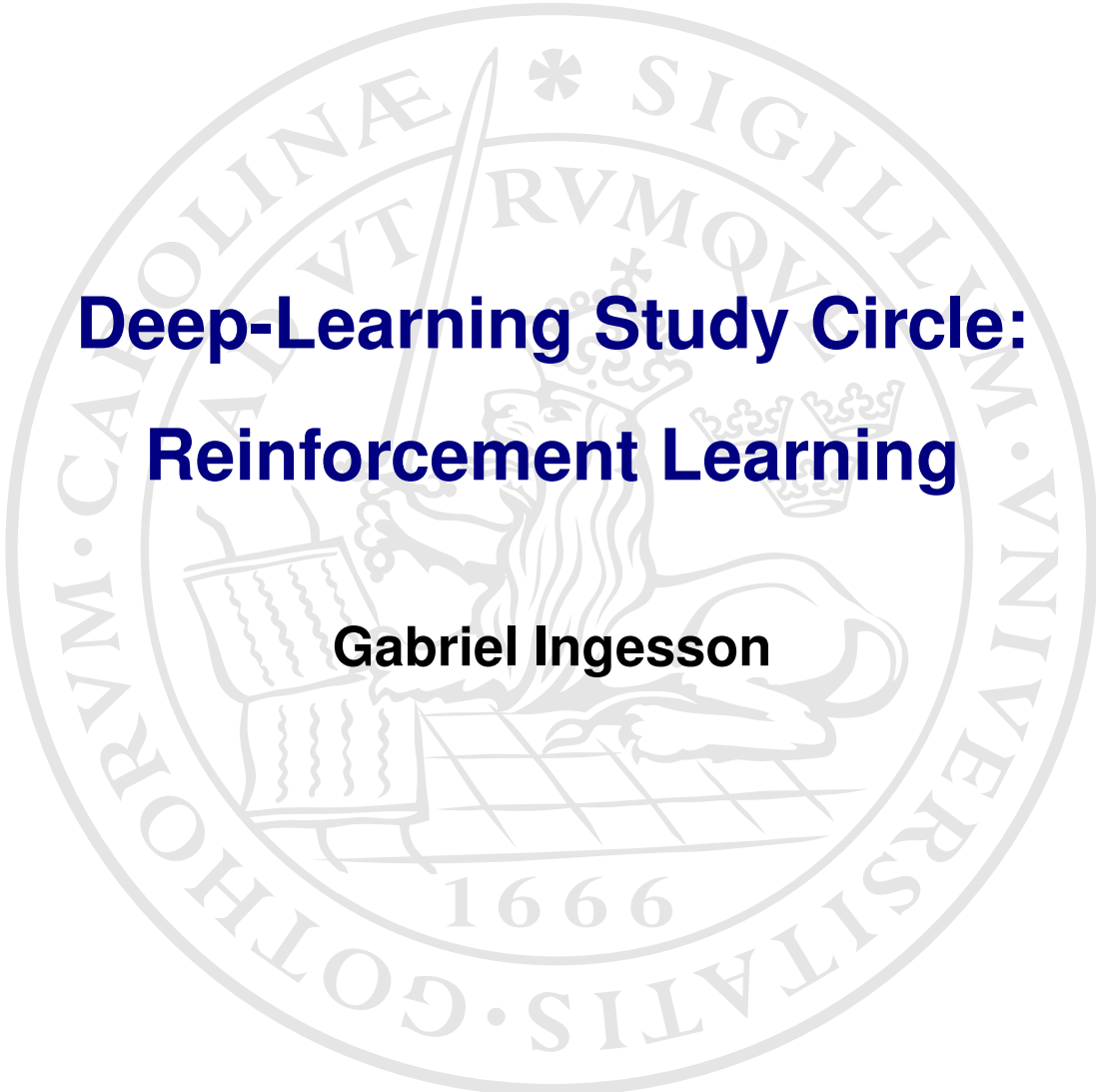


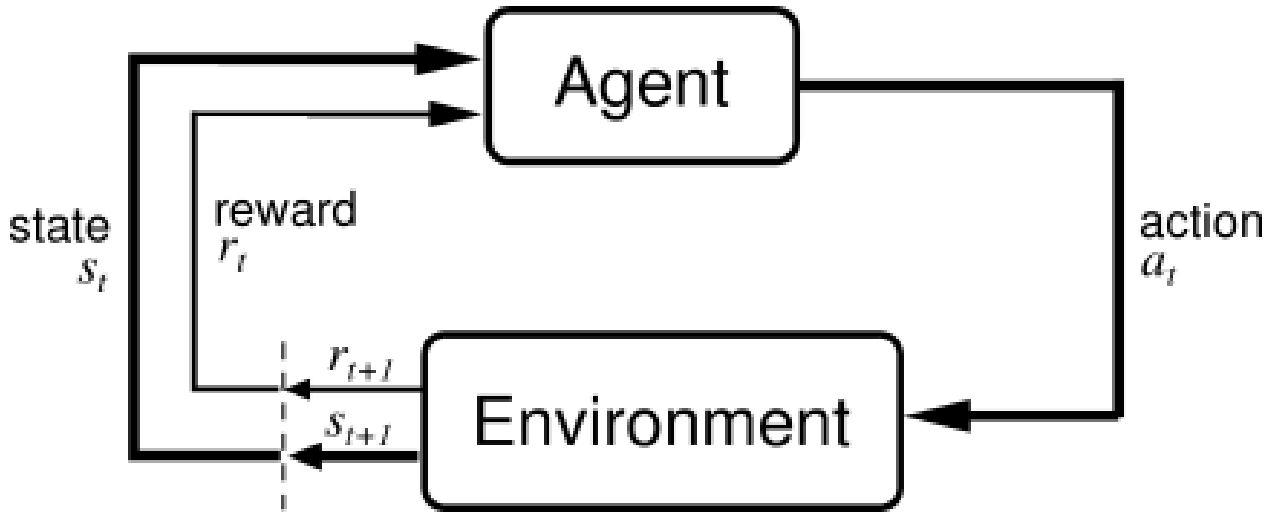
Deep-Learning Study Circle: Reinforcement Learning

Gabriel Ingesson

Reinforcement Learning

- The problem where an agent has to learn a policy (behavior) by taking actions in an environment, with the goal that the policy should maximize a cumulative reward.
- Different from supervised and unsupervised learning:
  - No labeled training data.
  - Reward signal instead of trying to find hidden structure.
- Reinforcement learning can be used in combination with deep neural networks that can be used to approximate policy and cumulative reward functions.

Reinforcement Learning



- Initially, the agent does not have to know anything about the environment.
- The agent recieves a reward signal and the environment state.
- Adjusts its actions in order to maximize the cumulative reward.

Examples

- Pancake Robot
- Atari Game

Overview

- Based on:
  - Sutton, Richard S., and Andrew G. Barto. Reinforcement learning: An introduction. Vol. 1. No. 1. Cambridge: MIT press, 1998.
  - David Silver's course
- Mathematical foundation:
  - Markov decision processes
  - Dynamic programming and the Bellman equation
- Different learning algorithms:
  - Monte Carlo learning
  - Temporal difference learning
    - SARSA
    - Q-Learning
- Relation to deep learning:
  - Deep ANN's for function approximation, Deep Q Network (DQN)
  - Policy gradients
- Homework - OpenAI Gym

Definitions

At each step t the agent:

- Receives observation of the environment state, S_t.
- Recieves scalar reward, R_t.
- Executes action, A_t.

and the environment:

- Recieves action, A_t.
- Updates state, S_{t+1}.
- Emits scalar reward, R_{t+1}.

Reward

- A reward R_t is a scalar feedback signal.
- Indicates how well agent is doing at step t.
- All goals can be described by the maximization of expected cumulative reward.

The return G_t is the total discounted reward from time-step t:

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1},$$

where $\gamma \in [0, 1)$ is a discount factor, favors immediate rewards.

Policy

A policy π is the agent's behaviour, a map from state to action.

- Deterministic: $a = \pi(s)$.
- Stochastic: $\pi(a|s) = \mathbb{P}[A_t = a|S_t = s]$.

The policy should be adjusted in order to maximize the return, G_t.

State-Value Function, $v_\pi(s)$

Evaluates a state, given a policy π.

The state-value function $v_\pi(s)$, is a prediction of the return G_t given a policy and the current state S_t:

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

is used to evaluate a state and helps to select actions.

Action-Value Function, $q_\pi(s, a)$

Evaluate an alternative action, given a policy π.

The action-value function $q_\pi(s, a)$ is the expected return starting from state s, taking a, and then following policy π

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a].$$

is used to evaluate actions and helps to update the policy.

The Markov Property

We assume that the environment state S_t fulfills the Markov property:

$$\mathbb{P}[S_{t+1}, R_{t+1} | A_t, S_t, A_{t-1}, S_{t-1}, \cdots, A_0, S_0] = \mathbb{P}[S_{t+1}, R_{t+1} | A_t, S_t]$$

The current state contains information of all past states and actions.

The Markov property is important in reinforcement learning because decisions and value functions are assumed to be a function only of the current state.

Markov Decision Process

A Markov decision process (MDP) is a Markov process with decisions and rewards, a framework for modeling decision making:

A finite Markov Decision Process is a tuple $< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma >$ where

- $\mathcal{S}$ is a finite set of states.
- $\mathcal{A}$ is a finite set of actions.
- $\mathcal{P}$ is a transition probability matrix: $\mathcal{P}^{a}_{s,s'} = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a]$
- $\mathcal{R}$ is a reward function: $\mathcal{R}^a_s = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$
- γ is a discount factor $\gamma \in [0, 1)$.

The core problem of MDPs is to find a policy for the agent, that maximizes return given the MDP. ”MDPs are 90% of modern reinforcement learning.”

Dynamic Programming

- Is an optimization method for solving a problem by breaking it down into simpler subproblems, solving each of those subproblems just once, and storing their solutions. Can be applied to Markov decision processes.

- Principle of Optimality: An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision. (See Bellman, 1957, Chap. III.3.)

The Bellman Equation

The value functions can be decomposed into immediate reward plus discounted value of successor state

A recursive relationship ((′) denotes subsequent state/action):

$$
\begin{aligned}
v_\pi(s) &= \mathbb{E}[R_{t+1} + \gamma v_\pi(s') | S_t = s] \\
&= \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \right) \\
q_\pi(s) &= \mathbb{E}[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a] \\
&= \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a' \in \mathcal{A}} \pi(a'|s') q_\pi(s', a')
\end{aligned}
$$

v_π and q_π are unique solutions to these equations, they can be used to compute, approximate and learn v_π and q_π.

Optimal state-value function

The optimal value functions is the maximum value function over all policies:

- $v_*(s) = \max_\pi v_\pi(s)$
- $q_*(s,a) = \max_\pi q_\pi(s,a)$

Theorem

For any Markov Decision Process

- *There exists an optimal policy π_* that is better than or equal to all other policies $\pi_* \geq \pi,\ \forall \pi$.*
- *All optimal policies achieve the optimal value function, $v_{\pi_*} = v_*(s)$.*
- *All optimal policies achieve the optimal action-value function, $q_{\pi_*}(s,a) = q_*(s,a)$.*

Bellman Optimality Equation

The optimal value functions are recursively related by the Bellman optimality equations:

$$v_*(s) = \max_a q_*(s,a) = \max_a \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s')$$

$$q_*(s) = \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a [r + \gamma \max_a q_*(s')]$$

- The Bellman optimality equations is non-linear system of equations.
- Extreme computational cost.
- In reinforcement learning one typically has to settle for approximate solutions.
  - Value Iteration
  - Policy Iteration
  - Q-learning
  - Sarsa

Iterative Policy Evaluation & Improvement

Policy evaluation, iterative application of the Bellman equation:

$$v_1 \to v_2 \to v_3 \cdots \to v_\pi$$

$$v_{k+1} = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s')$$

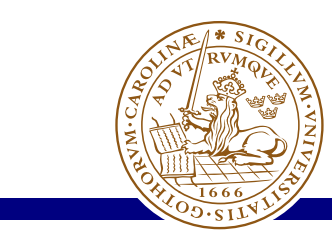
Policy improvement:

$$\pi' = \text{greedy}(v') = \text{argmax}_{a \in \mathcal{A}} q_\pi(s, a).$$

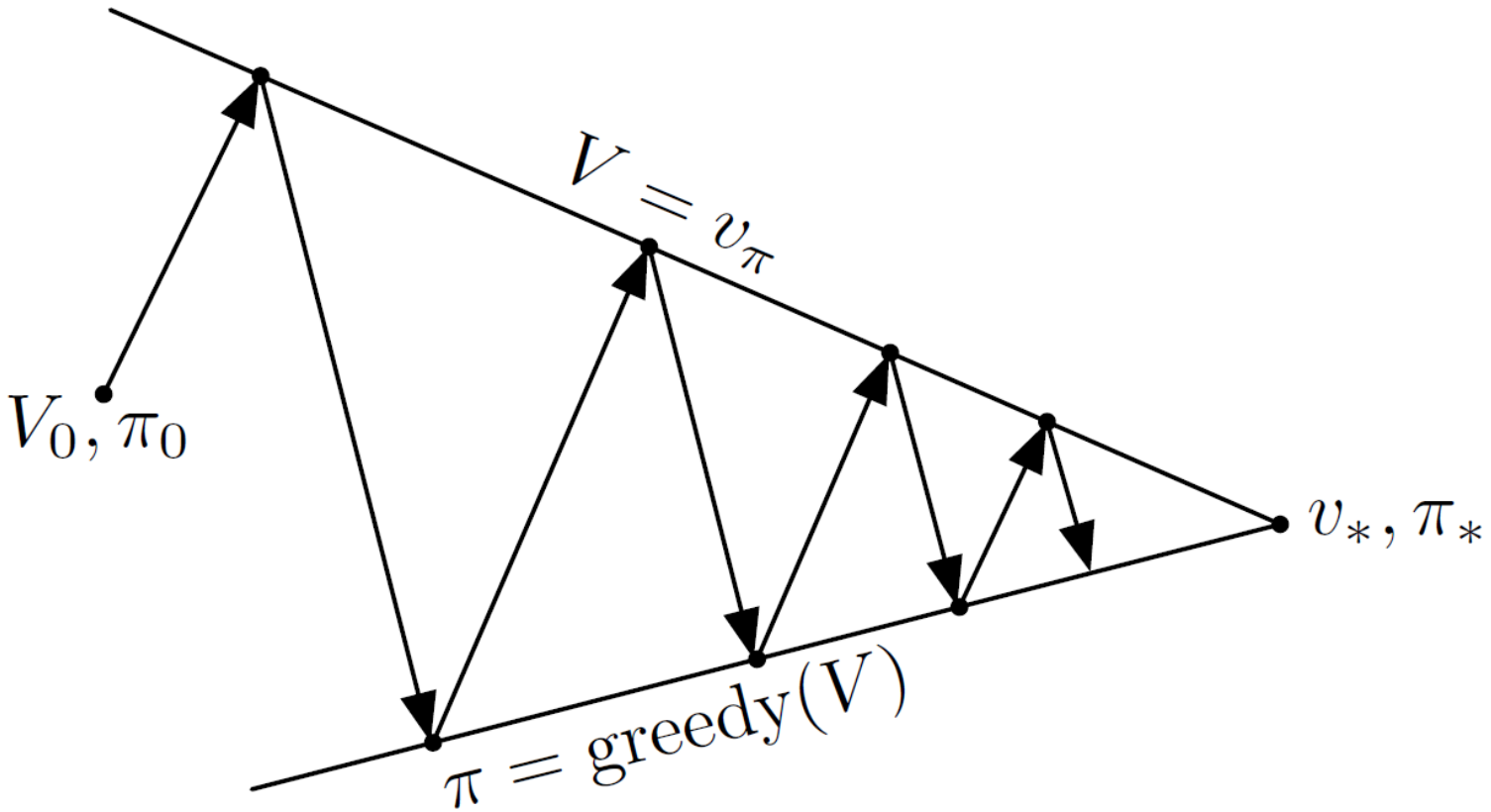
Policy iteration:

$$\pi_0 \to v_{\pi_0} \to \pi_1 \to v_{\pi_1} \to \cdots \to \pi_* \to v_{\pi_*}$$

If improvements stop, then the Bellman optimality equation has been satisfied.



Iterative Policy Evaluation & Improvement



Source: Sutton, Richard S., and Andrew G. Barto. Reinforcement learning: An introduction.

Iterative Policy Evaluation & Improvement

1. Initialize $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$.

2. Policy Evaluation:

Repeat

$\Delta \leftarrow 0$

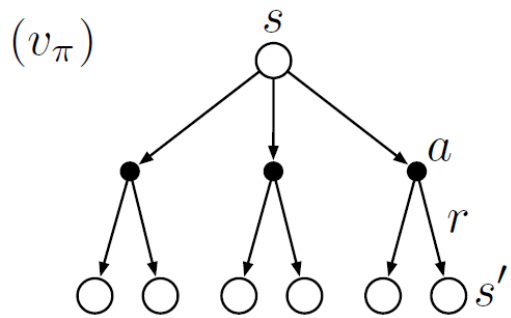
For each $s \in \mathcal{S}$

$v \leftarrow V(s)$

$V(s) \leftarrow \sum_{s',r} p(s',r|s,\pi(s))[r+\gamma V(s')]$

$\Delta \leftarrow \max(\Delta, |v - V(s)|)$

until $\Delta \leq \epsilon$



3. Policy Improvement:

policy-stable $\leftarrow$ true

For each $s \in \mathcal{S}$:

old action $\leftarrow$ argmax$_a \sum_{s',r} p(s',r|s,a)[r+\gamma V(s')]$

If old action $\neq \pi(s)$, then policy-stable $\leftarrow$ false

If policy-stable, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$ else go to 2

General Policy Iteration

- We need complete knowledge of the environment.
- Policy iteration also suffers from Bellman's curse of dimensionality for large problems.
- However, the general idea of letting policy evaluation and policy improvement processes interact, is used in almost all reinforcement-learning methods.

Monte-Carlo Reinforcement Learning

- Model free, no prior knowledge about the environment.
- Monte Carlo methods require only experience, i.e. sample sequences of states, actions, and rewards from interaction with the environment.
- Learns from complete episodes, updates policy from computed return, G_t.

Monte-Carlo Policy Evaluation

Initialize:

$\pi \leftarrow$ policy to be evaluted
$V \leftarrow$ an arbitrary state-value function
Returns$(s) \leftarrow$ and empty list, $\forall s \in \mathcal{S}$

Repeat:

Generate an episode using π
For each state s appearing in the episode:
$G \leftarrow$ following the first occurance of s
Append G to Returns(s)
$V(s) \leftarrow$ average(Returns(s))

By the law of large numbers:

$V(s) \rightarrow v_\pi(s)$ as the numbers of visits at $s \rightarrow \infty$
error σ falls as $1/\sqrt{n}$

Monte-Carlo Reinforcement Learning

Initialize, for all $s \in \mathcal{S}, a \in \mathcal{A}$:

$Q(s,a) \leftarrow$ arbitrarily
$\pi(s) \leftarrow$ arbitrarily
Returns$(s,a) \leftarrow$ an empty list

Repeat:

Choose $S_0 \in \mathcal{S}$ and $A_0 \in \mathcal{A}(S_0)$ randomly
For each state s, a appearing in the episode:
$G \leftarrow$ return following the first occurance of s, a
Append G to Returns(s,a)
$Q(s,a) \leftarrow$ average(Returns(s,a))
for each s in the episode:

$$\pi(a|s) = \begin{cases} 1 - \epsilon + \epsilon/|\mathcal{A}| & \text{if } a = \text{argmax}_a Q(s,a) \\ \epsilon/|\mathcal{A}| & \text{otherwise} \end{cases}$$

Exploitation vs. Exploration

ϵ - Greedy Exploration

$$\pi(a|s) = \begin{cases} 1 - \epsilon + \epsilon/|\mathcal{A}| & \text{if } a = \text{argmax}_{a \in A} Q(s,a) \\ \epsilon/|\mathcal{A}| & \text{otherwise} \end{cases}$$

Monte-Carlo Reinforcement Learning

- MC methods do not use any local information, i.e. bootstrapping like in policy iteration.
- They do not update their value estimates on the basis of other value estimates.
- Only updates $v(s)$ and $q(s, a)$ after completed episodes.

Temporal-Difference (TD) Learning

- TD learning is a combination of Monte Carlo ideas and dynamic programming (DP) ideas.
- Like Monte Carlo methods, TD methods can learn directly from raw experience without a model.
- Like DP, TD methods update estimates based in part on other learned estimates, i.e. use bootstrapping.

Bellman equation:

$$v_\pi(s) = \mathbb{E}[R_{t+1} + \gamma v_\pi(s') | S_t = s, A_t = a]$$

Estimate update, $\alpha \in [0, 1]$:

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)))$$

TD Policy Evaluation

Initialize:

$\pi \leftarrow$ policy to be evaluted
$V \leftarrow$ an arbitrary state-value function

Repeat (for each episode):

Initialize S
Repeat (for each step of episode) S
$A \leftarrow$ action given by π for S
Take action A, observe R', S'
$V(S) \leftarrow V(S) + \alpha(R' + \gamma V(S') - V(S)))$
$S \leftarrow S'$
until S is terminal

n-Step TD

Feedback from the n-step return:

$$G_t^n = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n})$$

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t^n - V(S_t))$$

Intermediate algorithm w.r.t TD(0) and MC.

TD(λ)

The λ return G_t^λ combines all n-step returns G_t^n

$$G_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} G_t^n$$

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t^\lambda - V(S_t))$$

Sarsa: TD control algorithm

State-Action-Reward-State-Action (SARSA)

$Q(S, A)$ depends on (S, A, R', S', A')

Policy evaluation, $Q \approx q_\pi$:

$$Q(S, A) \leftarrow Q(S, A) + \alpha(R' + \gamma Q(S', A') - Q(S, A))$$

Policy improvement is then chosen ϵ- greedy w.r.t. $Q(S, A)$.

Sarsa: TD control algorithm

- Initialize $Q(s,a)$, $\forall s \in \mathcal{S}$, $a \in \mathcal{A}$, and $Q(\text{terminal-state}, \cdot) = 0$
- Repeat (for each episode)
  - Initialize S
  - Choose A from S using policy derived from Q (e.g. ϵ - greedy)
  - Repeat (for each step of episode):
    - Take action A, observe R', S'.
    - Choose A' from S' using policy derived from Q (e.g. ϵ - greedy)
    - $Q(S,A) \leftarrow Q(S,A) + \alpha[R' + \gamma Q(S',A') - Q(S,A)]$
    - $S \leftarrow S',\ A \leftarrow A'$
  - until S is terminal

Q-Learning

- Initialize $Q(s, a)$, $\forall s \in \mathcal{S}$, $a \in \mathcal{A}$, and $Q(\text{terminal-state}, \cdot) = 0$
- Repeat (for each episode)
  - Initialize S
  - Repeat (for each step of episode):
    - Choose A from S using policy derived from Q (e.g. ϵ - greedy)
    - Take action A, observe R', S'.
    - $Q(S, A) \leftarrow Q(S, A) + \alpha[R' + \gamma \max_a Q(S', a) - Q(S, A)]$
    - $S \leftarrow S'$.
  - until S is terminal

Large-Scale Reinforcement Learning

Reinforcement learning can be used to solve large problems, e.g.

- Backgammon 10^{20} states
- Computer Go: 10^{170} states
- Helicopter: continuous state space

So far we have represented the value functions as lookup table, this becomes slow and memory expensive for large problems.

A solution is to estimate the value function with function approximation:

$$\hat{v}(s, \theta) \approx v_\pi(s)$$
$$\hat{q}(s, a, \theta) \approx q_\pi(s, a)$$

And update the parameter θ using MC or TD learning.

Function Approximators

Examples of approximators:

- Linear, $\hat{v}(s, \theta) = \sum_i \theta_i \phi(s)$
- Nonlinear, neural networks

Value-function approximation

Goal: find a parameter θ that minimizes the mean-squared error between approximate value function $\hat{v}(S,\theta)$ and the true value function $v_\pi(s)$.

$$J(\theta) = \frac{1}{2}\mathbb{E}_\pi[(v_\pi(S) - \hat{v}(S,\theta))^2]$$

Gradient descent:

$$\Delta\theta = -\alpha\nabla_\theta J(\theta) = \alpha\mathbb{E}[(v_\pi(S) - \hat{v}(S,\theta))\nabla_\theta\hat{v}(S,\theta)]$$

Stochastic gradient descent samples:

$$\Delta\theta = \alpha\left(v_\pi(S) - \hat{v}(S,\theta)\right)\nabla_\theta\hat{v}(S,\theta)$$

where

$$v_\pi(S) = \begin{cases} G_t & \text{in MC} \\ R_{t+1} + \gamma\hat{v}(S_{t+1},\theta) & \text{in TD(0)} \\ G_t^\lambda & \text{in TD}(\lambda) \end{cases}$$

Action-Value Function Approximation

Goal: find a parameter θ that minimizes the mean-squared error between approximate value function $\hat{q}(S, A, \theta)$ and the true action-value function $q_\theta(S, A)$.

$$J(\theta) = \mathbb{E}_\pi[(q_\pi(S) - \hat{q}(S, A, \theta))^2]$$

Gradient descent:

$$\Delta\theta = -\frac{1}{2}\alpha\nabla_\theta J(\theta) = \alpha\mathbb{E}[(q_\pi(S, A) - \hat{q}(S, A, \theta))\nabla_\theta\hat{q}(S, A, \theta)]$$

Stochastic Gradient Descent samples:

$$\Delta\theta = \alpha\left(q_\pi(S, A) - \hat{q}(S, A, \theta)\right)\nabla_\theta\hat{q}(S, A, \theta)$$

where

$$q_\pi(S, A) = \begin{cases} G_t & \text{in MC} \\ R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, \theta) & \text{in TD(0)} \\ G_t^\lambda & \text{in TD}(\lambda) \end{cases}$$

Episodic Semi-gradient Sarsa for Control

- Input: a differentiable function $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^n \to \mathbb{R}$
- Initialize value-function weights $\theta \in \mathbb{R}^n$
- Repeat (for each episode)
  - $S, A \leftarrow$ initial state and action of episode (e.g., ϵ-greedy)
  - Repeat (for each step of episode):
    - Take action A, observe R, S'.
    - Choose A' as a function of $\hat{q}(S', \cdot, \theta)$ (e.g. ϵ-greedy)
    - $\theta \leftarrow \theta + \alpha[R + \gamma\hat{q}(S', A', \theta) - \hat{q}(S, A, \theta)]\nabla\hat{q}(S, A, \theta)$
    - $S \leftarrow S'$
    - $A \leftarrow A'$

Continuing Tasks

Quality of a policy is instead defined as the average rate of reward:

$$\eta(\pi) = \lim_{t \to \infty} \mathbb{E}[R_t | A_{0:t-1} \sim \pi]$$

and returns are defined in terms of differences between rewards and average reward:

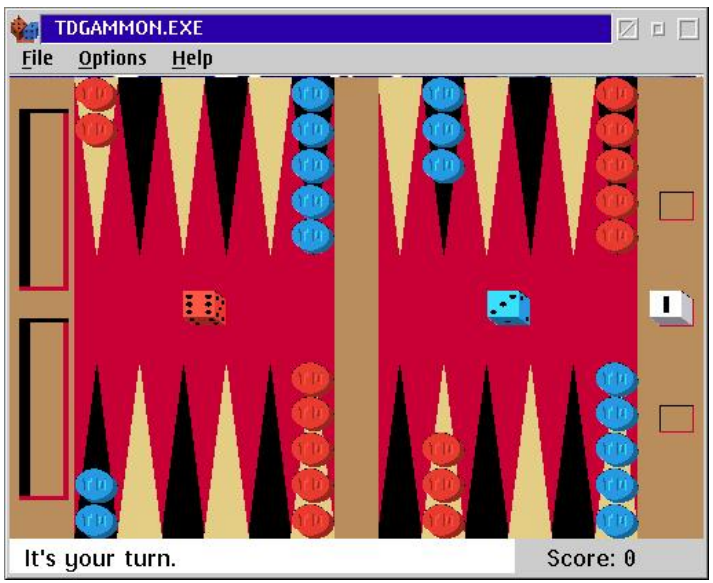
$$G_t = R_{t+1} - \eta(\pi) + R_{t+2} - \eta(\pi) + R_{t+3} - \eta(\pi) + \cdots$$

Differential Semi-gradient Sarsa for Control

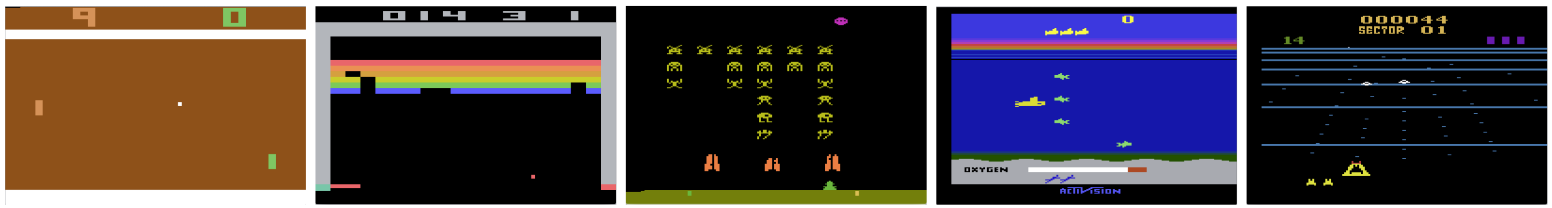
- Input: a differentiable function $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^n \to \mathbb{R}$
- Initialize value-function weights $\theta \in \mathbb{R}^n$
- Initialize average reward $\bar{R}$ arbitrarily.
- Repeat (for each episode)
  - $S, A \leftarrow$ initial state and action of episode (e.g., ϵ-greedy)
  - Repeat (for each step of episode):
    - Take action A, observe R, S'.
    - Choose A' as a function of $\hat{q}(S', \cdot, \theta)$ (e.g. ϵ-greedy)
    - $\theta \leftarrow \theta + \alpha[R - \bar{R} + \hat{q}(S', A', \theta) - \hat{q}(S, A, \theta)]\nabla\hat{q}(S, A, \theta)$
    - $\bar{R} \leftarrow \bar{R} + \beta[R - \bar{R} + \hat{q}(S', A', \theta) - \hat{q}(S, A, \theta)]$
    - $S \leftarrow S'$
    - $A \leftarrow A'$

TD-Gammon

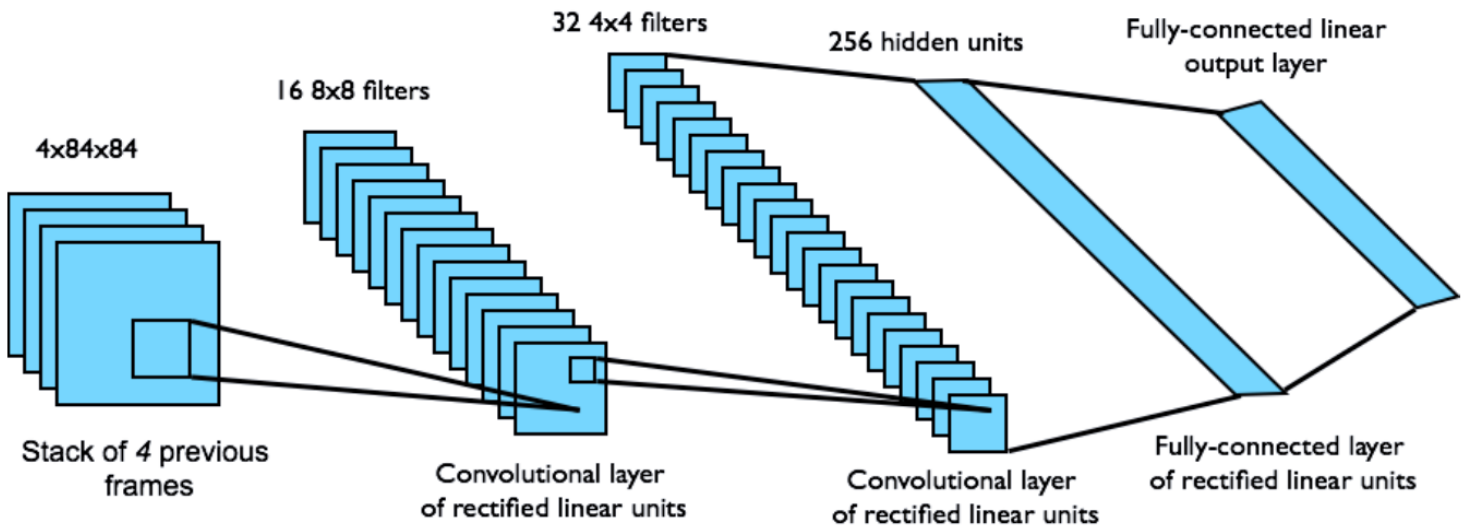
- Used a multi-layer artificial neural network trained by TD(λ) to evaluate each possible move.
- Achieved a level of play just slightly below that of the top human backgammon player in 1992.
- Found new strategies.



Google Deepmind's Deep Q-Network (DQN)



Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." (2013)



David Silver's presentation on function approximation

Policy-Gradient Methods

Previously we approximated

$$Q_\theta(s,a) \approx Q^\pi(s,a)$$

and then generated a policy from the approximated action-value function.

In some cases its better to directly parametrize the policy

$$\pi_\theta(a|s,\theta) = \mathbb{P}(a|s,\theta)$$

which is more effective in high-dimensional or continuous action spaces. We then update θ using gradient descent:

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta J(\theta_t)$$

where J is a policy objective function.

Policy Evaluation

State-value function from initial state, s_1:

$$J_1(\theta) = V^{\pi_\theta}(s_1)$$

Average value

$$J_{\text{avV}}(\theta) = \sum_s d^{\pi_\theta}(s) V^{\pi_\theta}(s)$$

where $d^{\pi_\theta}(s)$ is the stationary Markov-chain distribution for π_θ.

Average reward per time-step

$$J_{\text{avR}}(\theta) = \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(s,a) \mathcal{R}_s^a$$

For any of the above (policy-gradient theorem):

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(S,A) Q^{\pi_\theta}(S,A)]$$

Reinforce Monte Carlo

Initialize θ

for each episode do $S_1, A_1, R_2, ..., S_{T-1}, A_{T-1}, R_T \sim \pi_\theta$ do

for $t = 1$ to $T - 1$

$\theta = \theta + \alpha \nabla_\theta \log \pi_\theta(S, A) G$

where G is the sample return from state S_t

end for

end for

return θ

end function

Q - Action Critic

With linear value function approximation $Q_w(S, A) = \phi(S, A)^T w$

Initialize s, θ
Sample $a \sim \pi_\theta$
for each step do
    sample reward R', sample transition S'
    sample action $A' \sim \pi_\theta(A'|S')$
    $\delta = R' + \gamma Q_w(S', A') - Q_w(S, A)$
    $\theta = \theta + \alpha \nabla_\theta \log \pi_\theta(A'|S') Q_w(S, A)$
    $w \leftarrow \beta \delta \phi(S, A)$
$A \leftarrow A'$, $S \leftarrow S'$
end for

Exercise: OpenAI Gym

OpenAI Gym:

- A toolkit for developing and comparing reinforcement-learning algorithms.
- From simulated robots to Atari games.
- A site for comparing and reproducing results.

Task:

- Run a RL algorithm on one of the OpenAI-gym examples.
- I suggest that you try an already working implementation:
  - Pong
  - Breakout
  - Space Invaders