Reinforcement Learning

Part 3: Advanced Topics

Machine Learning, Fall 2024

Michael Wand

using some material from Paulo Rauber, Faustino Gomez, Jan Koutnik

TA: Dylan Ashley (dylan.ashley@idsia.ch)

Summary so far

- We have covered
  - (Generalized) Policy Iteration: standard RL training loop
  - Model-based RL: Dynamic Programming algorithms
  - Model-free RL: Monte Carlo methods and Temporal Difference Learning
- For Dynamic programming, we needed access to the model
  - ...then one can algorithmically approximate the optimal state values and the optimal policy.
- If the model is intractable or unknown
  - … need to generate “experience”.
  - Need to balance *Exploration* and *Exploitation.*

Overview for today

So far we have assumed

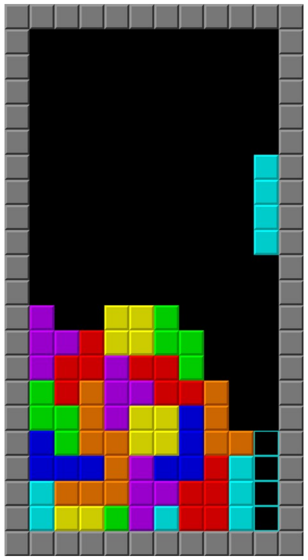
- a finite set of states and actions
- states are fully observable.

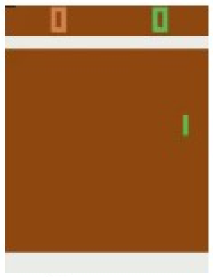
In this lecture we will briefly introduce *approximative* methods which work if the set of states is continuous, not fully known, or if the states are not fully observable.

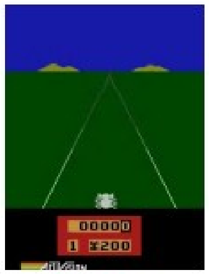
- These methods also allow to deal with *very large* state sets.

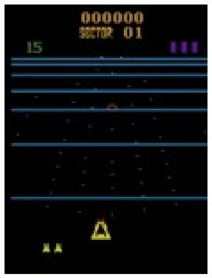
Examples

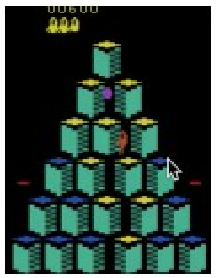
Examples: Tetris, Atari gameplay, Robot exploring the world with its own “eyes”

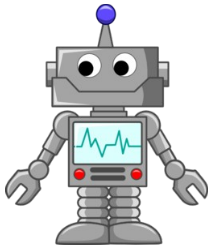



Pong


Enduro


Beamrider


Q*bert



Tetris, ~10^{60} states (Image: Wikipedia)
Atari: ~10^{16992} states (Images: Deep RL bootcamp)
Robot: ??? states?

Overview for today

Fundamental idea:

- *generalize* across states
- requires
  - good state features, “smooth” representation
  - versatile representation of the action-value function, or the policy.

We will cover two major lines of algorithms:

- Deep Q-Learning (approximate action-value function)
- Policy Gradient methods (approximate the policy).

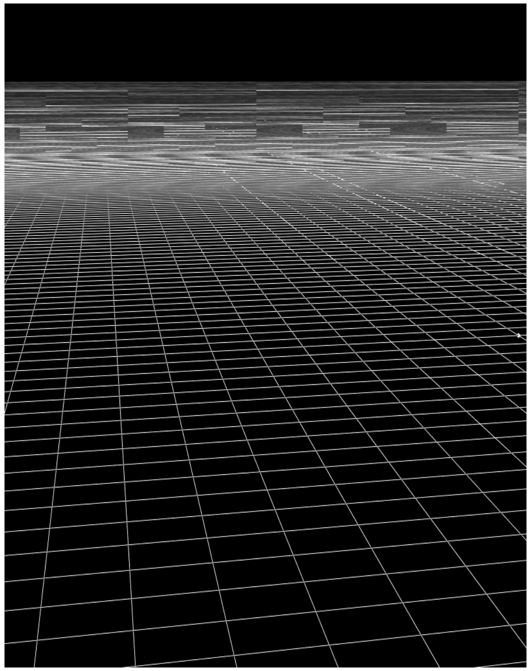
Deep Q-Learning

Deep Q-Learning

- Recap Tabular Q-Learning:
  - Have a table of value estimates for each state-action pair
  - Update rule after each step:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[\underbrace{(R_{t+1} + \gamma \max_a Q(S_{t_1}, a))}_{target} - Q(S_t, A_t) \right]$$

  - If there are "many" states, this table can get very large indeed…



Deep Q-Learning

- Deep Q-Learning (Mnih et al. 2013) is among the most successful neural network based RL approaches.
- Idea: Train a neural network to predict the action-value function *Q*
  - this means that states can take *continuous* values!
  - information is shared between “similar” states!
- Training follows the original Q-Learning algorithm.
- Task: In the original paper, task is to play (classical) computer games (in a simulated environment)
- Input: Computer screen (210 * 160 pixels)
- Discrete actions (move joystick in any of 8 directions, buttons)
- Reward: change in game score

Deep Q-Learning

- Step 1: parametrize *Q* by a neural network:

$$Q = Q(s, a, \theta)$$

- Step 2: squared loss, written as expectation over the *behavioral distribution ρ*

$$L(\theta_i) = E_{s,a\sim\rho}(y_i - Q(s,a,\theta_i))^2$$

$$y_i = E_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q(s', a', \theta_{i-1}) \middle| s, a\right]$$

- where y_i is the target of the optimization, and $\mathcal{E}$ represents the environment.
- The target depends on the weights of the NN! In order to compute target, use weights from previous iteration.

Deep Q-Learning

- Learning: replace expectation by samples, train by stochastic gradient descent

$$\nabla_{\theta_i} L_i\left(\theta_i\right) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\mathcal{E}}\left[\left(r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i)\right)\nabla_{\theta_i} Q(s,a;\theta_i)\right]$$

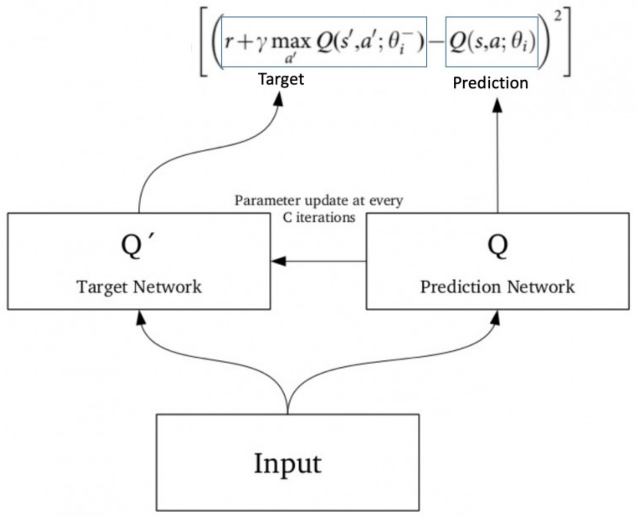
- Common ε-greedy setup to balance exploration and exploitation.
- Greedy target policy improves with each update step.
- Everything ok?

Deep Q-Learning

- There are pitfalls:
  - convergence is not guaranteed even for small learning rates, even if the NN is powerful enough to represent the “true” *Q*
    - this is due to the “moving target” nature of learning
    - note that each weight update changes the action-values of *all* states
  - how to measure training quality?
    - rewards are very noisy, no independent development set
  - updates within a trajectory are highly correlated!
    - violates the fundamental iid assumption of machine learning!

Deep Q-Learning

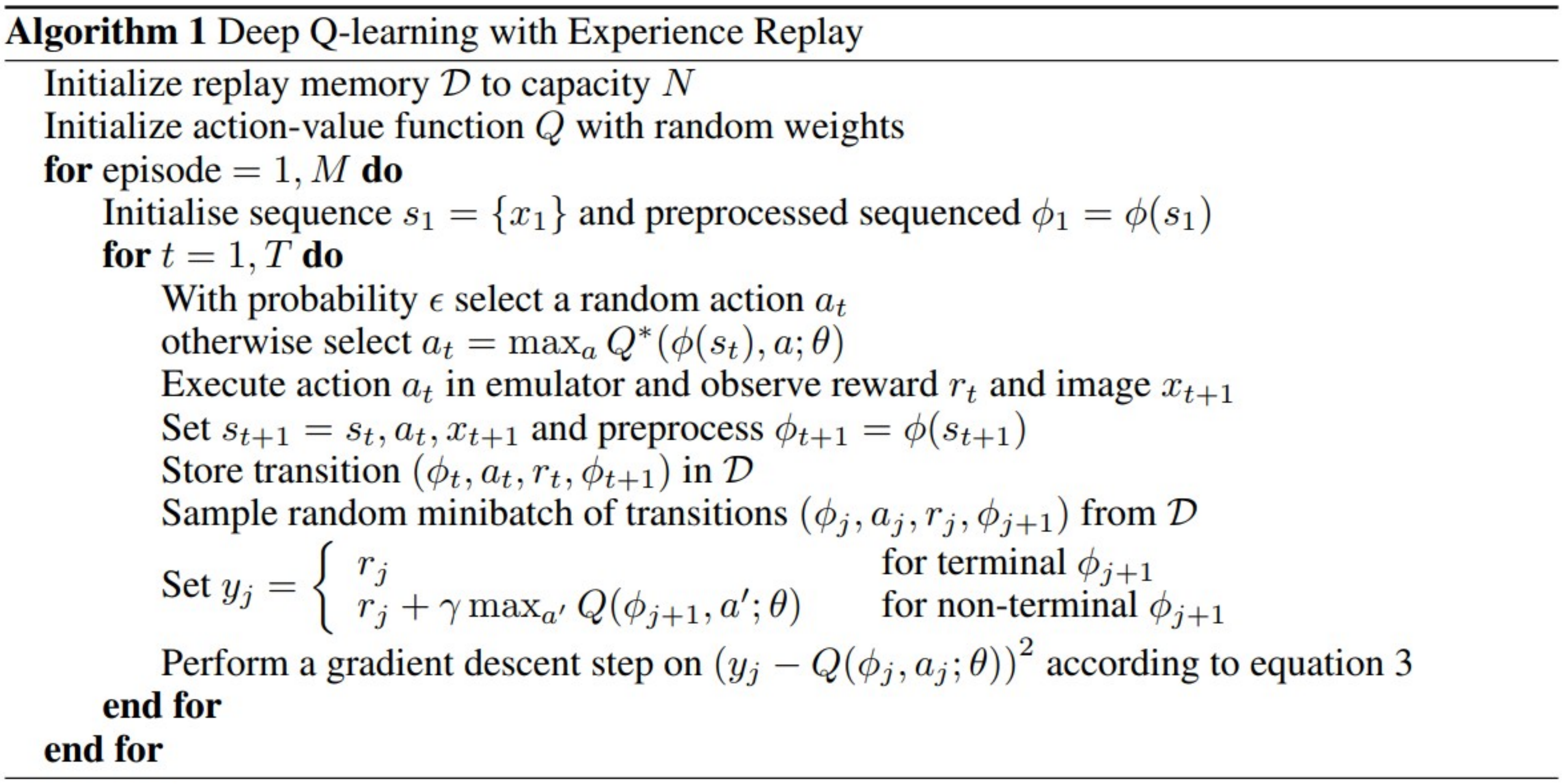
- High-level idea: stabilize training by making it more similar to classical supervised learning.
  - *Experience Replay* (Lin, 1993): Update *Q* on batches of past experience
    - makes the data distribution more stationary
    - increases data efficiency
  - *Target network:* keep target weights fixed for some iterations, update periodically
- Clearly, both methods make the learning strongly off-policy!



$\left[\left(r+\gamma \max_{a'} Q(s',a';\theta_i^-) - Q(s,a;\theta_i)\right)^2\right]$
Target
Prediction
Parameter update at every C iterations
Q′
Target Network
Q
Prediction Network
Input

Deep Q-Learning - Algorithm

(source: Mnih et al. 2013)



Algorithm 1 Deep Q-learning with Experience Replay

```
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
for episode $= 1, M$ do
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
    end for
end for
```

Deep Q-Learning - Results

- (sources: Mnih et al. 2013)

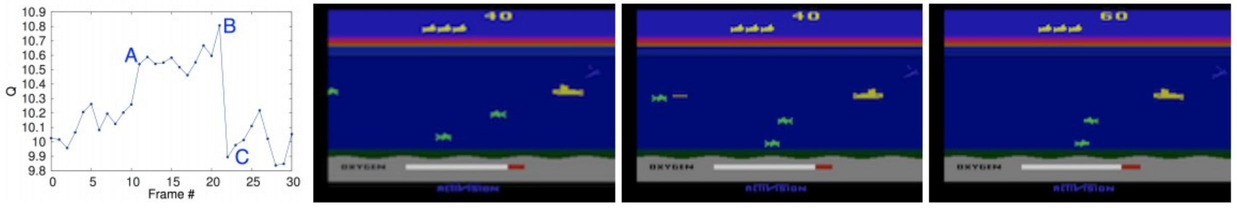
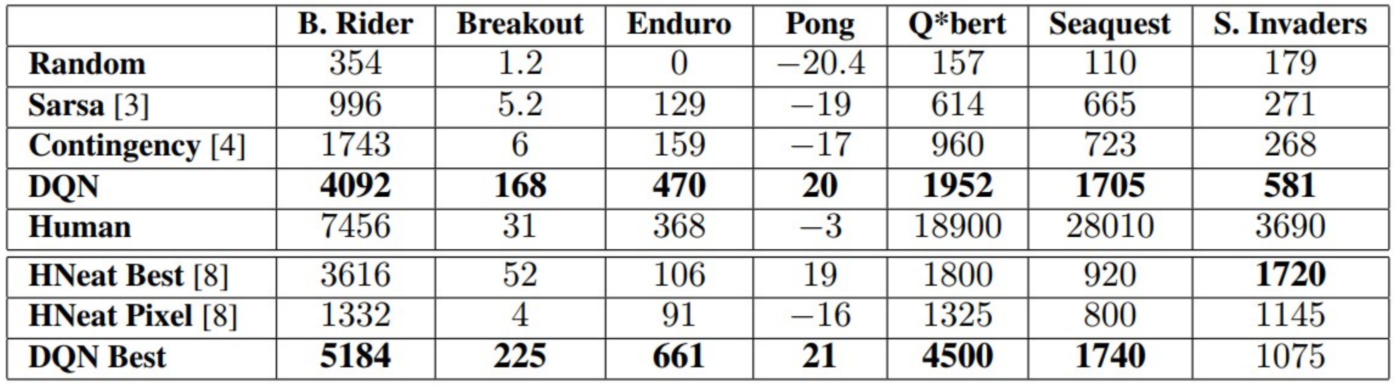


Figure 3: The leftmost plot shows the predicted value function for a 30 frame segment of the game Seaquest. The three screenshots correspond to the frames labeled by A, B, and C respectively.

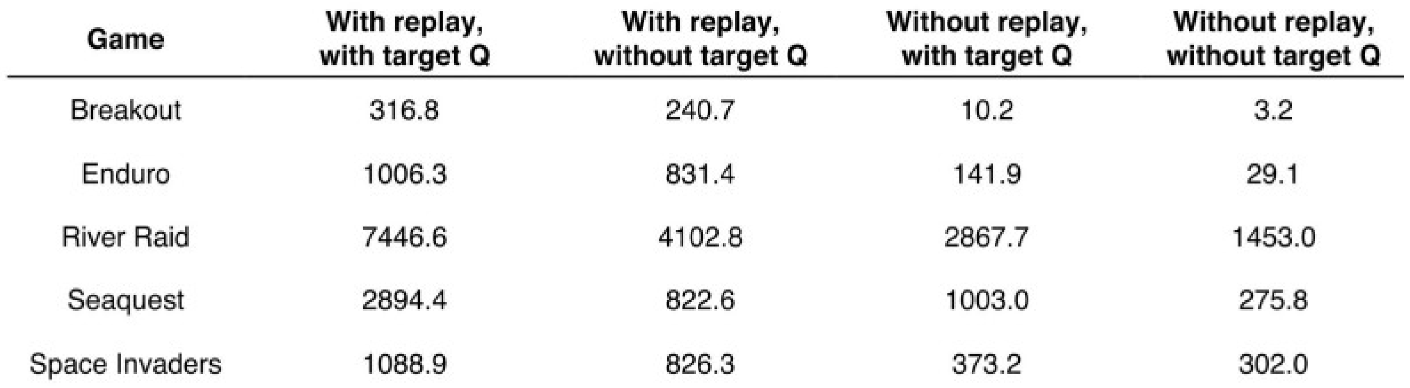


	B. Rider	Breakout	Enduro	Pong	Q*bert	Seaquest	S. Invaders
Random	354	1.2	0	−20.4	157	110	179
Sarsa [3]	996	5.2	129	−19	614	665	271
Contingency [4]	1743	6	159	−17	960	723	268
DQN	**4092**	**168**	**470**	**20**	**1952**	**1705**	**581**
Human	7456	31	368	−3	18900	28010	3690
HNeat Best [8]	3616	52	106	19	1800	920	**1720**
HNeat Pixel [8]	1332	4	91	−16	1325	800	1145
DQN Best	**5184**	**225**	**661**	**21**	**4500**	**1740**	1075

Table 1: The upper table compares average total reward for various learning methods by running an ϵ-greedy policy with $\epsilon = 0.05$ for a fixed number of steps. The lower table reports results of the single best performing episode for HNeat and DQN. HNeat produces deterministic policies that always get the same score while DQN used an ϵ-greedy policy with $\epsilon = 0.05$.

Deep Q-Learning - Results

- Efect of stabilizing techniques (average returns, source: Deep RL Bootcamp, lecture 3)



Game	With replay, with target Q	With replay, without target Q	Without replay, with target Q	Without replay, without target Q
Breakout	316.8	240.7	10.2	3.2
Enduro	1006.3	831.4	141.9	29.1
River Raid	7446.6	4102.8	2867.7	1453.0
Seaquest	2894.4	822.6	1003.0	275.8
Space Invaders	1088.9	826.3	373.2	302.0

Policy Gradients

Policy Gradients

- So far, we have computed action-value functions for model-free reinforcement learning
  - ...both in the tabular case (SARSA, Q-Learning, etc.) and approximatively (deep Q-Learning).
- We now consider optimizing the policy directly via gradient descent:
  - may occasionally be simpler than computing action values
  - easy to inject prior knowledge of the form of the policy
  - actions depend smoothly on parameters, straightforward for continuous/large action spaces (where computing the maximum value over all possible actions is difficult or impossible)
  - naturally find stochastic policies
    - so far, we always had greedy policy improvement!

Policy Gradients

- Asume a policy *$\pi=\pi(a|s)$*, parametrized by *θ.*
- The parametrization can be in any way, as long as it is differentiable (e.g. neural network with softmax output layer).
- The optimization target is defined as

$$J(\theta) = \sum_{s\in\mathcal{S}} d^\pi(s) V^\pi(s) = \sum_{s\in\mathcal{S}} d^\pi(s) \sum_{a\in\mathcal{A}} \pi(a|s) Q^\pi(s,a)$$

where $d^\pi(s)$ is the stationary distribution of the Markov chain for π. (In other words, state values are weighted by the long-term probability of actually being in the corresponding state.)

Policy Gradients

- In order to perform gradient *ascent* (*increase* the average state value), we need to compute the gradient of J w.r.t. θ.
- Problem: This gradient depends on the policy π *and* on the stationary state distribution d^π.
  - The stationary state distribution also depends on the environment, which is generally unknown, may be non-differentiable, and can be very complex.
  - Also in conflict with the goal of doing *model-free* RL.

Policy Gradient Theorem

- The *Policy Gradient Theorem* gives a reformulation of the gradient which does not require to compute the gradient of the state distribution. It states that

$$\begin{aligned}\nabla_\theta J(\theta) &= \nabla_\theta \sum_{s\in\mathcal{S}} d^\pi(s) \sum_{a\in\mathcal{A}} Q^\pi(s,a)\pi(a|s) \\ &\underset{\substack{\nearrow \\ \text{proportional to}}}{\propto} \sum_{s\in\mathcal{S}} d^\pi(s) \sum_{a\in\mathcal{A}} Q^\pi(s,a)\nabla_\theta\pi(a|s)\end{aligned}$$

- Furthermore, one can show that the proportionality factor is the episode length (for episodic tasks) and 1 (for continuous tasks).
- In any case, one can compensate for the proportionality by adapting the learning rate.

REINFORCE

- The REINFORCE algorithm (Williams, 1992) is a straightforward MC-based policy gradient method.
- Idea: sample gradients along complete episodes.
- Requires an episodic task (like all Monte Carlo algorithms).

- In order to derive REINFORCE (or any other practical algorithm), we need to convert the formal formulation in the policy gradient theorem into a sample-based representation.

REINFORCE

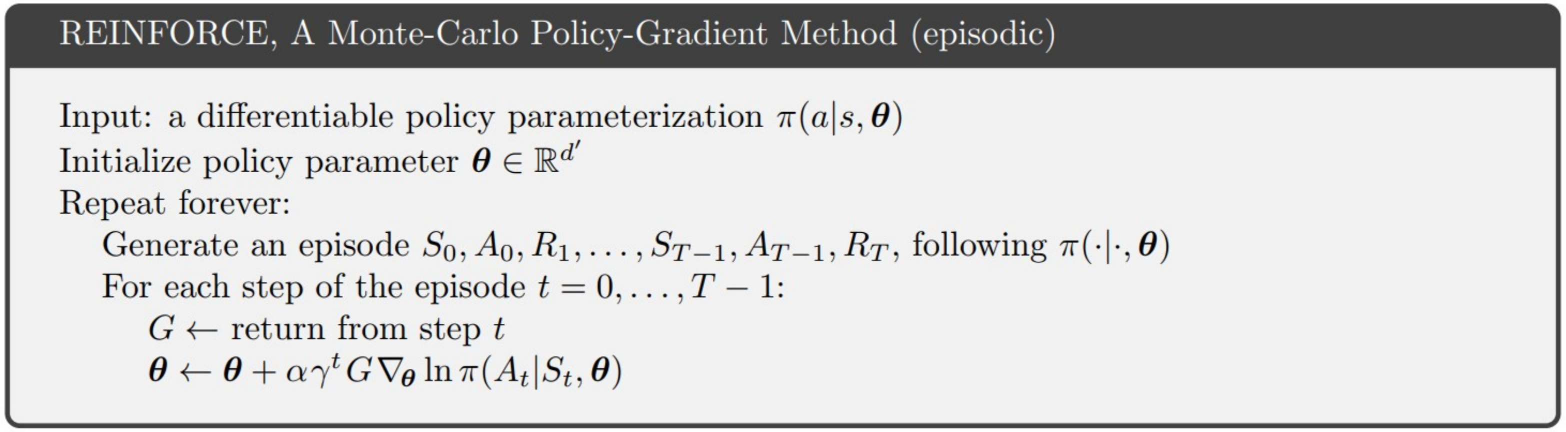
- For REINFORCE, rewrite the gradient like this:

$$
\begin{aligned}
\nabla_\theta J(\theta) &= \sum_{s\in\mathcal{S}} d^\pi(s) \sum_{a\in\mathcal{A}} Q^\pi(s,a)\nabla_\theta \pi(a|s) \\
&= E_\pi\left[\sum_{a\in\mathcal{A}} Q^\pi(S_t,a)\nabla_\theta \pi(a|S_t)\right] && \text{replace weighted sum by expectation} \\
&= E_\pi\left[\sum_{a\in\mathcal{A}} \pi(a|S_t) Q^\pi(S_t,a)\frac{\nabla_\theta \pi(a|S_t)}{\pi(a|S_t)}\right] \\
&= E_\pi\left[Q^\pi(S_t,A_t)\frac{\nabla_\theta \pi(A_t|S_t)}{\pi(A_t|S_t)}\right] && \text{(again) replace weighted sum by expectation} \\
&= E_\pi\left[G_t\frac{\nabla_\theta \pi(A_t|S_t)}{\pi(A_t|S_t)}\right] && \text{by definition of Return} \\
&= E_\pi\left[G_t \nabla_\theta \ln \pi(A_t|S_t)\right] && \text{derivative of the logarithm}
\end{aligned}
$$

- Note that the return of a full episode appears in the equation.

REINFORCE

- This is the full algorithm:



REINFORCE, A Monte-Carlo Policy-Gradient Method (episodic)

```
Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$
Repeat forever:
    Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
    For each step of the episode $t = 0, \ldots, T-1$:
        $G \leftarrow$ return from step $t$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla_{\boldsymbol{\theta}} \ln \pi(A_t|S_t, \boldsymbol{\theta})$
```

- Note that the policy is typically parametrized by a neural network, making the gradient easy to compute.

REINFORCE

- The parameter update

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha G_t \frac{\nabla_{\boldsymbol{\theta}} \pi(A_t|S_t, \boldsymbol{\theta}_t)}{\pi(A_t|S_t, \boldsymbol{\theta}_t)}$$

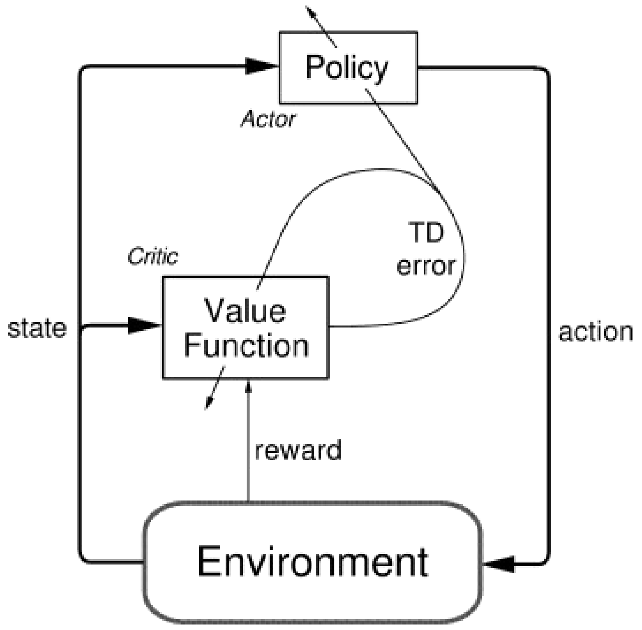
  can be interpreted as the product of the return G and the gradient of the probability of taking the action, divided by a “normalization” factor.
- The gradient is a vector which points in the direction in which the probability increases most quickly. Thus, the probability of taking action A_t in state S_t
  - increases if G is positive
  - decreases if G is negative.
- In general, probability of state-action trajectories with high return will be increased.

REINFORCE

- It can be shown that the REINFORCE policy converges to a local optimum, provided the step size is small enough.
- However, the estimate has a high variance, thus learning can be slow.
- One source of this variance is the high variation of possible returns.

Actor-Critic Methods

- Idea: use ideas of Temporal Difference (TD) learning in the context of policy gradients.
- High-level concept: *Actor-Critic* methods
  - TD methods where the the policy is represented independently of the value function
  - Policy: *actor,* estimated value function: *critic*
  - Estimator updates occur in form of TD error, for both actor and critic



Actor-Critic Policy Gradients

- One can replace the REINFORCE MC return G with a TD-style estimate, taking the value function (critic) into account:

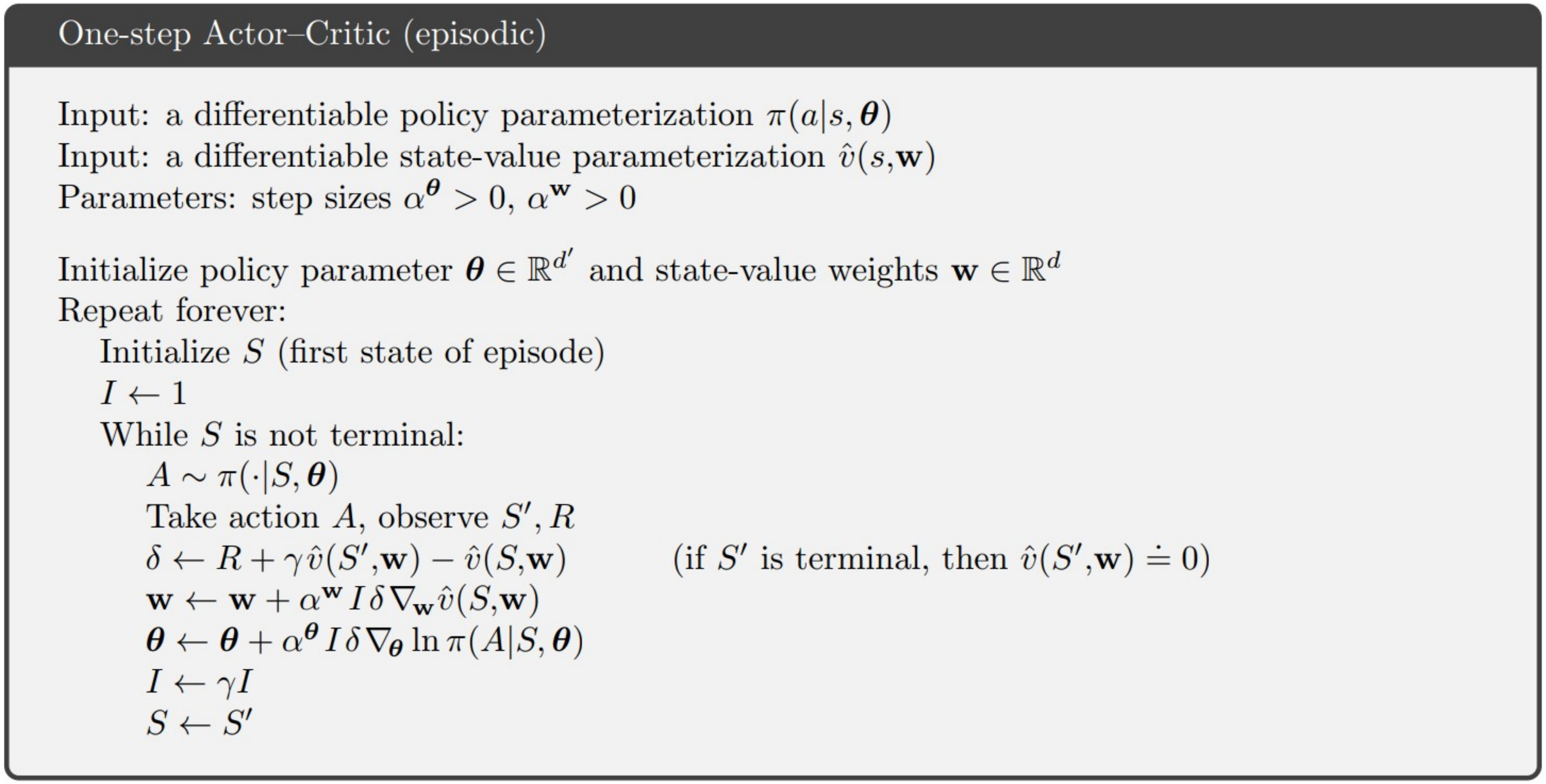
$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha G_t \frac{\nabla_{\boldsymbol{\theta}} \pi(A_t|S_t, \boldsymbol{\theta}_t)}{\pi(A_t|S_t, \boldsymbol{\theta}_t)}$$

$$\Downarrow$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \Big(R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w}) \Big) \frac{\nabla_{\boldsymbol{\theta}} \pi(A_t|S_t, \boldsymbol{\theta}_t)}{\pi(A_t|S_t, \boldsymbol{\theta}_t)}$$

- $\hat{v}$ is the estimate of the value function, parametrized by a vector w.
- The update value $R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w})$ is also used to update the parametrized value function.

Actor-Critic Policy Gradients



One-step Actor–Critic (episodic)

```
Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Input: a differentiable state-value 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 $\mathbf{w} \in \mathbb{R}^{d}$
Repeat forever:
    Initialize $S$ (first state of episode)
    $I \leftarrow 1$
    While $S$ is not terminal:
        $A \sim \pi(\cdot|S, \boldsymbol{\theta})$
        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 $\hat{v}(S',\mathbf{w}) \doteq 0$)
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}} I \delta \nabla_{\mathbf{w}} \hat{v}(S,\mathbf{w})$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I \delta \nabla_{\boldsymbol{\theta}} \ln \pi(A|S, \boldsymbol{\theta})$
        $I \leftarrow \gamma I$
        $S \leftarrow S'$
```

Summary

- Today we have covered two major approaches for approximative reinforcement learning: Deep Q-Learning and Policy Gradient methods.
- This is a very hot topic of research, and many architectural and algorithmic variations have been presented, or are currently under investigation.
  - Besides improving the algorithm itself, several variants deal with improved parallelization (e.g. A3C, *Asynchronous Advantage Actor Critic* algorithm).