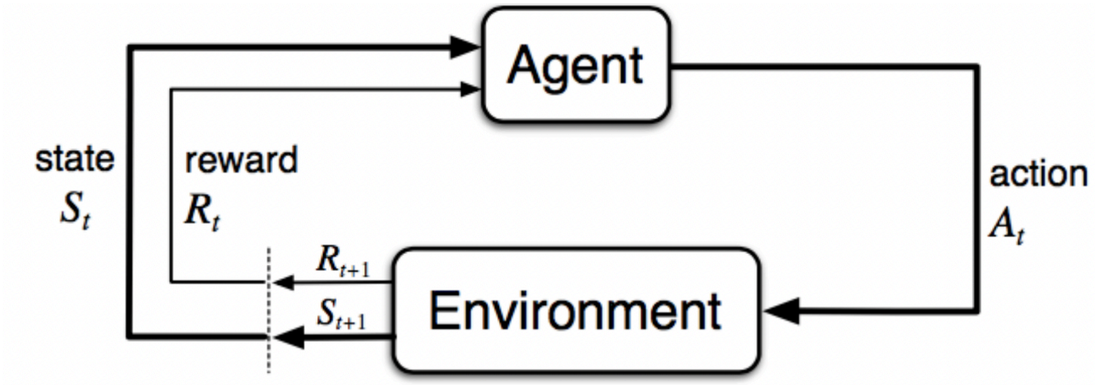
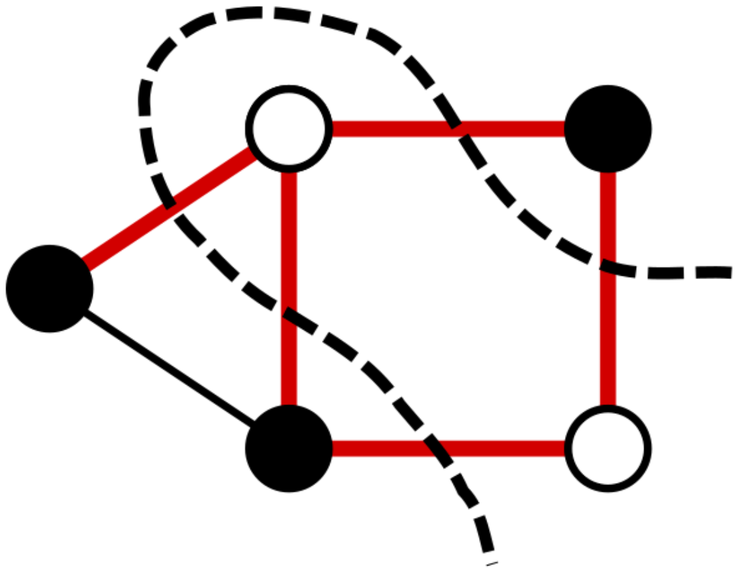
Reinforcement Learning for Combinatorial Optimization

CS618 - Theoretical Foundations of Machine Learning





Guntas Singh Saran (22110089)

Indian Institue of Technology Gandhinagar
Palaj, Gujarat - 382355

Major Source: https://lilianweng.github.io/posts/2018-02-19-rl-overview/

Combinatorial Optimization (CO) Problems

The Introduction

Definition 1. Let V be a set of elements and $f : V \mapsto \mathbb{R}$ be a cost function. *Combinatorial optimization problem* aims to find an optimal value of the function f and any corresponding optimal element that achieves that optimal value on the domain V.

Typically the set V is finite, in which case there is a global optimum, and, hence, a trivial solution exists for any CO problem by comparing values of all elements $v \in V$. Note that the definition 1 also includes the case of decision problems, when the solution is binary (or, more generally, multi-class), by associating a higher cost for the wrong answer than for the right one.

Travelling Salesman Problem

Maximum Cut Problem

Minimum Vertex Cover Problem

Maximum Independent Set Problem

Bin Packing Problem

Source: Reinforcement Learning for Combinatorial Optimization: A Survey

Maximum Cut (Max Cut) Problem

Definition

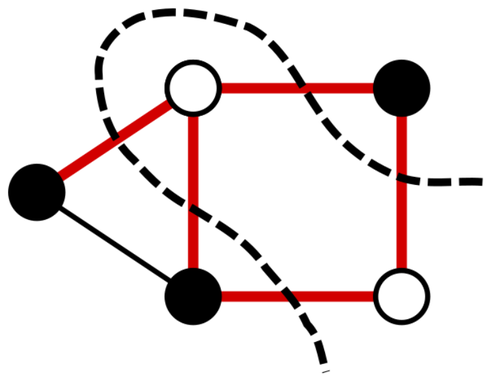
Given: undirected graph $G = (V, E)$, a **cut** in G is a subset $S \subseteq V$. Let $\overline{S} = V \backslash S$

Define: $E(S, \overline{S})$ denote the set of edges with one vertex in S and one vertex in $\overline{S}$.

Goal: to find the cut S that maximizes $|E(S, \overline{S})|$

W is a weight function $W : E \to \mathbb{R}$ assigning a weight $w_{ij} \geq 0$ to each edge $(i, j) \in E$.

$$\text{Maximize: } C(S, G) = \sum_{i \in S, j \in V \backslash S} w_{ij}.$$



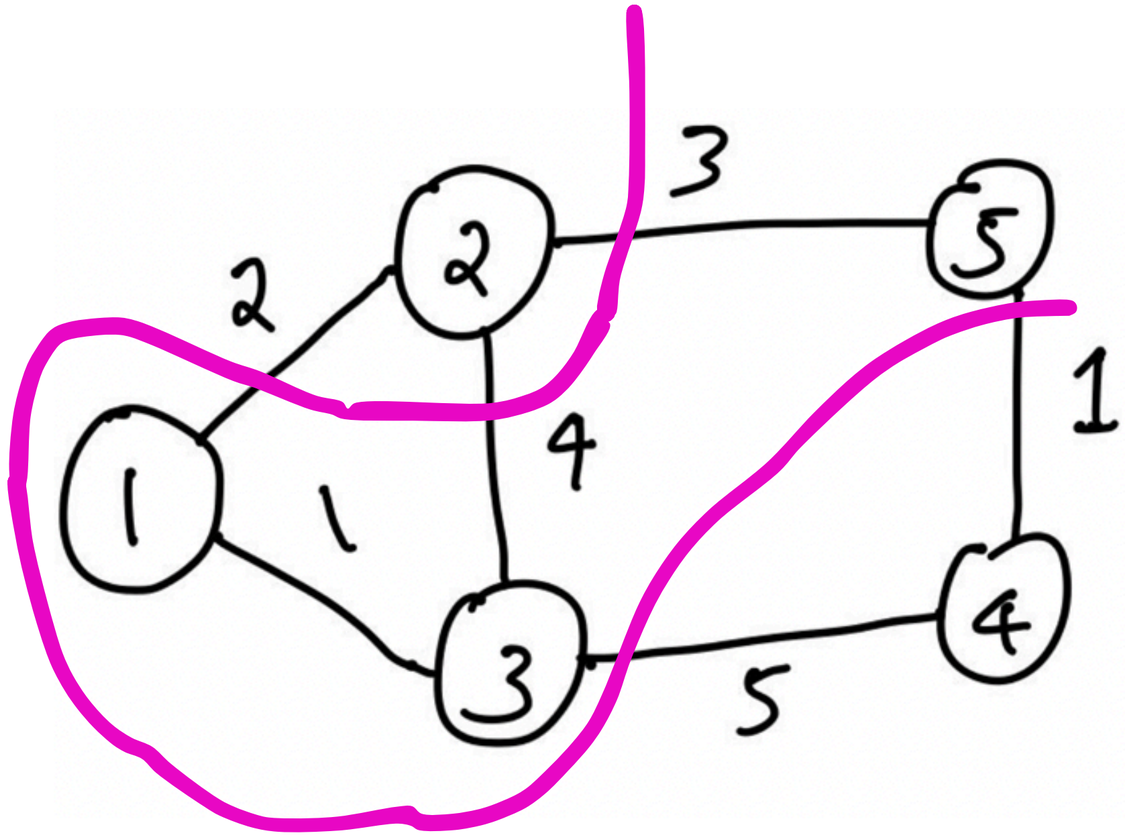
Maximum Cut (Max Cut) Problem

Finding the Optimal Cut

The problem is NP-complete.

1. For $S = \{1, 2, 3\}$, $\mathcal{W}(S) = 8$.
2. For $S = \{1, 2, 3, 4\}$, $\mathcal{W}(S) = 4$.
3. For $S = \emptyset$, $\mathcal{W}(S) = 0$.
4. For $S = \{1, 3, 5\}$, $\mathcal{W}(S) = 15$.

The maxcut is the cut $S = \{1, 3, 5\}$.



Max Cut using DQN

The Setup

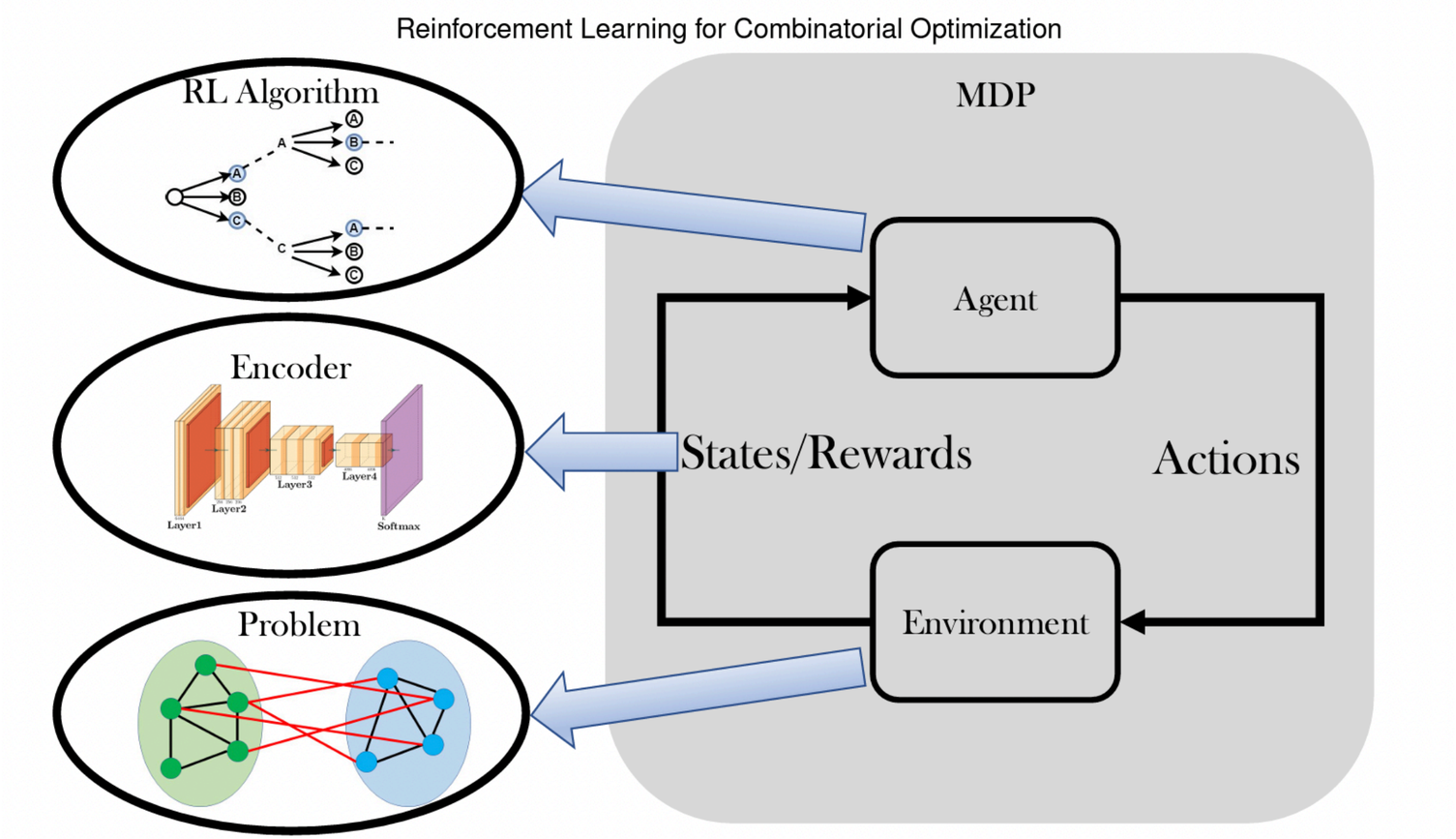
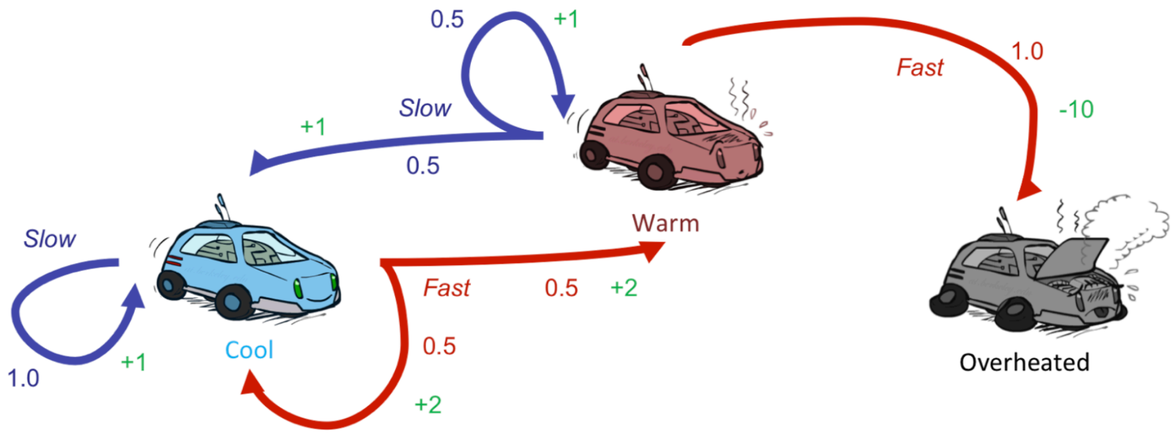


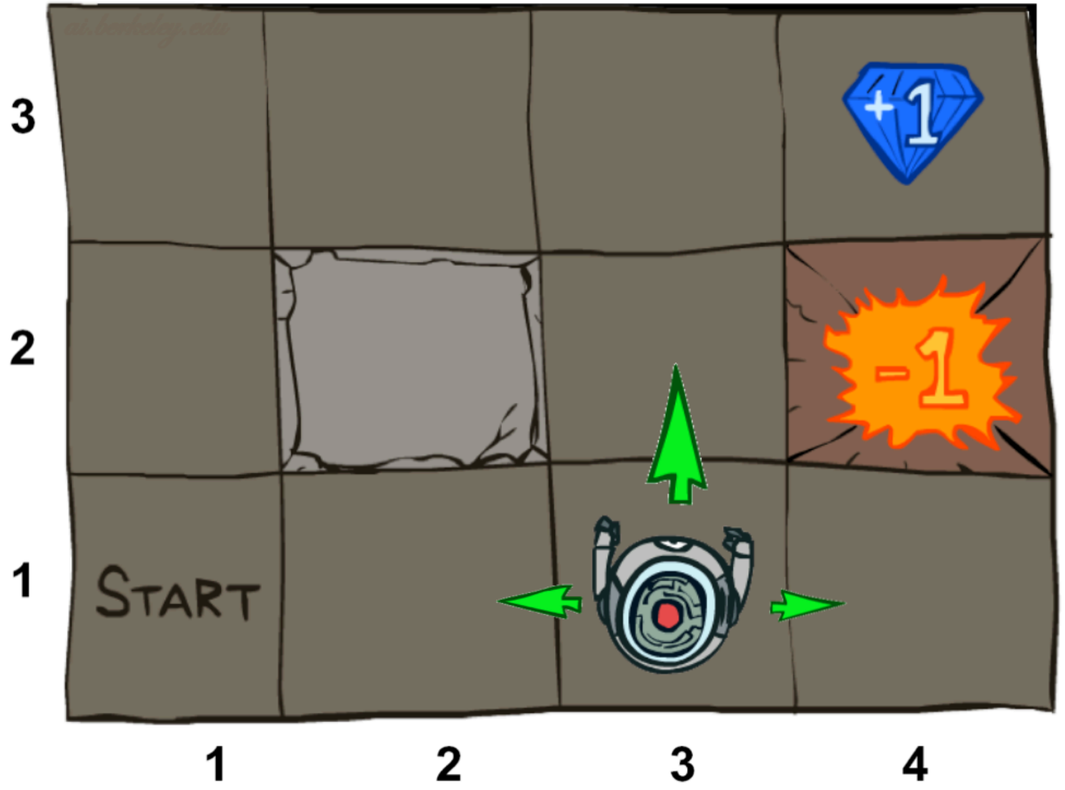
Figure 1: Solving a CO problem with the RL approach requires formulating MDP. The environment is defined by a particular instance of CO problem (e.g. Max-Cut problem). States are encoded with a neural network model (e.g. every node has a vector representation encoded by a graph neural network). The agent is driven by an RL algorithm (e.g. Monte-Carlo Tree Search) and makes decisions that move the environment to the next state (e.g. removing a vertex from a solution set).

Markov Decision Process (MDP)

The Basics

$$\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_1, \ldots, S_t]$$





Source: [CS188: Artifiicial Intelligence Course - UC Berkeley]

Markov Decision Process (MDP)

The Ingredients of an MDP

States: S

Actions: A

Transitions: $T(s'|s,a)$

Rewards: $R(s,a,s')$

Start state: s_0

Discount: γ

Policy: map of states to actions

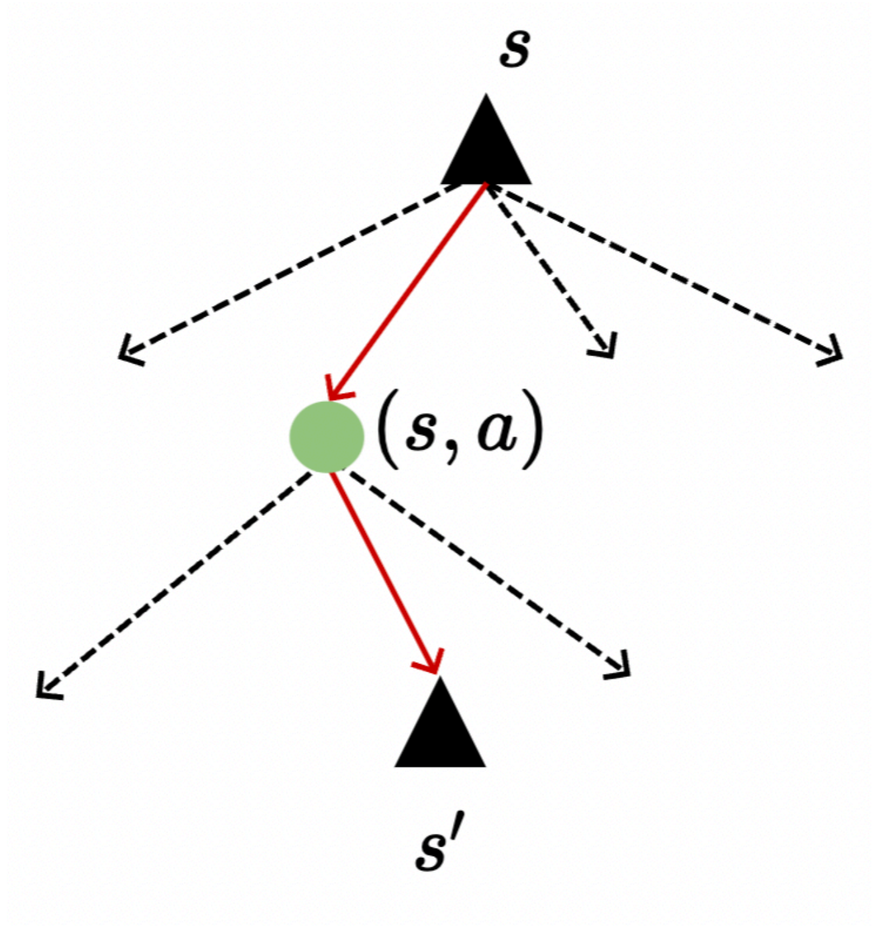
Utility: sum of discounted rewards

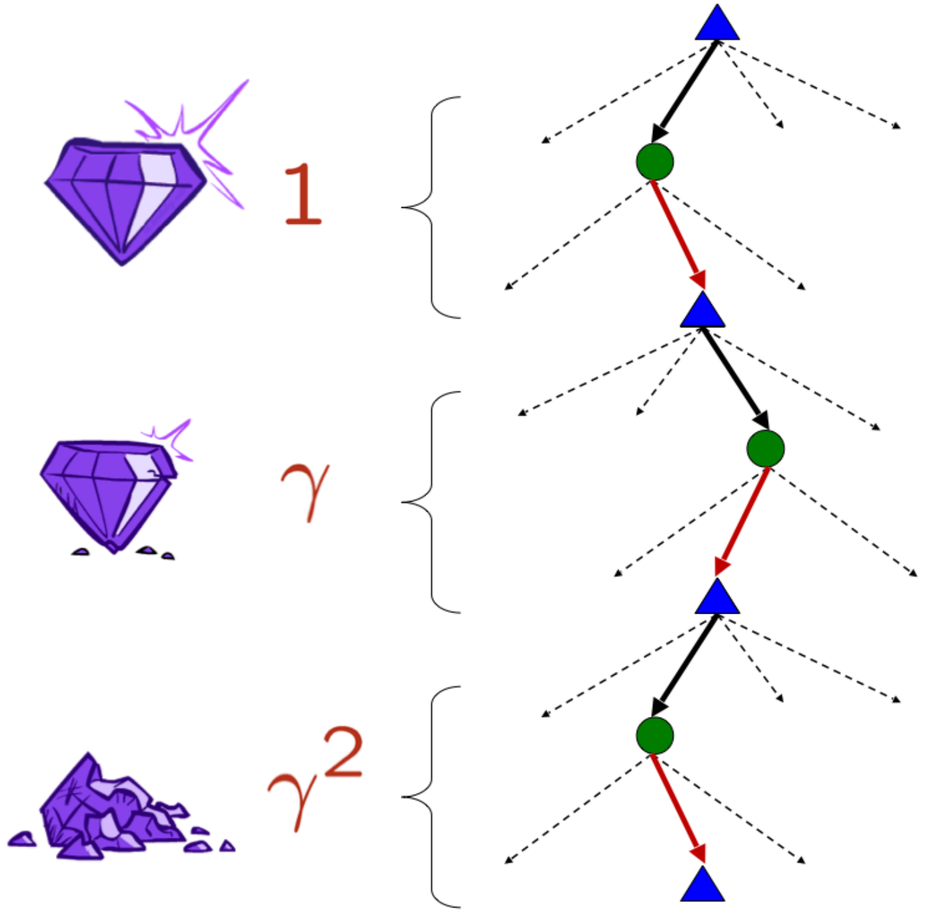
Values: expected future utility from a state

Q-values: expected future utility from a state+action pair

Markov Decision Process

The Discount Factor





Source: [CS188: Artifiicial Intelligence Course - UC Berkeley]

Optimal Value

The Basics

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

The **state-value** of a state s is the expected return if we are in this state at time t, $S_t = s$:

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

Similarly, we define the **action-value** ("Q-value"; Q as "Quality" I believe?) of a state-action pair as:

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

Additionally, since we follow the target policy π, we can make use of the probility distribution over possible actions and the Q-values to recover the state-value:

$$V_\pi(s) = \sum_{a \in \mathcal{A}} Q_\pi(s, a) \pi(a|s)$$

Source: https://lilianweng.github.io/posts/2018-02-19-rl-overview/

Optimal Value

$$V^*(s) = \max_a Q^*(s, a)$$

Optimal Value

$$V^*(s) = \max_a Q^*(s,a)$$

$$Q^*(s,a) = \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s') \right]$$

Optimal Value

$$V^*(s) = \max_a Q^*(s,a)$$

$$Q^*(s,a) = \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s') \right]$$

$$V^*(s) = \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s') \right]$$

Optimal Value

$$V^*(s) = \max_a Q^*(s,a)$$

$$Q^*(s,a) = \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s') \right]$$

$$V^*(s) = \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V^*(s') \right]$$

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_k(s') \right]$$

What is Reinforcement Learning?

The Setting

The agent is acting in an **environment**. How the environment reacts to certain actions is defined by a **model** which we may or may not know. The agent can stay in one of many **states** ($s \in \mathcal{S}$) of the environment, and choose to take one of many **actions** ($a \in \mathcal{A}$) to switch from one state to another. Which state the agent will arrive in is decided by transition probabilities between states (P). Once an action is taken, the environment delivers a **reward** ($r \in \mathcal{R}$) as feedback.

The agent's **policy** $\pi(s)$ provides the guideline on what is the optimal action to take in a certain state with **the goal to maximize the total rewards**. Each state is associated with a **value** function $V(s)$ predicting the expected amount of future rewards we are able to receive in this state by acting the corresponding policy. In other words, the value function quantifies how good a state is. Both policy and value functions are what we try to learn in reinforcement learning.

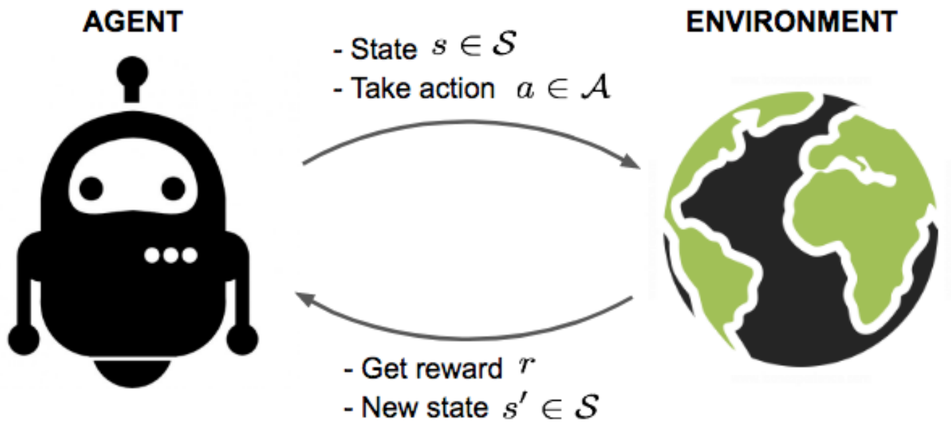
Source: https://lilianweng.github.io/posts/2018-02-19-rl-overview/

What is Reinforcement Learning?

The Setting

The interaction between the agent and the environment involves a sequence of actions and observed rewards in time, $t = 1, 2, \ldots, T$. During the process, the agent accumulates the knowledge about the environment, learns the optimal policy, and makes decisions on which action to take next so as to efficiently learn the best policy. Let's label the state, action, and reward at time step t as S_t, A_t, and R_t, respectively. Thus the interaction sequence is fully described by one **episode** (also known as "trial" or "trajectory") and the sequence ends at the terminal state S_T:

$$S_1, A_1, R_2, S_2, A_2, \ldots, S_T$$



Source: https://lilianweng.github.io/posts/2018-02-19-rl-overview/

What is Reinforcement Learning?

Terminology

- **Model-based**: Rely on the model of the environment; either the model is known or the algorithm learns it explicitly.
- **Model-free**: No dependency on the model during learning.

Q-Learning

The Basics

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} T(s, a, s') \left[R(s, a, s') + \gamma V_k(s') \right]$$

How to know this when T(s, a, s') and R(s, a, s') are UNKNOWN

Q-Learning

The Basics

- We want to improve our estimate of V by computing these averages:

$$V_{k+1}^{\pi}(s) \leftarrow \sum_{s'} T(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V_k^{\pi}(s')]$$

- Idea: Take samples of outcomes s' (by doing the action!) and average

$$sample_1 = R(s, \pi(s), s'_1) + \gamma V_k^{\pi}(s'_1)$$
$$sample_2 = R(s, \pi(s), s'_2) + \gamma V_k^{\pi}(s'_2)$$
$$\cdots$$
$$sample_n = R(s, \pi(s), s'_n) + \gamma V_k^{\pi}(s'_n)$$

$$V_{k+1}^{\pi}(s) \leftarrow \frac{1}{n} \sum_i sample_i$$

Q-Learning

The Basics

- We'd like to do Q-value updates to each Q-state:

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \max_{a'} Q_k(s',a') \right]$$

  - But can't compute this update without knowing T, R

- Instead, compute average as we go
  - Receive a sample transition (s, a, r, s')
  - This sample suggests

$$Q(s,a) \approx r + \gamma \max_{a'} Q(s',a')$$

  - But we want to average over results from (s, a)
  - So keep a running average

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + (\alpha) \left[r + \gamma \max_{a'} Q(s',a') \right]$$

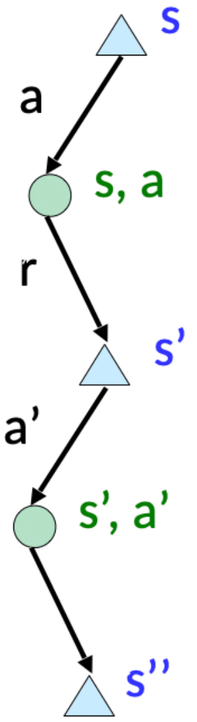
Q-Learning

The Basics

- Model-free (temporal difference) learning
  - Experience world through episodes

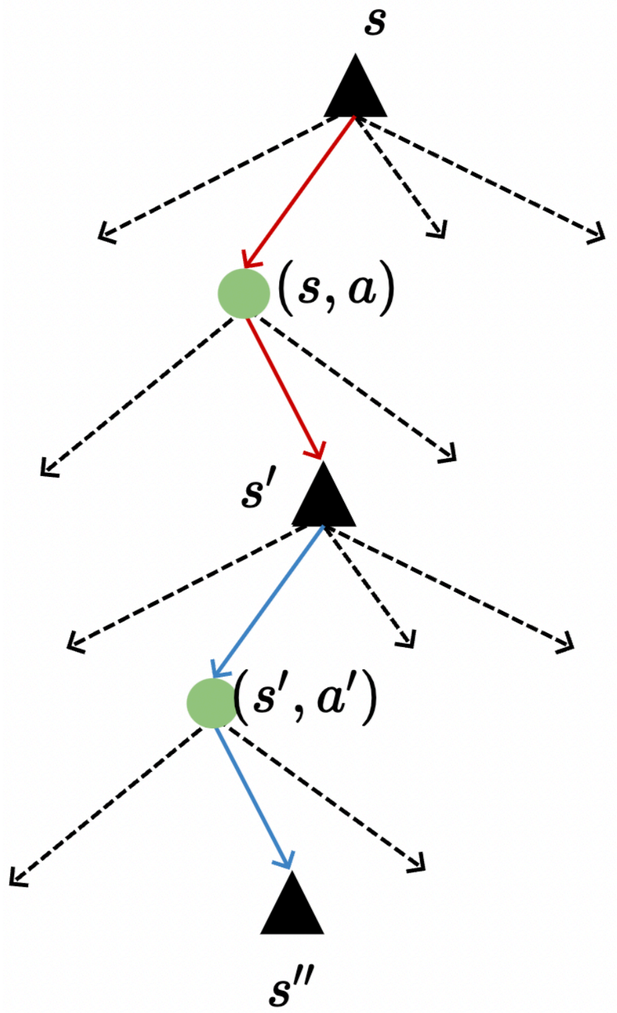
$$(s, a, r, s', a', r', s'', a'', r'', s'''' \ldots)$$

  - Update estimates each transition (s, a, r, s')
  - Over time, updates will mimic Bellman updates



Q-Learning

Q-Learning



Input: S, A and a sequence

$$s_0, a_0, r_0; s_1, a_1, r_1; \cdots$$

In particular, T and R are unknown.

Noisy Estimate of Q:

$$Q^+(s,a) = r_t + \gamma \max_{a'} Q(s', a')$$

$$\delta = Q^+(s,a) - Q(s,a)$$

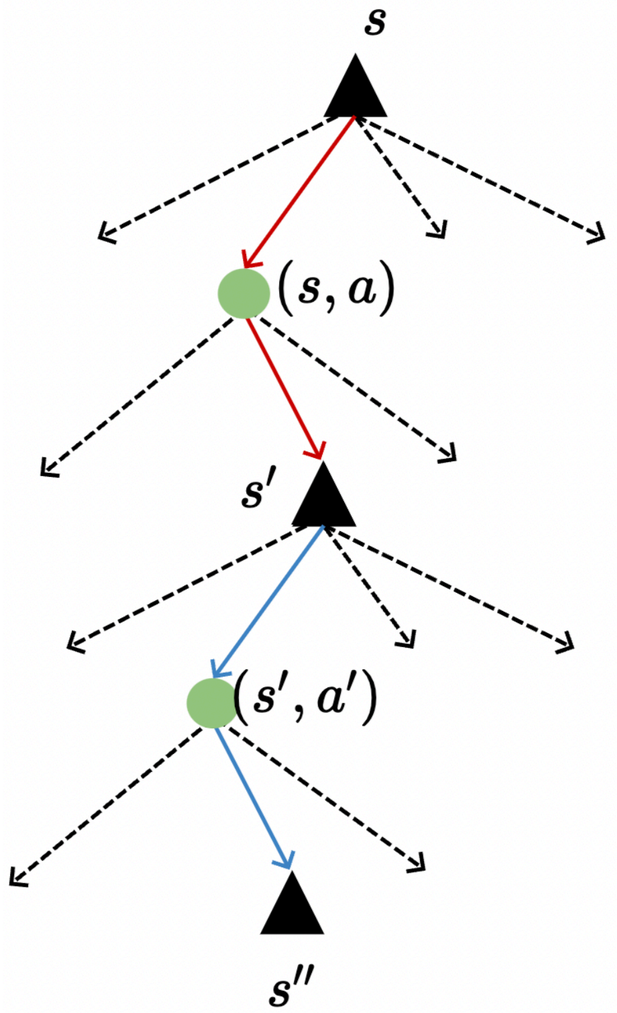
$$Q(s,a) = Q(s,a) + \alpha\delta$$

Policy Improvement

$$\pi^\epsilon(s) \triangleq \begin{cases} \text{argmax}_a\, Q^\pi(s_e a), & \text{with probability } 1-\epsilon \\ \text{UniformRandom}(\mathcal{A}), & \text{with probability } \epsilon \end{cases}$$

Q-Learning

SARSA



Input: S, A and a sequence

$$s_0, a_0, r_0; s_1, a_1, r_1; \cdots$$

In particular, T and R are unknown.

Noisy Estimate of Q:

$$Q^+(s,a) = r_t + \gamma Q(s', a')$$
$$\delta = Q^+(s,a) - Q(s,a)$$
$$Q(s,a) = Q(s,a) + \alpha\delta$$

Policy Improvement

$$\pi^\epsilon(s) \triangleq \begin{cases} \text{argmax}_a\, Q^\pi(s_e a), & \text{with probability } 1-\epsilon \\ \text{UniformRandom}(\mathcal{A}), & \text{with probability } \epsilon \end{cases}$$

Q-Learning

Algorithm Q-Learning Off Policy

while True **do**
  Initialize s_0, $\alpha, t = 0$
  for $t = 0, 1, \ldots$ **do**
    Obtain a_t using ϵ-greedy
    Obtain a reward r_{t+1} and s_{t+1} from the env
    Obtain a_{t+1} from greedy
    $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$
  end for
end while

Source: Reinforcement Learning: Sutton & Barto

SARSA

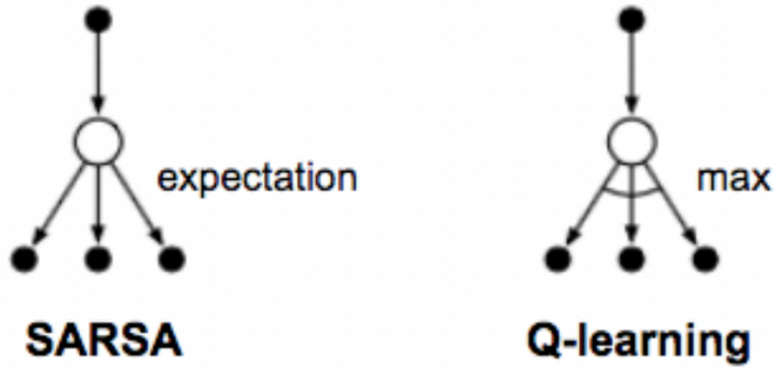
Algorithm SARSA On Policy

while True **do**
  Initialize s_0, $\alpha, t = 0$
  for $t = 1, 2, \ldots$ **do**
    Obtain a_t using ϵ-greedy
    Obtain a reward r_{t+1} and s_{t+1} from the env
    Obtain a_{t+1} from ϵ-greedy
    $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$
  end for
end while

Source: Reinforcement Learning: Sutton & Barto

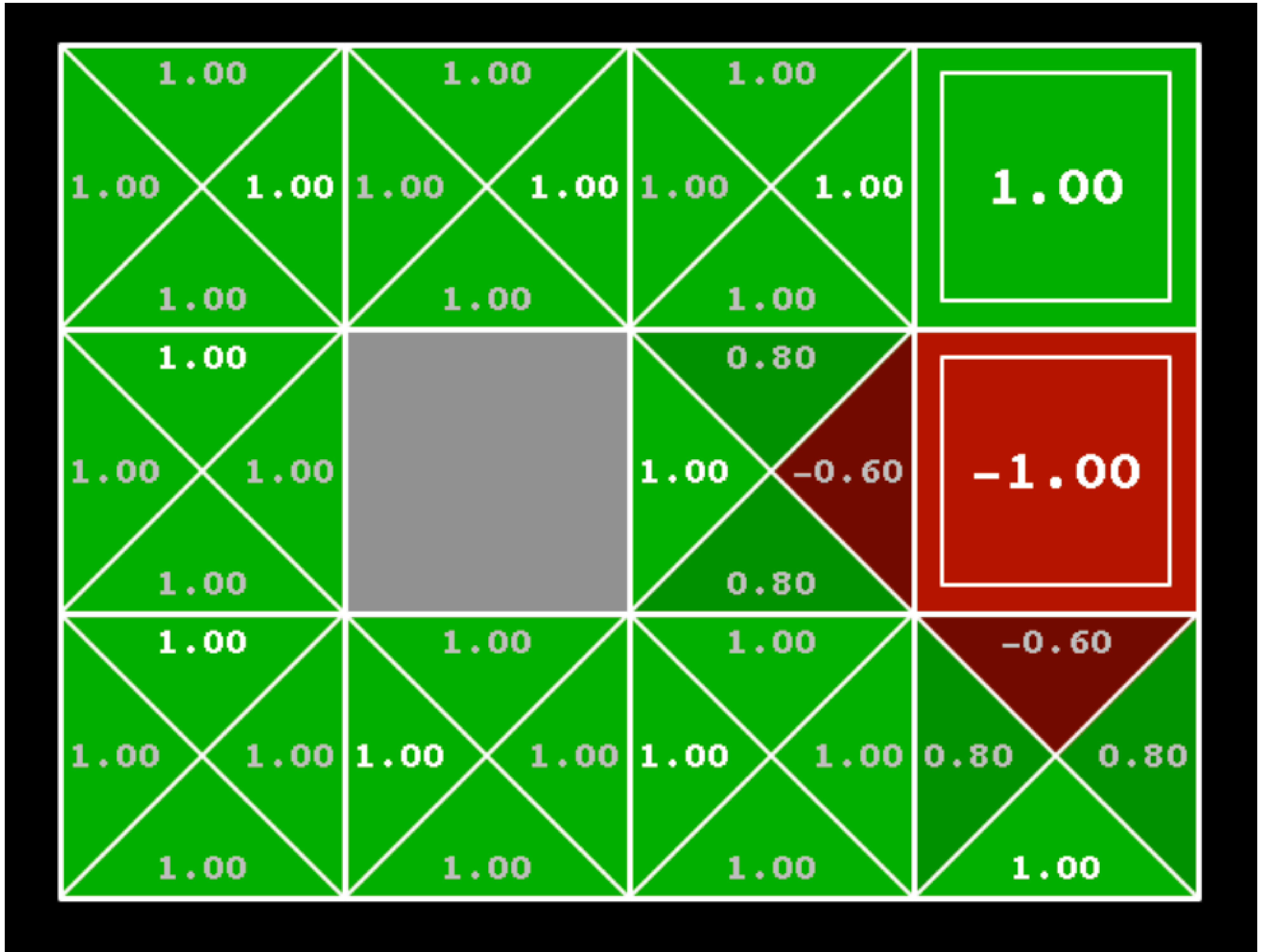
SARSA vs. Q-Learning

The key difference from SARSA is that Q-learning does not follow the current policy to pick the second action A_{t+1}. It estimates Q^* out of the best Q values, but which action (denoted as a^*) leads to this maximal Q does not matter and in the next step Q-learning may not follow a^*.



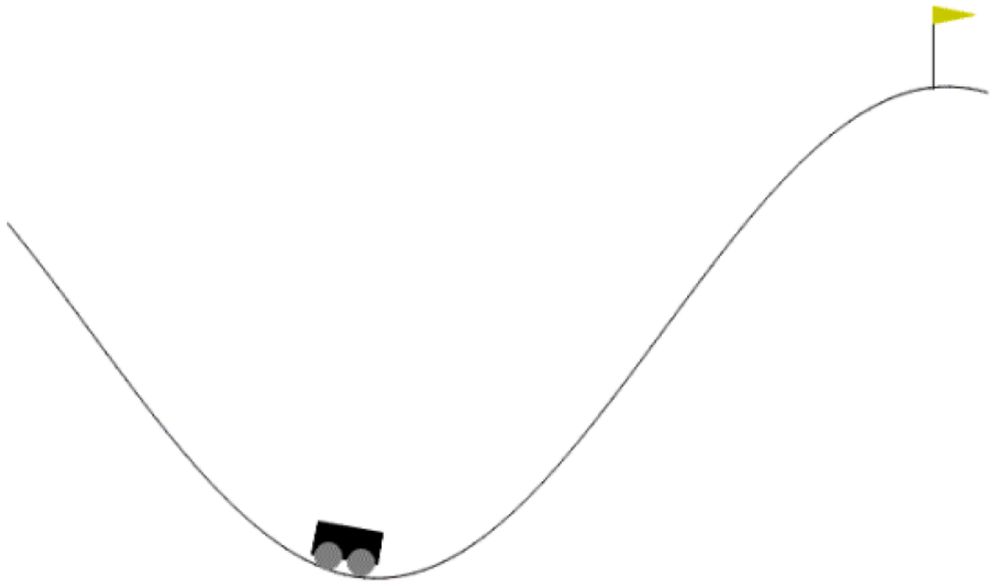
Q-Learning

The Basics



Q-VALUES AFTER 100 ITERATIONS

Mountain Car



- State (position, velocity)
- Actions (accelerate right, accelerate left, don't accelerate)

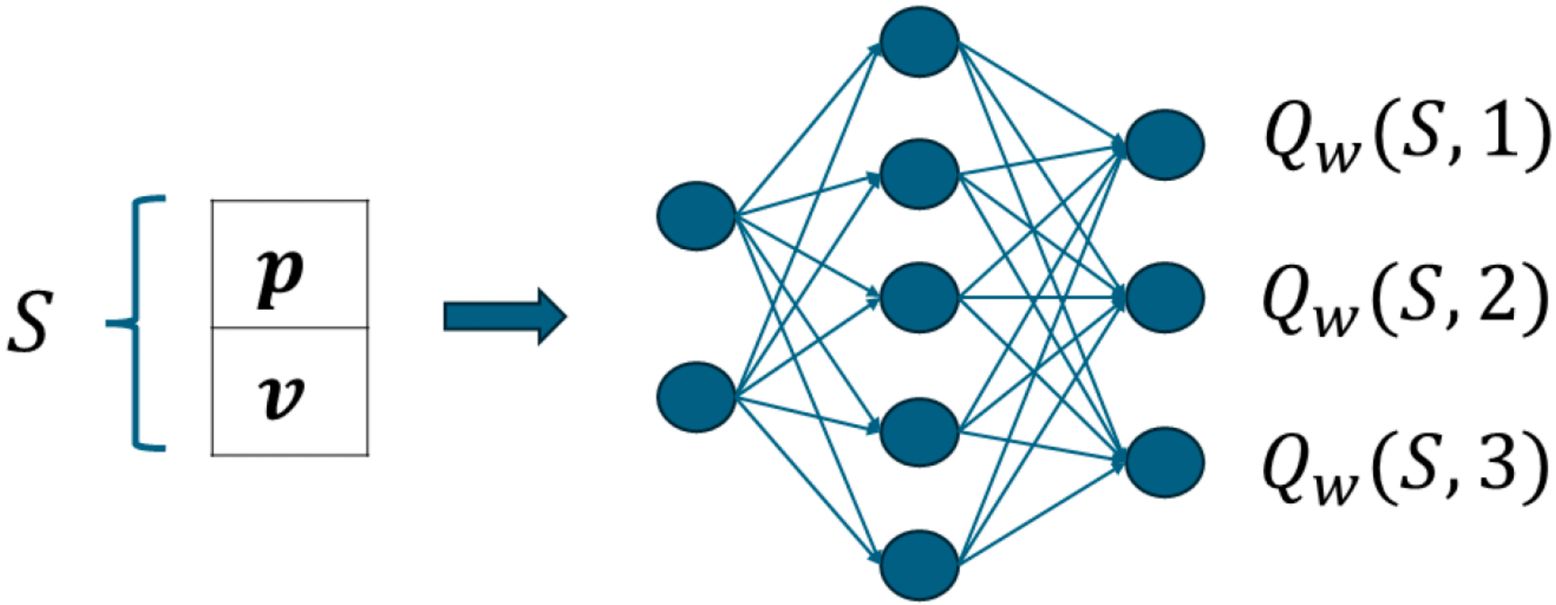
Deep Q-Network

Deep Q-Network

Theoretically, we can memorize $Q_*(.)$ for all state-action pairs in Q-learning, like in a gigantic table. However, it quickly becomes computationally infeasible when the state and action space are large. Thus people use functions (i.e. a machine learning model) to approximate Q values and this is called **function approximation**. For example, if we use a function with parameter θ to calculate Q values, we can label Q value function as $Q(s, a; \theta)$.

Approximation Methods

Approximate $Q_\pi(s,a)$ with $Q_w(s,a)$



Deep Q-Learning

Algorithm Q-Learning Off Policy

while True **do**
  Initialize s_0, $\alpha, t = 0$ and w_0
  for $t = 1, 2, \ldots$ **do**
    Obtain a_t using ϵ-greedy on Q_w
    Obtain a reward r_{t+1} and s_{t+1} from the env
    Obtain a_{t+1} using greedy
    $w_{t+1} \leftarrow w_t + \eta \left[r_{t+1} + \gamma Q_{w_t}(s_{t+1}, a_{t+1}) - Q_{w_t}(s_t, a_t) \right] \nabla Q_{w_t}(s_t, a_t)$
  end for
end while

The above is not stable!

Deep Q-Learning

Deep Q-Network ("DQN"; Mnih et al. 2015) aims to greatly improve and stabilize the training procedure of Q-learning by two innovative mechanisms:

- **Experience Replay**: All the episode steps $e_t = (S_t, A_t, R_t, S_{t+1})$ are stored in one replay memory $D_t = \{e_1, \ldots, e_t\}$. D_t has experience tuples over many episodes. During Q-learning updates, samples are drawn at random from the replay memory and thus one sample could be used multiple times. Experience replay improves data efficiency, removes correlations in the observation sequences, and smooths over changes in the data distribution.
- **Periodically Updated Target**: Q is optimized towards target values that are only periodically updated. The Q network is cloned and kept frozen as the optimization target every C steps (C is a hyperparameter). This modification makes the training more stable as it overcomes the short-term oscillations.

The loss function looks like this:

$$\mathcal{L}(\theta) = \mathbb{E}_{(s,a,r,s') \sim U(D)} \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta) \right)^2 \right]$$

where $U(D)$ is a uniform distribution over the replay memory D; θ^- is the parameters of the frozen target Q-network.

Deep Q-Learning

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$

For episode $= 1, M$ **do**

  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

  For $t = 1$,T **do**

    With probability ε select a random action a_t

    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$

    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 $\left(\phi_t, a_t, r_t, \phi_{t+1}\right)$ in D

    Sample random minibatch of transitions $\left(\phi_j, a_j, r_j, \phi_{j+1}\right)$ from D

    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}\left(\phi_{j+1}, a'; \theta^{-}\right) & \text{otherwise} \end{cases}$

    Perform a gradient descent step on $\left(y_j - Q\left(\phi_j, a_j; \theta\right)\right)^2$ with respect to the network parameters θ

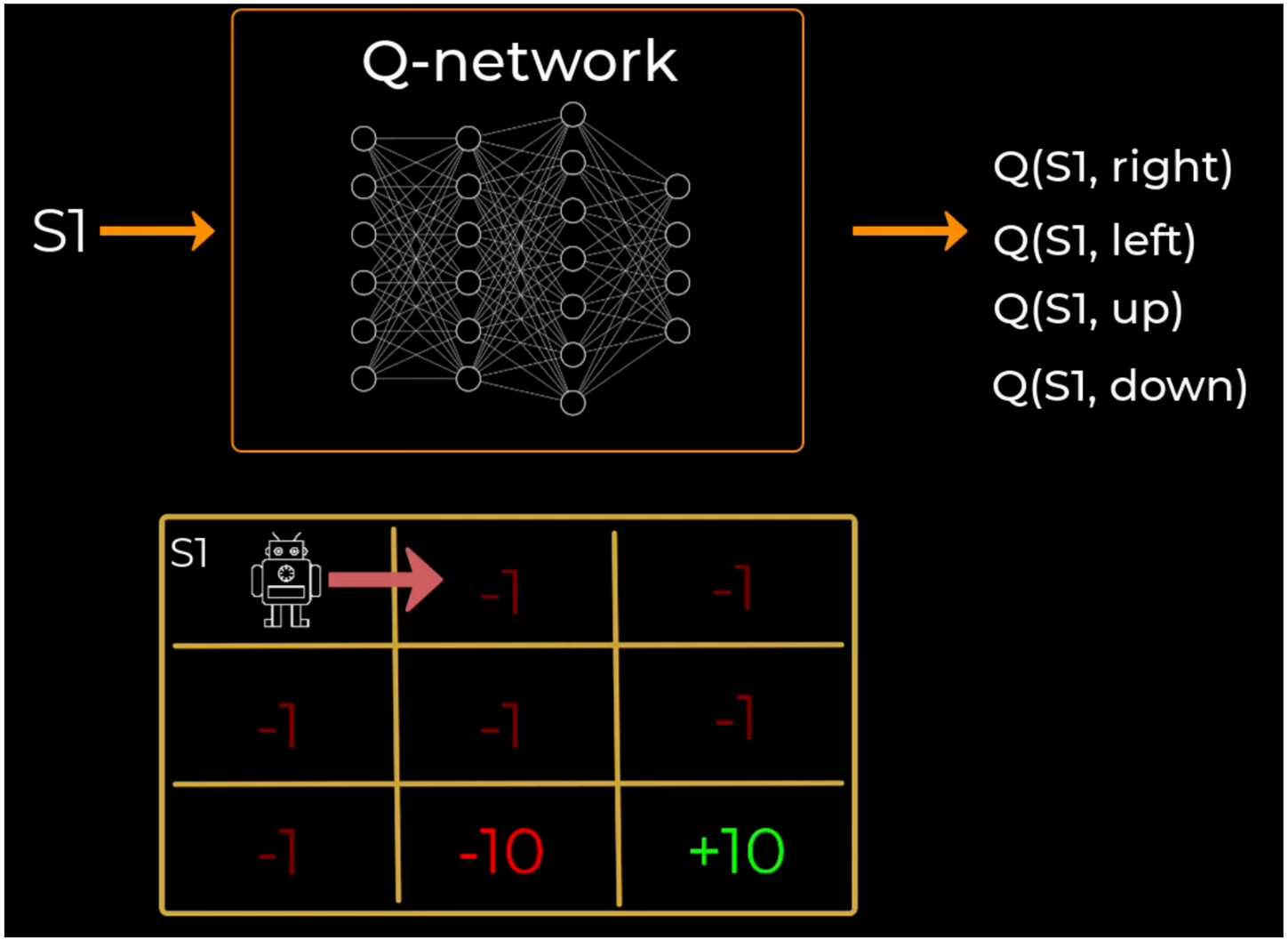
    Every C steps reset $\hat{Q} = Q$

  End For

End For

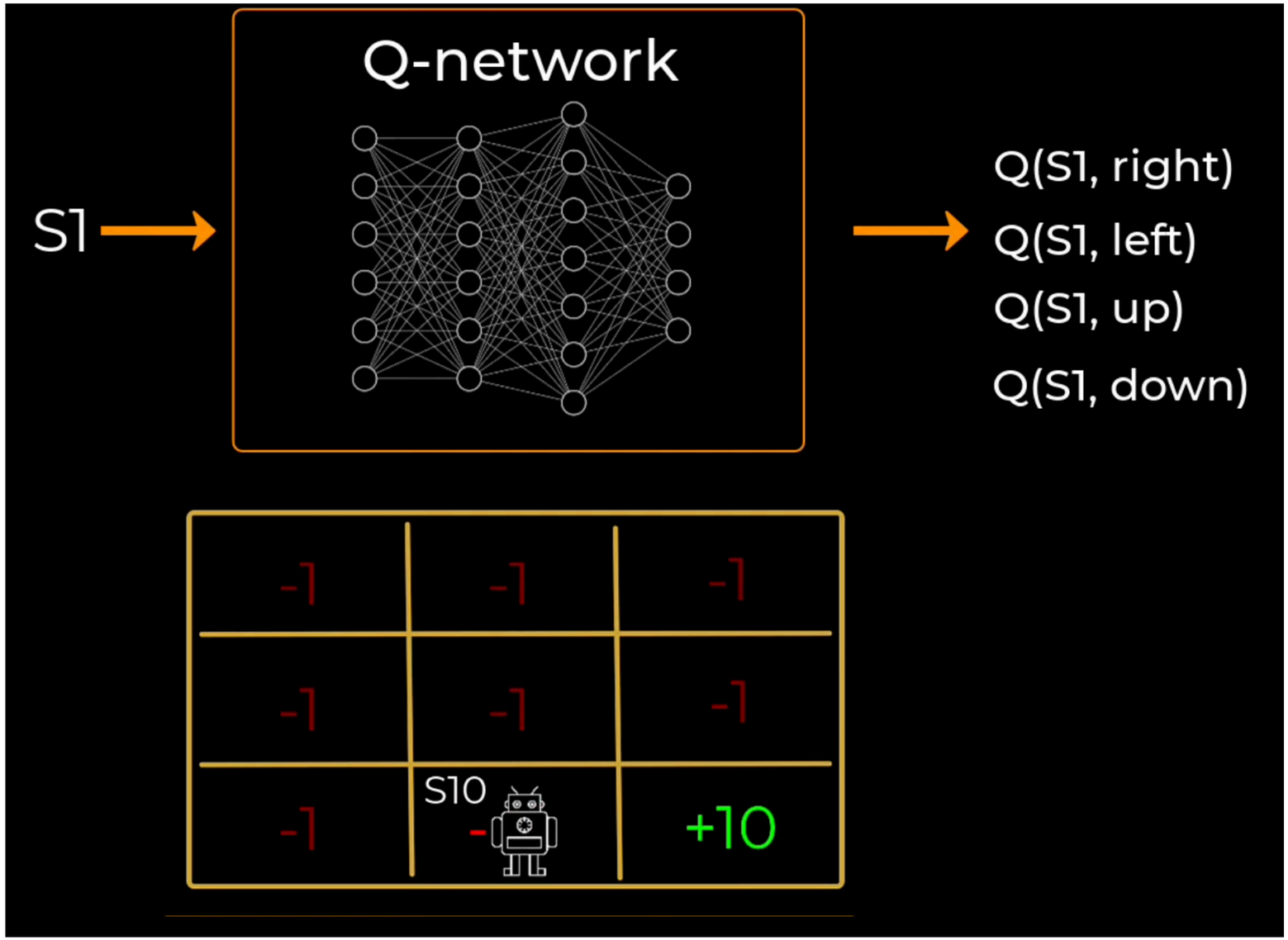
Source: Human-level control through deep reinforcement learning (Minh et al. 2015)

Deep Q-Learning



Source: [Deep Q-Networks Explained]

Deep Q-Learning

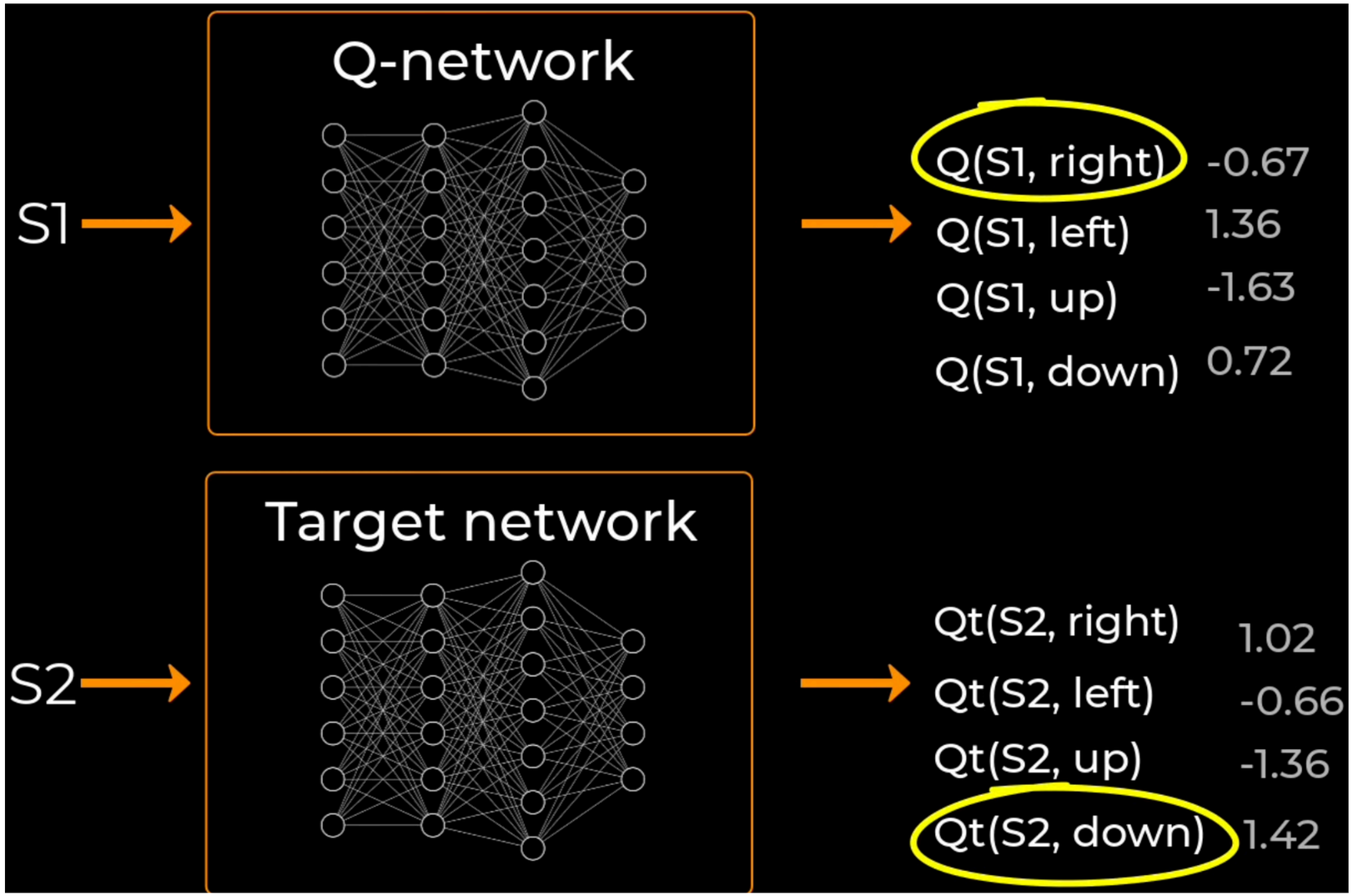


Experience Replay Buffer

(S1, right, -1, S2)

(S4, down, -10, S10)

Deep Q-Learning



Max Cut using DQN

The Setup

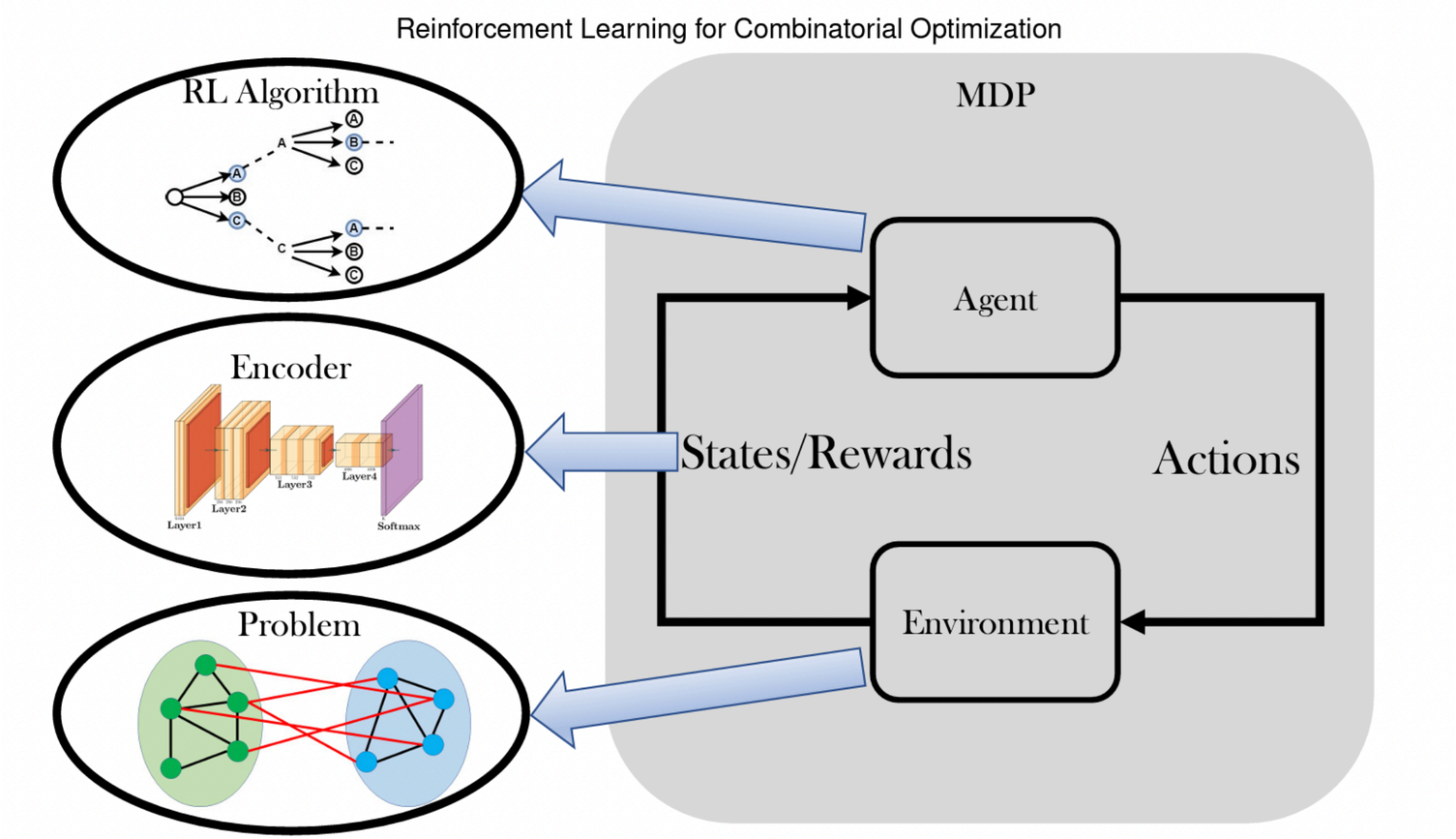
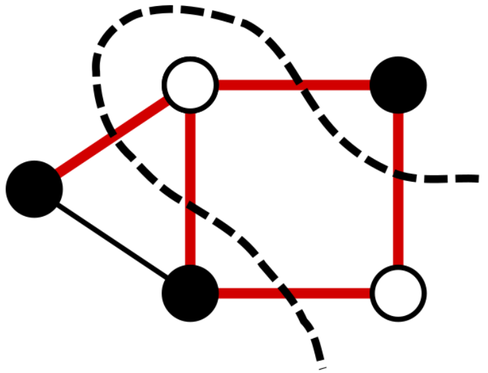


Figure 1: Solving a CO problem with the RL approach requires formulating MDP. The environment is defined by a particular instance of CO problem (e.g. Max-Cut problem). States are encoded with a neural network model (e.g. every node has a vector representation encoded by a graph neural network). The agent is driven by an RL algorithm (e.g. Monte-Carlo Tree Search) and makes decisions that move the environment to the next state (e.g. removing a vertex from a solution set).

Max Cut using DQN



Learning Combinatorial Optimization Algorithms over Graphs

Hanjun Dai[†*], **Elias B. Khalil**[†*], **Yuyu Zhang**[†], **Bistra Dilkina**[†], **Le Song**[†§]
[†] College of Computing, Georgia Institute of Technology
[§] Ant Financial
{hanjun.dai, elias.khalil, yuyu.zhang, bdilkina, lsong}@cc.gatech.edu

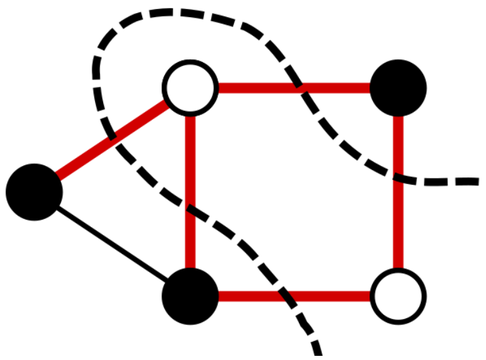
Abstract

The design of good heuristics or approximation algorithms for NP-hard combinatorial optimization problems often requires significant specialized knowledge and trial-and-error. Can we automate this challenging, tedious process, and learn the algorithms instead? In many real-world applications, it is typically the case that the same optimization problem is solved again and again on a regular basis, maintaining the same problem structure but differing in the data. This provides an opportunity for learning heuristic algorithms that exploit the structure of such recurring problems. In this paper, we propose a unique combination of reinforcement learning and graph embedding to address this challenge. The learned greedy policy behaves like a meta-algorithm that incrementally constructs a solution, and the action is determined by the output of a graph embedding network capturing the current state of the solution. We show that our framework can be applied to a diverse range of optimization problems over graphs, and learns effective algorithms for the Minimum Vertex Cover, Maximum Cut and Traveling Salesman problems.

Source: Khalil et al. (2017)

Max Cut using DQN

MDP Formulation

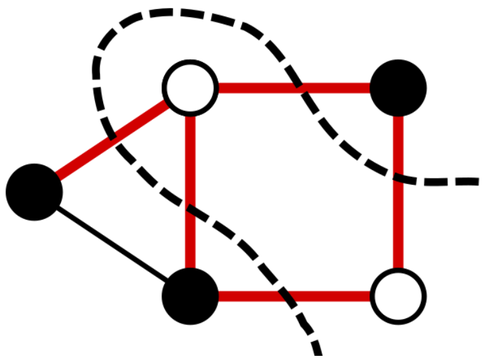


States (S)

- A state represents a **partial solution** to the Max-Cut problem.
- At any time t, the state s_t is a subset of vertices $S_t \subseteq V$ that have already been assigned to one of the two partitions.

Max Cut using DQN

MDP Formulation



States (S)

- A state represents a **partial solution** to the Max-Cut problem.
- At any time t, the state s_t is a subset of vertices $S_t \subseteq V$ that have already been assigned to one of the two partitions.

Actions (A)

- The action space consists of all **remaining vertices** that are not yet assigned to a partition.
- An action a_t selects a vertex $v \in V \setminus S_t$ and assigns it to one of the two subsets.

Max Cut using DQN

MDP Formulation

Transition Function ($T(s_t, a_t, s_{t+1})$)

- The transition function deterministically updates the state s_t based on the selected action a_t.
- Example: If a_t assigns vertex v to S_t, the new state s_{t+1} is $S_{t+1} = S_t \cup \{v\}$.

Max Cut using DQN

MDP Formulation

Transition Function ($T(s_t, a_t, s_{t+1})$)

- The transition function deterministically updates the state s_t based on the selected action a_t.
- Example: If a_t assigns vertex v to S_t, the new state s_{t+1} is $S_{t+1} = S_t \cup \{v\}$.

Reward (R)

- The reward is the **incremental change in the cut weight** after assigning a vertex.
- For an action a_t that assigns a vertex v to S_t:

$$R(s_t, a_t) = \Delta C(S_t, G) = \sum_{j \in V \setminus S_t} w_{vj},$$

where w_{vj} is the weight of edge (v, j).

Max Cut using DQN

1. A problem instance G of a given optimization problem is sampled from a distribution $\mathbb{D}$, i.e. the V, E and w of the instance graph G are generated according to a model or real-world data.
2. A partial solution is represented as an ordered list $S = (v_1, v_2, \ldots, v_{|S|})$, $v_i \in V$, and $\overline{S} = V \setminus S$ the set of candidate nodes for addition, conditional on S. Furthermore, we use a vector of binary decision variables x, with each dimension x_v corresponding to a node $v \in V$, $x_v = 1$ if $v \in S$ and 0 otherwise. One can also view x_v as a tag or extra feature on v.
3. A maintenance (or helper) procedure $h(S)$ will be needed, which maps an ordered list S to a combinatorial structure satisfying the specific constraints of a problem.
4. The quality of a partial solution S is given by an objective function $c(h(S), G)$ based on the combinatorial structure h of S.
5. A generic greedy algorithm selects a node v to add next such that v maximizes an evaluation function, $Q(h(S), v) \in \mathbb{R}$, which depends on the combinatorial structure $h(S)$ of the current partial solution. Then, the partial solution S will be extended as

$$S := (S, v^*), \quad \text{where} \quad v^* := \operatorname{argmax}_{v \in \overline{S}} \; Q(h(S), v), \tag{1}$$

and (S, v^*) denotes appending v^* to the end of a list S. This step is repeated until a termination criterion $t(h(S))$ is satisfied.

Source: Khalil et al. (2017)

Max Cut using DQN

1. A problem instance G of a given optimization problem is sampled from a distribution $\mathbb{D}$, i.e. the V, E and w of the instance graph G are generated according to a model or real-world data.
2. A partial solution is represented as an ordered list $S = (v_1, v_2, \ldots, v_{|S|})$, $v_i \in V$, and $\overline{S} = V \setminus S$ the set of candidate nodes for addition, conditional on S. Furthermore, we use a vector of binary decision variables x, with each dimension x_v corresponding to a node $v \in V$, $x_v = 1$ if $v \in S$ and 0 otherwise. One can also view x_v as a tag or extra feature on v.
3. A maintenance (or helper) procedure $h(S)$ will be needed, which maps an ordered list S to a combinatorial structure satisfying the specific constraints of a problem.
4. The quality of a partial solution S is given by an objective function $c(h(S), G)$ based on the combinatorial structure h of S.
5. A generic greedy algorithm selects a node v to add next such that v maximizes an evaluation function, $Q(h(S), v) \in \mathbb{R}$, which depends on the combinatorial structure $h(S)$ of the current partial solution. Then, the partial solution S will be extended as

$$S := (S, v^*), \quad \text{where} \quad v^* := \operatorname{argmax}_{v \in \overline{S}} \; Q(h(S), v), \tag{1}$$

and (S, v^*) denotes appending v^* to the end of a list S. This step is repeated until a termination criterion $t(h(S))$ is satisfied.

- **MAXCUT:** The helper function divides V into two sets, S and its complement $\overline{S} = V \setminus S$, and maintains a cut-set $C = \{(u, v) \,|\, (u, v) \in E, u \in S, v \in \overline{S}\}$. Then, the cost is $c(h(S), G) = \sum_{(u,v) \in C} w(u, v)$, and the termination criterion does nothing.

Source: Khalil et al. (2017)

Max Cut using DQN

Structure2Vector

Discriminative Embeddings of Latent Variable Models for Structured Data

Hanjun Dai, Bo Dai, Le Song

College of Computing, Georgia Institute of Technology
{hanjundai, bodai}@gatech.edu, lsong@cc.gatech.edu

Abstract

Kernel classifiers and regressors designed for structured data, such as sequences, trees and graphs, have significantly advanced a number of interdisciplinary areas such as computational biology and drug design. Typically, kernels are designed beforehand for a data type which either exploit statistics of the structures or make use of probabilistic generative models, and then a discriminative classifier is learned based on the kernels via convex optimization. However, such an elegant two-stage approach also limited kernel methods from scaling up to millions of data points, and exploiting discriminative information to learn feature representations.

We propose, `structure2vec`, an effective and scalable approach for structured data representation based on the idea of embedding latent variable models into feature spaces, and learning such feature spaces using discriminative information. Interestingly, `structure2vec` extracts features by performing a sequence of function mappings in a way similar to graphical model inference procedures, such as mean field and belief propagation. In applications involving millions of data points, we showed that `structure2vec` runs 2 times faster, produces models which are $10,000$ times smaller, while at the same time achieving the state-of-the-art predictive performance.

Source: Dai et al. (2016)

Max Cut using DQN

3.1 Structure2Vec

We first provide an introduction to `structure2vec`. This graph embedding network will compute a p-dimensional feature embedding μ_v for each node $v \in V$, given the current partial solution S. More specifically, `structure2vec` defines the network architecture recursively according to an input graph structure G, and the computation graph of `structure2vec` is inspired by graphical model inference algorithms, where node-specific tags or features x_v are aggregated recursively according to G's graph topology. After a few steps of recursion, the network will produce a new embedding for each node, taking into account both graph characteristics and long-range interactions between these node features. One variant of the `structure2vec` architecture will initialize the embedding $\mu_v^{(0)}$ at each node as 0, and for all $v \in V$ update the embeddings synchronously at each iteration as

$$\mu_v^{(t+1)} \leftarrow F\left(x_v, \{\mu_u^{(t)}\}_{u \in \mathcal{N}(v)}, \{w(v,u)\}_{u \in \mathcal{N}(v)}; \Theta\right), \tag{2}$$

where $\mathcal{N}(v)$ is the set of neighbors of node v in graph G, and F is a generic nonlinear mapping such as a neural network or kernel function.

Based on the update formula, one can see that the embedding update process is carried out based on the graph topology. A new round of embedding sweeping across the nodes will start only after the embedding update for all nodes from the previous round has finished. It is easy to see that the update also defines a process where the node features x_v are propagated to other nodes via the nonlinear propagation function F. Furthermore, the more update iterations one carries out, the farther away the node features will propagate and get aggregated nonlinearly at distant nodes. In the end, if one terminates after T iterations, each node embedding $\mu_v^{(T)}$ will contain information about its T-hop neighborhood as determined by graph topology, the involved node features and the propagation function F. An illustration of two iterations of graph embedding can be found in Figure 1.

Source: Khalil et al. (2017)

More Works on Max Cut using DQN

[Barrett et al., 2020] improved on the work of [Khalil et al., 2017] in terms of the approximation ratio as well as the generalization by proposing an ECO-DQN algorithm. The algorithm kept the general framework of S2V-DQN but introduced several modifications. The agent was allowed to remove vertices from the partially constructed solution to better explore the solution space. The reward function was modified to provide a normalized incremental reward for finding a solution better than seen in the episode so far, as well as give small rewards for finding a locally optimal solution that had not yet been seen during the episode. In addition, there were no penalties for decreasing cut value. The input of the state encoder was modified to account for changes in the reward structure. Since in this setting the agent had been able to explore indefinitely, the episode length was set to $2|V|$. Moreover, the authors allowed the algorithm to start from an arbitrary state, which could be useful by combining this approach with other methods, e.g. heuristics. This method showed better approximation ratios than S2V-DQN, as well as better generalization ability.

Source: Reinforcement Learning for Combinatorial Optimization: A Survey

RL Algorithms Overview

Algorithm	Year	Developers	Key Features	Notable Applications
Q-Learning	1989	Chris Watkins	Model-free, off-policy, learns Q-values for state-action pairs	Game playing, robotics
SARSA	1996	G. A. Rummery and M. Niranjan	Model-free, on-policy, learns Q-values for state-action pairs	Game playing, autonomous navigation
Deep Q-Network (DQN)	2013	Volodymyr Mnih et al. (DeepMind)	Combines Q-learning with deep neural networks, handles high-dimensional inputs	Atari games, robotics
Double DQN	2015	Hado van Hasselt et al. (DeepMind)	Reduces overestimation bias in Q-learning	Atari games, robotics
Dueling DQN	2016	Ziyu Wang et al. (DeepMind)	Separates state value and advantage functions	Atari games
Policy Gradient	1992	Ronald J. Williams	Directly optimizes policy, suitable for continuous action spaces	Robotics, finance
Actor-Critic	2000	W. T. Miller, R. S. Sutton, and P. J. Werbos	Combines policy gradient (actor) and value function (critic)	Robotics, game playing
A3C (Asynchronous Advantage Actor-Critic)	2016	Volodymyr Mnih et al. (DeepMind)	Parallel training of multiple agents, improves stability and efficiency	Game playing, robotics
PPO (Proximal Policy Optimization)	2017	John Schulman et al. (OpenAI)	Balances exploration and exploitation, stable and efficient training	Robotics, game playing, finance
SAC (Soft Actor-Critic)	2018	Tuomas Haarnoja et al. (UC Berkeley)	Maximizes entropy to encourage exploration, suitable for continuous action spaces	Robotics, autonomous driving
AlphaGo	2016	David Silver et al. (DeepMind)	Combines deep RL with Monte Carlo Tree Search, achieved superhuman performance	Go, chess
AlphaZero	2017	David Silver et al. (DeepMind)	Generalized version of AlphaGo, learns from self-play	Chess, shogi, Go
MuZero	2019	Julian Schrittwieser et al. (DeepMind)	Learns a model of the environment, combines model-based and model-free RL	Chess, shogi, Go, Atari games

Source: Reinforcement Learning: Sutton & Barto