Optimal Control

ME598/494 Lecture

Max Yi Ren

Department of Mechanical Engineering, Arizona State University

November 30, 2017

Outline

Optimal control problems

Optimal control in general: Optimization problems that have an infinite number of variables (along time).

Optimal control (in **engineering**): Physics-based system model exists (system identification is done). The goal: Find control policies for engineering systems.

Examples:

1. Optimal control of hybrid powertrain (power split between engine and motors for minimal fuel consumption under sustained battery state of charge)
2. Drone maneuvering to follow a pre-defined trajectory (optimal control gains for a PID controller, or Linear Quadratic Regulator)
3. Robot gait design for energy efficient movement

Optimal control problems

Reinforcement learning: Physics-based model does not exist but observations can be made. The goal: Understand how machines should learn to behave in a new environment.

Examples:

1. Learn to play various board and video games
2. Learn to grasp objects
3. Learn new locomotions for unknown environments

Optimal control problems

There are overlaps between the two fields: Physics-based models or prior knowledge can be integrated into reinforcement learning; RL algorithms can be applied to engineering control problems where uncertainty is high.

Optimal control is like finding a path through a maze where the maze, albeit complicated, is visible. Reinforcement learning is like finding a path through a maze, without initially knowing how it looks like. The latter requires exploration.

Reinforcement learning is of significant importance as it leads to adaptive machines.

At this moment, we only have algorithms for solving individual problems. Knowledge transfer/transfer learning/meta-reinforcement learning, i.e., solving new problems faster by practicing on similar problems is a key area of research.

Markov chain

A reward in the future is not worth as much as its current value.

An example of a Markov chain:

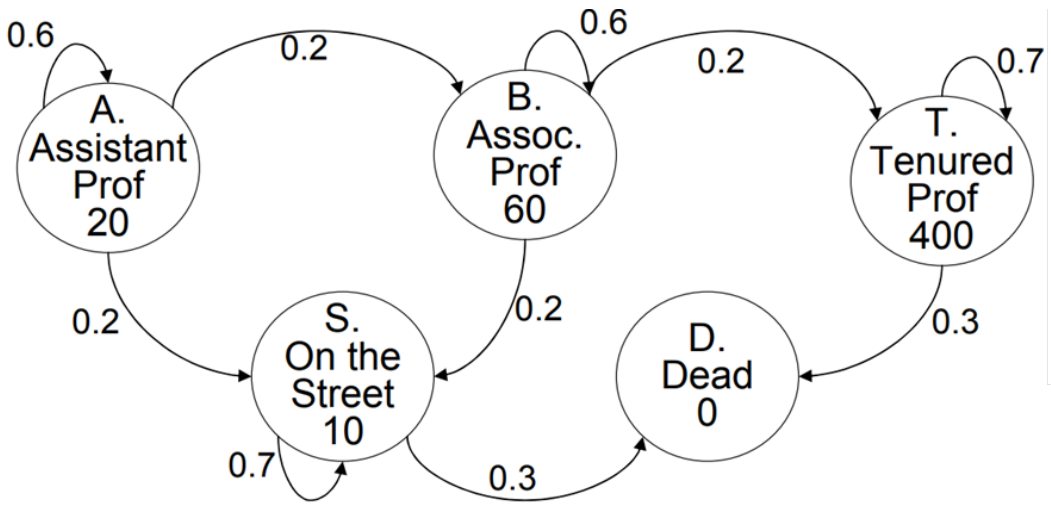


Figure: from Andrew W. Moore's notes

Markov chain

A Markov chain

1. has a set of **states**
2. has a **transition probability matrix**
3. each state has a **reward**
4. a discount factor γ

What are the expected value of being in each state of the Markov system?
(Hint: you need to solve a system of linear equations)

value iteration

Instead of solving a system of linear equations, one can use value iteration:

1. $V^1(s_i)$ = expected discounted sum of rewards over the next 1 time step
2. $V^2(s_i)$ = expected discounted sum of rewards over the next 2 time step
3. ...

Or

1. $V^1(s_i) = r_i$, r_i = reward at state i.
2. $V^2(s_i) = r_i + \gamma \sum_{j=1}^{N} p_{ij} V^1(s_j)$, N = #states, p_{ij} = transition probability
3. ...
4. $V^{k+1}(s_i) = r_i + \gamma \sum_{j=1}^{N} p_{ij} V^k(s_j)$

Markov Decision Process

A Markov decision process has

1. a set of states $\mathcal{S}$
2. a set of **actions** $\mathcal{A}$
3. a transition probability function $T(s', a, s)$
4. a reward function $r(s, a)$
5. a discount factor γ

Markov Decision Process

An example of a Markov decision process:

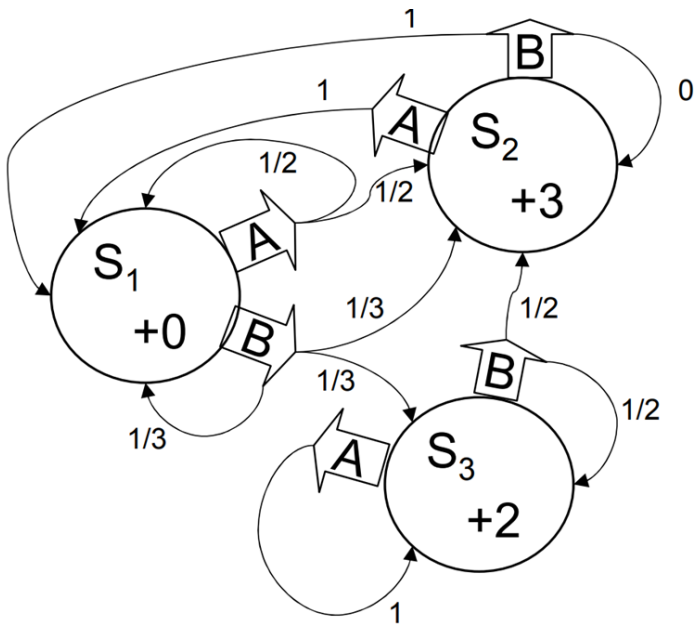


Figure: from Andrew W. Moore's notes

Optimal value function

An optimal control policy leads to an optimal value function: $V^*(s_i) =$ expected discounted future rewards, starting from state s_i, assuming we use the optimal policy.

Compute optimal value function through value iteration: Define $V^n(s_i) =$ maximum possible expected sum of discounted rewards starting at s_i for n time steps. $V^1(s_i) = r_i$

Bellman's Equation: $V^{n+1}(s_i) = \max_k [r_i + \gamma \sum_{j=1}^{N} P_{ij}^k V^n(s_j)]$

Value iteration for solving MDPs:

1. Compute $V^1(s_i)$ for all i
2. Compute $V^2(s_i)$ for all i
3. ...
4. Compute $V^n(s_i)$ for all i until convergence

This is also called **Dynamic Programming**

Policy Iteration

Define $\pi(s_i) =$ action taken at s_i. $\pi^t =$ policy at the tth iteration.

Policy iteration:

1. $n = 0$
2. For all i, compute $V^n(s_i) =$ long term reward starting at s_i using π^n
3. $\pi^{n+1}(s_i) = \arg\max_k [r_i + \gamma \sum_j P_{ij}^k V^n(s_j)]$
4. Go to 2 until convergence

Policy Iteration vs. Value Iteration

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

2. Policy Evaluation
   Repeat
      $\Delta \leftarrow 0$
      For each $s \in \mathcal{S}$:
         $temp \leftarrow v(s)$
         $v(s) \leftarrow \sum_{s'} p(s'|s, \pi(s))\Big[r(s, \pi(s), s') + \gamma v(s')\Big]$
         $\Delta \leftarrow \max(\Delta, |temp - v(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
      $temp \leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s'} p(s'|s, a)\Big[r(s, a, s') + \gamma v(s')\Big]$
      If $temp \neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return v and π; else go to 2

Figure 4.3: Policy iteration (using iterative policy evaluation) for v_*. This algorithm has a subtle bug, in that it may never terminate if the policy continually switches between two or more policies that are equally good. The bug can be fixed by adding additional flags, but it makes the pseudocode so ugly that it is not worth it. :-)

finding optimal value function

Initialize array v arbitrarily (e.g., $v(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat
   $\Delta \leftarrow 0$
   For each $s \in \mathcal{S}$:
      $temp \leftarrow v(s)$
      $v(s) \leftarrow \max_a \sum_{s'} p(s'|s, a)[r(s, a, s') + \gamma v(s')]$
      $\Delta \leftarrow \max(\Delta, |temp - v(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, π, such that
   $\pi(s) = \arg\max_a \sum_{s'} p(s'|s, a)\Big[r(s, a, s') + \gamma v(s')\Big]$

Figure 4.5: Value iteration.

one policy update (extract policy from the optimal value function

Figure: from stackoverflow and Sutton & Barto

Policy Iteration vs. Value Iteration

Main difference:

1. Policy iteration includes repeated policy evaluation and policy improvement
2. Value iteration includes finding optimal value function and policy extraction
3. Policy evaluation (in policy iteration) and finding optimal value function (in value iteration) are similar except that the latter requires an iteration over all actions (which can be costly!)

Which one to use:

1. Lots of actions? Policy iteration
2. Already have a fair policy? Policy iteration
3. Few actions? Value iteration

Q-learning

Value iteration and policy iteration assumes that the state transitions and the rewards are known. Reinforcement learning solves problems where these are not known to an agent at the beginning.

Two types of RL algorithms exist: **Model-based** methods model the transition and reward functions so that policy or value iteration can be performed. **Model-free** methods derive optimal policies without modeling the transition and reward functions.

We start with Q-learning, which is model-free. A Q function $Q(s,a)$ measures the long-term reward of action a at state s. The optimal Q function derives the optimal control policy, i.e., $V^*(s) = \max_a Q^*(s,a)$, and $\pi^*(s) = arg\max_a Q^*(s,a)$.

From Bellman equation, we have $Q^*(s,a) = r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V^*(s')$.

Temporal-Difference

One approach to the derivation of Q^* is through Q-learning (off-policy Temporal-Difference):
$Q(s,a) = (1-\alpha)Q(s,a) + \alpha(r(s,a) + \gamma \max_{a'} Q(s',a'))$, where s' is the observed next state, and α is the learning rate.

Q-learning algorithm:

1. Initialize Q table randomly
2. Run agent with policy $\pi(s) = arg\max_a Q(s,a)$ with probability $1-\epsilon$, or other random actions at probability ϵ for T time steps (an episode), for each step:
   - 2.1 Take action a, observe r, s'
   - 2.2 $Q(s,a) = (1-\alpha)Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a')]$
3. Go to 2

Here we introduced ϵ-greedy policy to balance exploitation and exploration.

SARSA

SARSA is an "on-policy" TD algorithm. The only difference is that it updates Q using the actual observed reward:

1. Initialize Q table randomly
2. Run agent with ϵ-greedy for an episode, for each step:
   - 2.1 Take action a, observe r, s'
   - 2.2 Choose a' from s' using ϵ-greedy
   - 2.3 $Q(s,a) = (1-\alpha)Q(s,a) + \alpha[r + \gamma Q(s',a')]$
3. Go to 2

On-policy vs. off-policy

Off-policy algorithms update Q values using the maximum of the existing Q values, rather than that of the actual actions taken. Thus they are off from the actual policy. With large enough exploration of the state and action spaces, such algorithm will converge to the optimal policy.

On-policy algorithms update Q values using the actual action taken (Note in SARSA we need the last “A”ction). They are usually faster to converge but may not reach the theoretical optimal policy.

If the action space is continuous, Q learning cannot be directly applied, since the max operation will become a nested optimization.

On-policy vs. off-policy

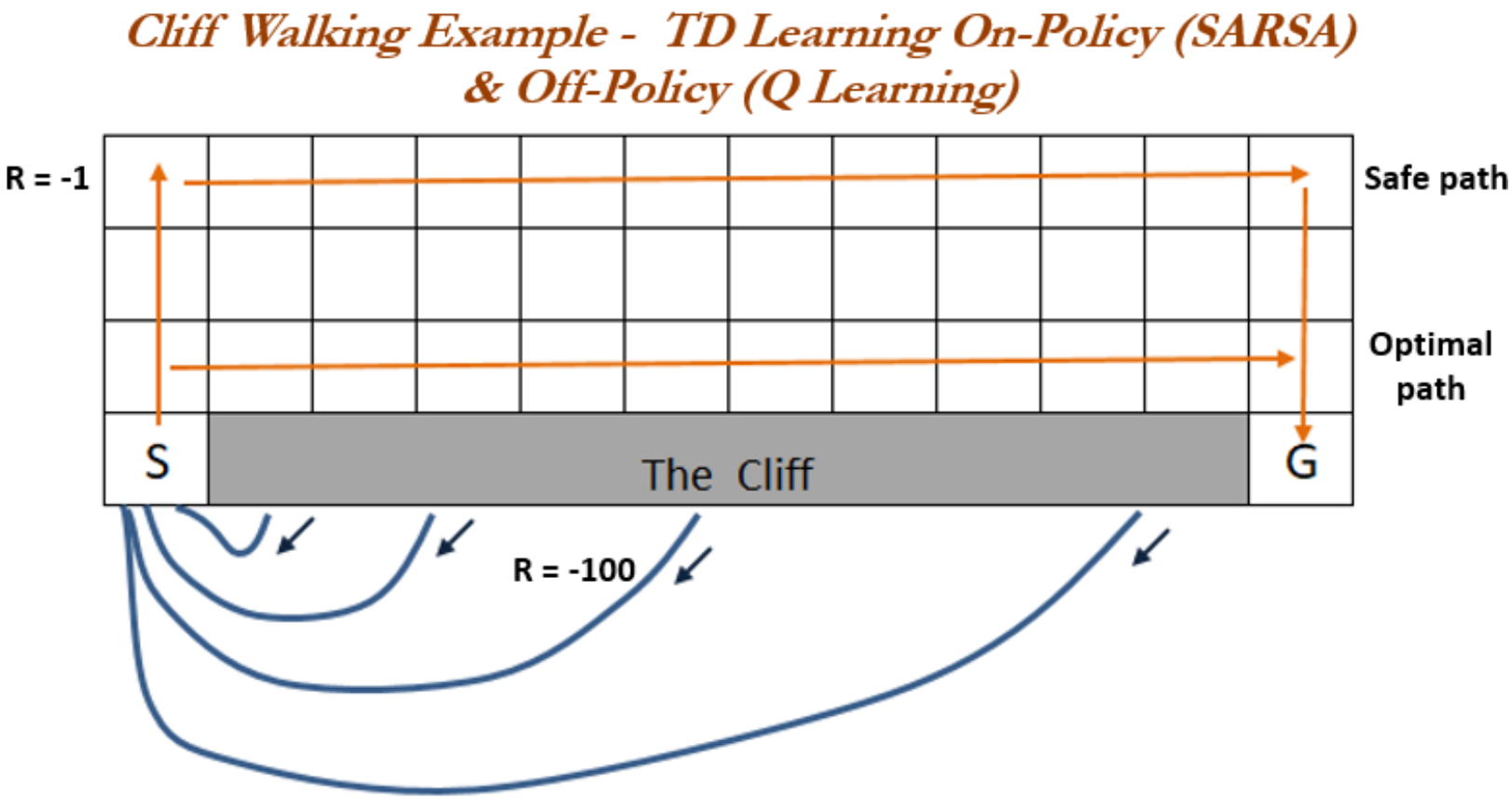


Figure: Q-learning reaches the optimal path; SARSA reaches the safe path.

From Q-learning to deep Q network

Q-learning algorithms such as TD have slow convergence when state and action spaces are large, due in large part to the fact that one cannot afford to exhaustively explore the spaces to populate the Q table. One solution to this is to learn an approximated Q function rather than updating a table. However, the real unknown Q function is often non-smooth. Deep neural networks come to rescue in these cases (sort of...).

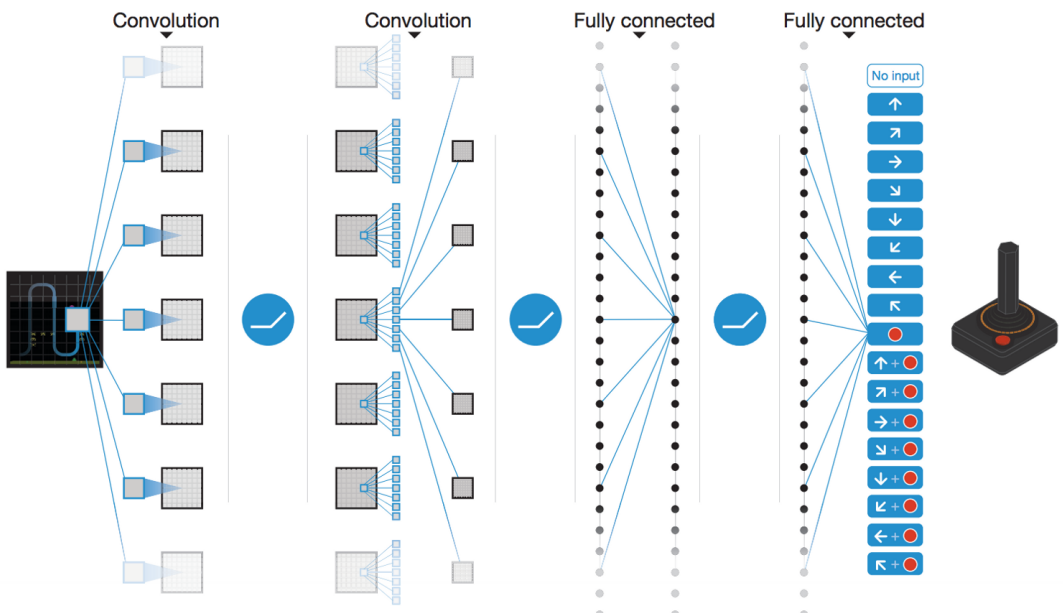


Figure: Deep Q-learning through convolutional neural network

Update the Q function by gradient descent

The loss function to be minimized is:

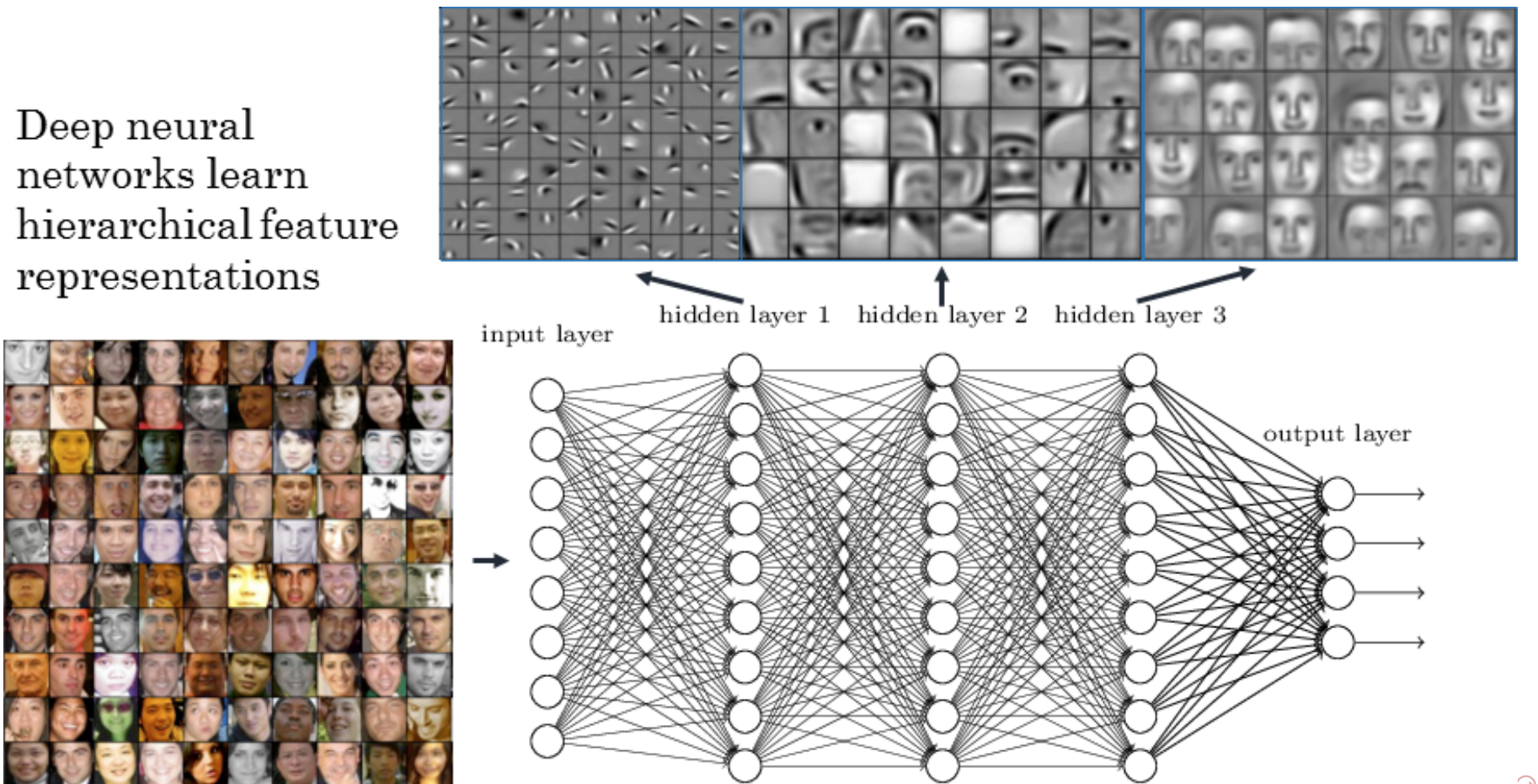
$$L(\theta) = \sum_{\{<s_i, a_i, r_i, s_i'>\}} (Q(s_i, a_i; \theta) - y_i)^2, \tag{1}$$

where $y_i = r_i + \gamma Q(s_i', arg\max_a Q(s_i', a); \theta)$. s_i, a_i, r_i, and s_i' are the state, action, reward, and next state, respectively. The update is off-policy. The dependence of y_i on θ is often neglected, which causes instability of the training (see "separate target network").

Deep Q Network

Components of a deep Q Network

1. **Convolutional layers** are often used to extract features from imagery inputs that can help determine the output.



Deep Q Network

Components of a deep Q network

2. **Experience relay** is referred to the training technique where previous memories (i.e., tuples of ¡state, action, reward, state¿) are randomly sampled to ensure that the agent does not only learn from the most recent experience.

3. A **separate target network** is used for calculating the target values y_i. This network takes a copy of the original Q function, and only updates its weights according to the Q function slowly. This addresses the instability issue of the training of the deep Q network.

Challenges with DQN

DQN associates high-dimensional visual inputs with actions. However, the robustness of convolutional neural network is still under question. Thus, controllers based on DQN can surfer from adversarial attacks, and can easily crash due to minor changes in the visual settings of the problem, see `https://www.youtube.com/watch?v=QHcAlAprFxA`.

Actor-Critic

Critic-only algorithms (e.g., Q-learning) rely only on approximating the optimal value functions (following the Bellman equation), and find policies indirectly.

Actor-only algorithms (e.g., policy search) uses simulation to approximate the gradient for improving a policy. The approximated gradient often has large variance, and changing the policy will require new gradient approximation and new simulations.

Actor-Critic

Actor-critic algorithms combine the strength of the two: We keep track of an approximation of the Q (or value) function, and use this approximation to derive the improve directions of a policy.

This is analogous to response surface method, where in each iteration, we learn a response surface (Q or value function) to approximate the unknown reward landscape, find a solution (policy) for this surface, which then helps to get more data to refine the response surface.

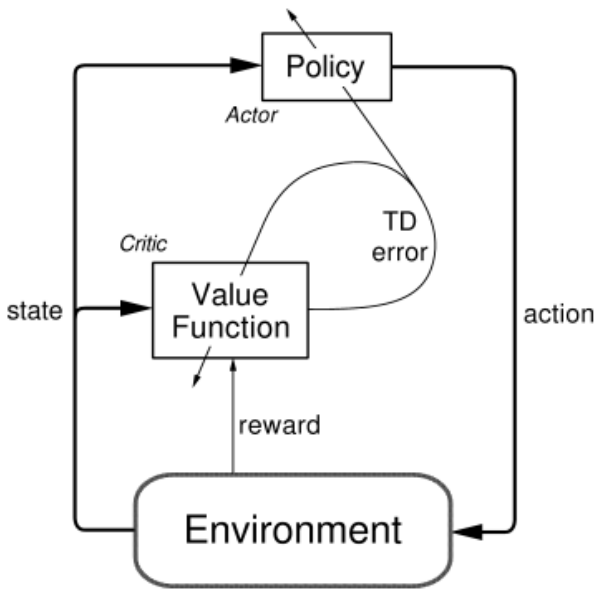


Figure: Actor-critic learning

Actor-Critic

Pseudo code for actor-critic learning

1. Initialize Q function $Q(s, a; \theta_q)$ and deterministic policy $\mu(s, \theta)$ with random θ_q and θ
2. Run an episode to collect tuples $\{< s_i, a_i, r_i, s_i' >\}$ for T time steps
3. Set $d\theta_q = 0$, $d\theta = 0$
4. For $i = T, ..., 1$:
   - 4.1 Accumulate $d\theta_q$: $d\theta_q = d\theta_q + \nabla_{\theta_q} L_q(\theta_q)$, where $L_q(\theta_q) = (Q(s_i, a_i; \theta_q) - y_i)^2$, and $y_i = r_i + \gamma Q(s_i'; \theta_q)$
   - 4.2 Accumulate $d\theta$: $d\theta = d\theta + \nabla_\theta Q(s_i, \mu(s_i, \theta); \theta_q)$
5. update θ_q and θ using $d\theta_q$ and $d\theta$, respectively
6. Go to 2

Summary

- Markov chain and Markov decision process - state, action, reward, transition, discount
- Known state transition and rewards - value/policy iteration
- Unknown state transition and rewards - Q-learning (SARSA, policy search)
- Large state/action spaces - Learn Q/policy functions
- Actor-critic combines Q-learning and policy search