less storage space, generalization to unvisited states, etc. but also the disadvantage of having to select the right class of functions.
- While this is an instance of supervised learning, it involves some challenges or peculiarities, e.g. bootstrapping, incremental, online, etc.
- Objective for prediction (a.k.a. policy evaluation): Find the function parameters or weights w that minimize the mean squared value error:

  $$\overline{VE}(w) = \sum_s \mu(s) \left[v_\pi(s) - \hat{v}(s, w) \right]^2$$

  where $\mu(s)$ is a probability indicating the relative importance of the state s, e.g. the fraction of time spent in s. Usually, the states in the learning data are distributed according to $\mu(s)$ and, thus, we can drop $\mu(s)$ from the equation above.

Stochastic Gradient Descent

- When a new example or observation $v_\pi(S_t)$ arrives, update the weights as

$$w_{t+1} = w_t - \frac{1}{2}\alpha\nabla\overline{VE}(w) = w_t + \alpha[v_\pi(S_t) - \hat{v}(S_t, w_t)]\nabla\hat{v}(S_t, w_t)$$

  where $\alpha > 0$ is the learning rate or step size. The update converges to a local minimum of $\overline{VE}(w)$ with e.g. $\alpha = 1/t$.
- If we do not observe $v_\pi(S_t)$ but a noisy version U_t of it (e.g., G_t or $R_{t+1} + \gamma\hat{v}(S_{t+1}, w_t)$), then the update above with U_t in the place of $v_\pi(S_t)$ still converges to a local minimum of $\overline{VE}(w)$ as long as U_t is an unbiased estimator of $v_\pi(S_t)$, i.e. $E[U_t|S_t] = v_\pi(S_t)$ for all t. For instance, $U_t = G_t$.

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

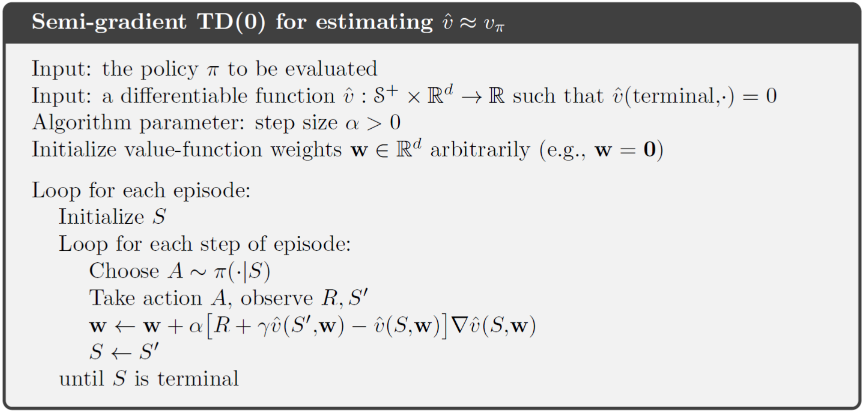
```
Input: the policy $\pi$ to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

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

- Note that updating the parameters may change the value of every state, not only of those visited, i.e. **generalization**.

Stochastic Gradient Descent

- On the other hand, if $U_t = R_{t+1} + \gamma \hat{v}(S_{t+1}, w_t)$ then U_t depends on w_t and, thus, it is not an unbiased estimator of $v_\pi(S_t)$ and, thus, we do not have a true SGD method but a **semi-gradient method**, which **may** still converge but to a point **near** a local optimum of $\overline{VE}(w)$. SSGD is typically faster than SGD, has less variance and allows online learning.
- SSGD update rule:

$$w_{t+1} = w_t + \alpha[R_{t+1} + \gamma \hat{v}(S_{t+1}, w_t) - \hat{v}(S_t, w_t)]\nabla \hat{v}(S_t, w_t).$$



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

```
Input: the policy $\pi$ to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ 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
```

Linear Approximations

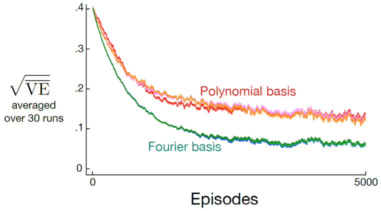
- The approximation $\hat{v}(s, w)$ is a linear function of the weights w, i.e.

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

  where $x_i : states \to \mathbb{R}$, and $x(s)$ is known as the **feature vector** of s.
- Since $\nabla v(s, w) = x(s)$, the updating rule in SGD or SSGD simplifies to

$$w_{t+1} = w_t + \alpha[U_t - \hat{v}(S_t, w_t)]x(S_t).$$

- Note that $\overline{VE}(w)$ is quadratic in w and, thus, it has a single (global) minimum. So, SGD converges to the global minimum, whereas SSGD converges to a point near the global minimum.
- Feature construction to incorporate **feature interactions**:
  - Polynomials: $x_i(s) = \prod_{j=1}^{k} s_j^{c_{i,j}}$ with $s = (s_1, \ldots, s_k)^\top$ and $c_{i,j} \in \{0, \ldots, n\}$.
  - Fourier basis: $x_i(s) = cos(\pi s^\top c^i)$ with $s = (s_1, \ldots, s_k)^\top$, $c^i = (c_1^i, \ldots, c_k^i)$ with $i = 1, \ldots, (n+1)^k$, and $c_j^i \in \{0, \ldots, n\}$.
  - Etc.



Non-Linear Approximations

- Neural networks: New example implies backpropagation to compute the gradient.
- k-Nearest neighbors: When $\hat{v}(s)$ is needed, find the k examples in memory with closest states to s, and return the weighted average of their values. Non-parametric.
- Kernel-based approximations:

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

  where D are the examples in memory, and $k(s,s')$ is a kernel function, e.g. $k(s,s') = exp\left(\frac{-||s-s'||^2}{2\sigma^2}\right)$ is the Gaussian kernel (a.k.a. radial basis function) which is parameterized by the so-called smoothing factor or width σ^2. The kernel function represents the relevance on an example for computing the desired estimate. Moreover, any linear approximation with feature vector $x(s)$ can be recast as a kernel-based approximation with $k(s,s') = x(s)^\top x(s')$. Note that we do not really need to construct the feature vectors explicitly, we just need to compute their inner product. Moreover, under some conditions, a (made up) kernel function can be written as $k(s,s') = x(s)^\top x(s')$ for some feature vector $x(s)$. This allows us to work in the feature space without actually constructing it. This is called the kernel trick.

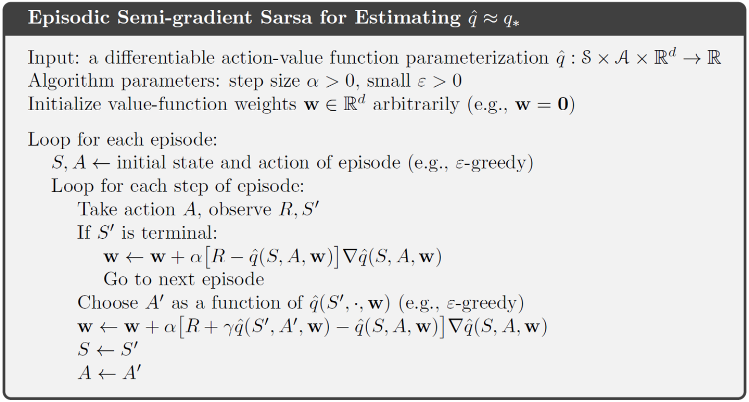
Semi-gradient Sarsa

- To find an approximate solution to a RL problem, we need to consider prediction and control steps (a.k.a. policy evaluation and improvement).
- To this end, let $\hat{q}(s, a, w)$ be an approximation of $q_\pi(s, a)$. Define the objective function

$$\overline{VE}(w) = \sum_s \mu(s)\big[q_\pi(s, a) - \hat{q}(s, a, w)\big]^2$$

and consider the SSGD update rule

$$w_{t+1} = w_t + \alpha\big[R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, w_t) - \hat{q}(S_t, A_t, w_t)\big]\nabla \hat{q}(S_t, A_t, w_t).$$

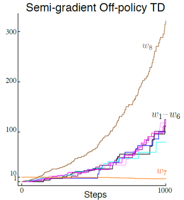


Episodic Semi-gradient Sarsa for Estimating $\hat{q} \approx q_*$

```
Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \rightarrow \mathbb{R}$
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., $\varepsilon$-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., $\varepsilon$-greedy)
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma \hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\big]\nabla \hat{q}(S, A, \mathbf{w})$
        $S \leftarrow S'$
        $A \leftarrow A'$
```

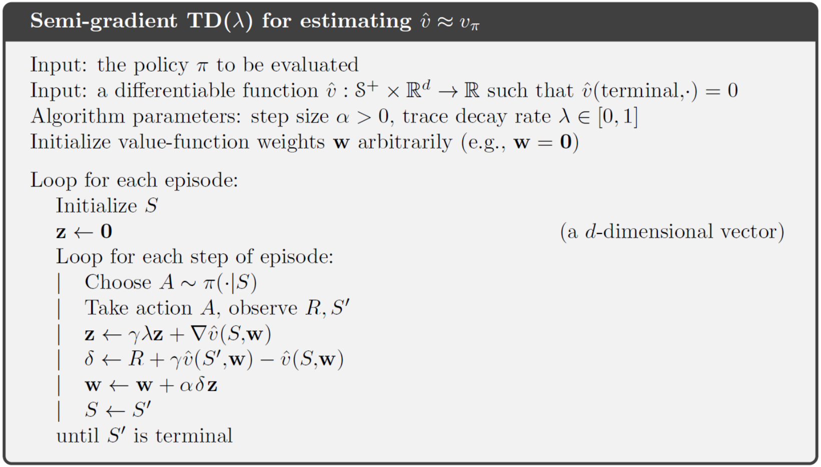
Off-policy Divergence

- Function approximation can also be used with off-policy methods:
$$w_{t+1} = w_t + \alpha\rho_t[R_{t+1} + \gamma\hat{v}(S_{t+1}, w_t) - \hat{v}(S_t, w_t)]\nabla\hat{v}(S_t, w_t)$$
with
$$\rho_t = \pi(A_t|S_t)/b(A_t|S_t).$$
- However, their convergence is **not** guaranteed: It depends on the class of functions and features. More research is needed, as we would not like to abandon these methods since they provide flexibility in the trade-off between exploration and exploitation.
- Example: Consider a MDP with states $s = 1, 2$, no rewards and $\hat{v}(s) = ws$. Assume that the only action in state 1 is to move to state 2, i.e. $\rho = 1$. Assume that $w_t = 10$. Then, $w_{t+1} = (1 + \alpha(2\gamma - 1))w_t > w_t$ if $\gamma > 0.5$. This implies that moving from state 1 to state 2 increases w and, thus, the values of both states. Assume that moving from state 2 to state 1 has probability zero under the target policy, i.e. $\rho = 0$. Then, moving back to state 1 does not change w, moving to 2 again increases w, and so on.



Sarsa(λ)

- The **eligibility trace** z_t indicates the eligibility of a component of w_t for undergoing updating. It does so by keeping track of which components have contributed to recent state valuations, where recent is defined in terms of $\gamma\lambda$, where $\lambda \in [0,1]$ is the trace decay parameter. That is, z_t is a short-term memory, as opposed to the long-term memory w_t.

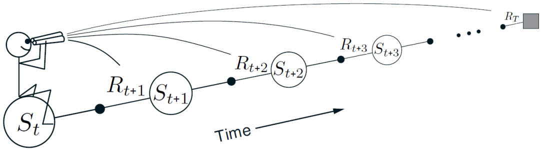


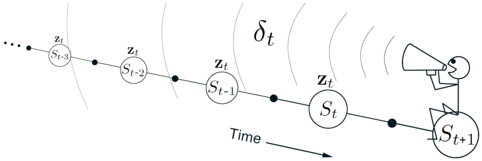
Semi-gradient TD(λ) for estimating $\hat{v} \approx v_\pi$

```
Input: the policy $\pi$ to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Algorithm parameters: step size $\alpha > 0$, trace decay rate $\lambda \in [0,1]$
Initialize value-function weights $\mathbf{w}$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
    Initialize $S$
    $\mathbf{z} \leftarrow \mathbf{0}$                                    (a $d$-dimensional vector)
    Loop for each step of episode:
    |   Choose $A \sim \pi(\cdot|S)$
    |   Take action $A$, observe $R, S'$
    |   $\mathbf{z} \leftarrow \gamma\lambda\mathbf{z} + \nabla\hat{v}(S,\mathbf{w})$
    |   $\delta \leftarrow R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$
    |   $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\mathbf{z}$
    |   $S \leftarrow S'$
    until $S'$ is terminal
```

Sarsa(λ)

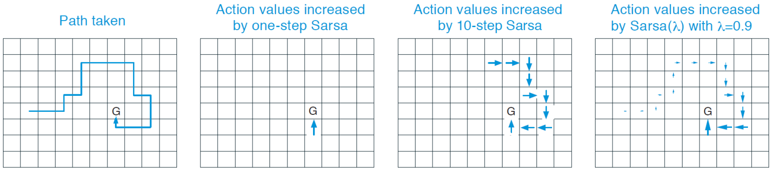
- If $\lambda = 0$, then we obtain the previous TD method, hence the name TD(0).
- If $\lambda = 1$, then we obtain a MC method but incremental and **online** (i.e., not delayed) and, thus, applicable to continuing tasks.
- Then, TD(λ) unifies TD and MC methods. So do *n*-step TD methods. However, TD(λ) has advantages:
  - No need to store the last *n* steps, it suffices storing the eligibility trace.
  - Incremental and online (i.e., not delayed) and, thus, more reactive to bad decisions.





Sarsa(λ)

- Sarsa(λ) is a generalization of Sarsa (a.k.a. Sarsa(0)) with the eligibility traces of TD(λ):
  - Like in TD(λ),
    $$w \leftarrow w + \alpha\delta z.$$
  - Like in Sarsa,
    $$\delta \leftarrow R + \gamma\hat{q}(S', A', w) - \hat{q}(S, A, w).$$
  - Like in TD(λ),
    $$z \leftarrow \gamma\lambda z + \nabla\hat{q}(S, A, w).$$


Path taken
Action values increased by one-step Sarsa
Action values increased by 10-step Sarsa
Action values increased by Sarsa(λ) with λ=0.9
G
G
G
G

The first panel shows the path taken by an agent in a single episode. The initial estimated values were zero, and all rewards were zero except for a positive reward at the goal location marked by G. The arrows in the other panels show, for various algorithms, which action-values would be increased, and by how much, upon reaching the goal. A one-step method would increment only the last action value, whereas an n-step method would equally increment the last n actions' values, and an eligibility trace method would update all the action values up to the beginning of the episode, to different degrees, fading with recency. The fading strategy is often the best. ■

- Q-learning can also be extended to eligibility traces, albeit not as easily.

Policy Gradient Methods

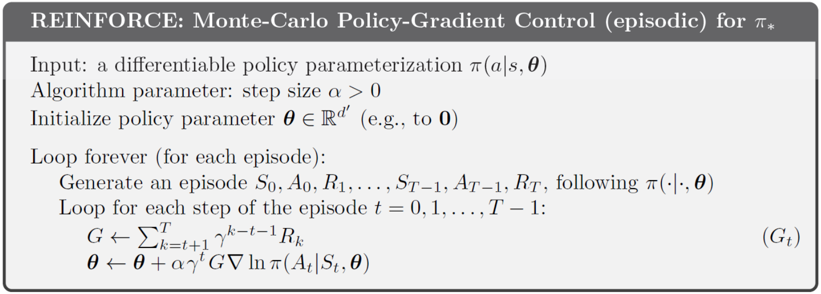
- We now aim to learn a **parameterized policy** to select actions without consulting state or action values.
- To this end, we define the probability of selecting action a in state s as

$$\pi(a|s,\theta) = e^{h(s,a,\theta)} / \sum_b e^{h(s,b,\theta)}$$

  where $h(s,a,\theta)$ represents our (unitless or relative) preference for a in s. **Note** that $h(s,a,\theta) \neq q(s,a,w)$.
- We can use any parameterization for $h(s,a,\theta)$, e.g. a linear approximation $h(s,a,\theta) = \theta^\top x(s,a)$, or a non-linear one such as a NN.
- Advantages of policy gradient methods:
  - Possibility to model deterministic and stochastic policies. The latter may be better in problems with significant function approximation or imperfect knowledge, e.g. bluffing in poker.
  - The policy may be a simpler function to approximate than the state or action value functions.
  - Possibility to inject prior knowledge through the parameterization chosen.
- Performance for **episodic** tasks: $J(\theta) = v_{\pi_\theta}(s_0)$ where s_0 is the initial state.
- Stochastic gradient **ascent**: $\theta_{t+1} \leftarrow \theta_t + \alpha \nabla J(\theta_t)$.
- It can also be adapted to continuing tasks.

Policy Gradient Methods

- **Policy gradient theorem**:
$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\theta)$$
where the transition model is not involved, which is typically unknown.
- If π is followed, then
$$\nabla J(\theta) \propto E[\sum_a q_\pi(S_t,a)\nabla\pi(a|S_t,\theta)] = E[\sum_a \pi(a|S_t,\theta) q_\pi(S_t,a)\frac{\nabla\pi(a|S_t,\theta)}{\pi(a|S_t,\theta)}]$$
$$= E[q_\pi(S_t,A_t)\frac{\nabla\pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)}] = E[G_t\frac{\nabla\pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)}] = E[G_t \nabla \ln \pi(A_t|S_t,\theta)].$$
- All this gives rise to the REINFORCE algorithm, which asymptotically converges to a local optimum.


REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

```
Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):
    Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
    Loop for each step of the episode $t = 0, 1, \ldots, T-1$:
        $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$                    $(G_t)$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\gamma^t G \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta})$
```

Policy Gradient Methods

- REINFORCE is a MC method and, as such, it has some advantages (e.g., convergence to local optimum) and disadvantages (slow convergence, high variance estimates, and not incremental online). As before, we can mitigate these problems via **bootstrapping**. That is, replace the full return G_t in REINFORCE with the one-step return $G_{t:t+1} = R_{t+1} + \gamma \hat{v}(S_{t+1}, w)$, and compare it with the baseline $\hat{v}(S_t, w)$:

$$\theta_{t+1} \leftarrow \theta_t + \alpha[R_{t+1} + \gamma \hat{v}(S_{t+1}, w) - \hat{v}(S_t, w)] \nabla \ln \pi(A_t | S_t, \theta_t)$$

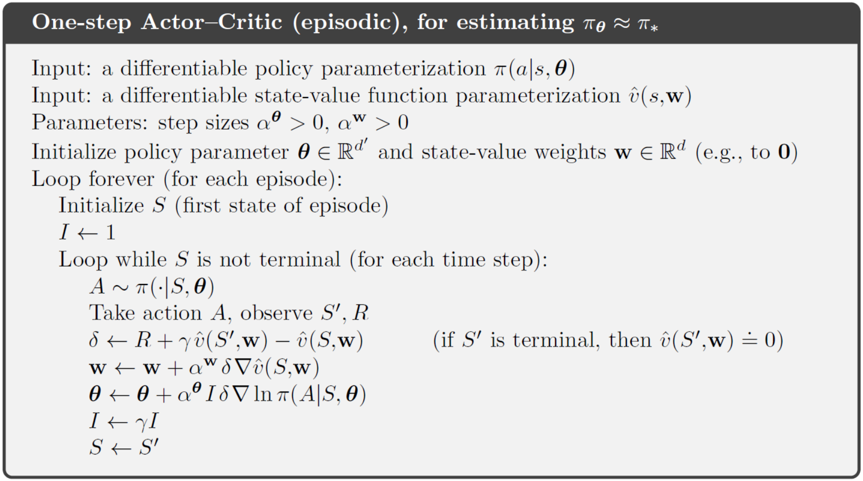
  which results in an **actor-critic** method (actor=policy, critic=baseline), which checks whether the action yields an outcome better than expected, i.e. $R_{t+1} + \gamma \hat{v}(S_{t+1}, w)$ vs $\hat{v}(S_t, w)$.

- This is a generalization of REINFORCE with a baseline, which is justified by the policy gradient theorem with baseline:

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a [q_\pi(s,a) - b(s)] \nabla \pi(a|s,\theta)$$

  which holds because $\sum_a b(s) \nabla \pi(a|s,\theta) = b(s) \nabla 1 = 0$.

Policy Gradient Methods



One-step Actor–Critic (episodic), for estimating $\pi_{\boldsymbol{\theta}} \approx \pi_*$

```
Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d}$ (e.g., to $\mathbf{0}$)
Loop forever (for each episode):
    Initialize $S$ (first state of episode)
    $I \leftarrow 1$
    Loop while $S$ is not terminal (for each time step):
        $A \sim \pi(\cdot|S, \boldsymbol{\theta})$
        Take action $A$, observe $S'$, $R$
        $\delta \leftarrow R + \gamma \hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$        (if $S'$ is terminal, then $\hat{v}(S',\mathbf{w}) \doteq 0$)
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S,\mathbf{w})$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla \ln \pi(A|S, \boldsymbol{\theta})$
        $I \leftarrow \gamma I$
        $S \leftarrow S'$
```

- It can also be adapted to continuing tasks.

Summary

- Function Approximation
- Stochastic Gradient Descent
- Linear Approximations
- Non-Linear Approximations
- Semi-gradient Sarsa
- Off-policy Divergence
- Sarsa(λ)
- Policy Gradient Methods

Thank you

AI COMPETENCE FOR SWEDEN

LINKÖPING UNIVERSITY