COMP9444
Neural Networks and Deep Learning

7b. Reinforcement Learning

Outline

■ Reinforcement Learning vs. Supervised Learning

■ Models of Optimality

■ Exploration vs. Exploitation

■ Value Function Learning

▶ TD-Learning

▶ Q-Learning

■ TD-Gammon

Supervised Learning

Recall: Supervised Learning

■ We have a training set and a test set, each consisting of a set of examples. For each example, a number of input attributes and a target attribute are specified.

■ The aim is to predict the target attribute, based on the input attributes.

■ Various learning paradigms are available:

▶ Decision Trees

▶ Neural Networks

▶ SVM

▶ .. others ..

Learning of Actions

Supervised Learning can also be used to learn Actions, if we construct a training set of situation-action pairs (called Behavioral Cloning).

However, there are many applications for which it is difficult, inappropriate, or even impossible to provide a "training set"

- optimal control
  - ▶ mobile robots, pole balancing, flying a helicopter

- resource allocation
  - ▶ job shop scheduling, mobile phone channel allocation

- mix of allocation and control
  - ▶ elevator control, backgammon

Reinforcement Learning Framework

- An agent interacts with its environment.

- There is a set $\mathcal{S}$ of *states* and a set $\mathcal{A}$ of *actions*.

- At each time step t, the agent is in some state s_t.
  It must choose an action a_t, whereupon it goes into state
  $s_{t+1} = \delta(s_t, a_t)$ and receives reward $r_t = \mathcal{R}(s_t, a_t)$

- Agent has a *policy* $\pi : \mathcal{S} \to \mathcal{A}$. We aim to find an optimal policy π^*
  which maximizes the cumulative reward.

- In general, δ, $\mathcal{R}$ and π can be multi-valued, with a random element,
  in which case we write them as probability distributions

$$\delta(s_{t+1} = s \,|\, s_t, a_t) \quad \mathcal{R}(r_t = r \,|\, s_t, a_t) \quad \pi(a_t = a \,|\, s_t)$$

Probabilistic Policies
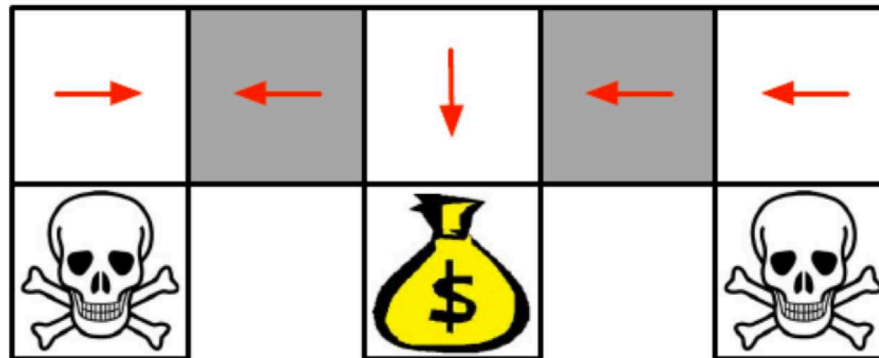
There are some environments in which any deterministic agent will perform very poorly, and the optimal (reactive) policy must be stochastic (i.e. randomized).

In 2-player games like Rock-Paper-Scissors, a random strategy is also required in order to make agent choices unpredictable to the opponent.

Models of optimality

Is a fast nickel worth a slow dime?

$$\text{Finite horizon reward} \quad \sum_{i=0}^{h-1} r_{t+i}$$

$$\text{Infinite discounted reward} \quad \sum_{i=0}^{\infty} \gamma^i r_{t+i}, \qquad 0 \le \gamma < 1$$

$$\text{Average reward} \quad \lim_{h \to \infty} \frac{1}{h} \sum_{i=0}^{h-1} r_{t+i}$$

- Finite horizon reward is simple computationally

- Infinite discounted reward is easier for proving theorems

- Average reward is hard to deal with, because can't sensibly choose between small reward soon and large reward very far in the future.

Comparing Models of Optimality
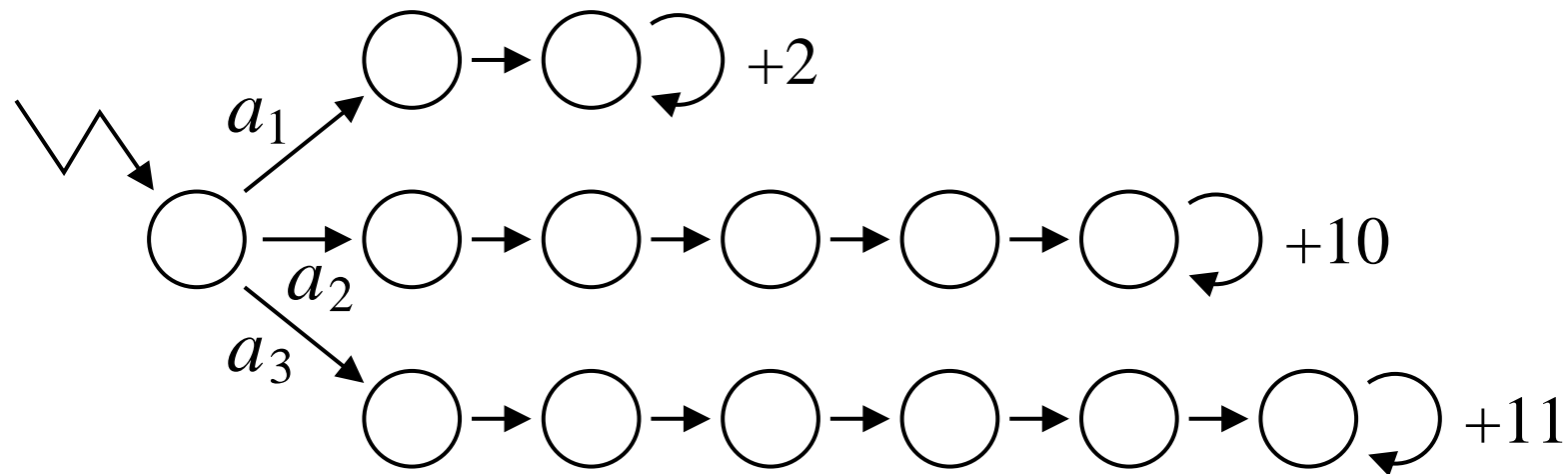
- Finite horizon, $k = 4$ $\rightarrow$ a_1 is preferred

- Infinite horizon, $\gamma = 0.9$ $\rightarrow$ a_2 is preferred

- Average reward $\rightarrow$ a_3 is preferred

RL Approaches

- Value Function Learning

  ▶ TD-Learning

  ▶ Q-Learning

- Policy Learning

  ▶ Evolution Strategies

  ▶ Policy Gradients

- Actor-Critic

  ▶ Combination of Value and Policy learning

Value Function Learning

Every policy π determines a Value Function $V^\pi : \mathcal{S} \to \mathrm{R}$ where $V^\pi(s)$ is the average discounted reward received by an agent who begins in state s and chooses its actions according to policy π.

If $\pi = \pi^*$ is optimal, then $V^*(s) = V^{\pi^*}(s)$ is the maximum (expected) discounted reward obtainable from state s. Learning this optimal value function can help to determine the optimal strategy.

The agent retains its own estimate $V()$ of the "true" value function $V^*()$. The aim of Value Function Learning is generally to start with a random V and then iteratively improve it so that it more closely approximates V^*. This process is sometimes called "Bootstrapping".

Value Function

(a)   (b)   (c)
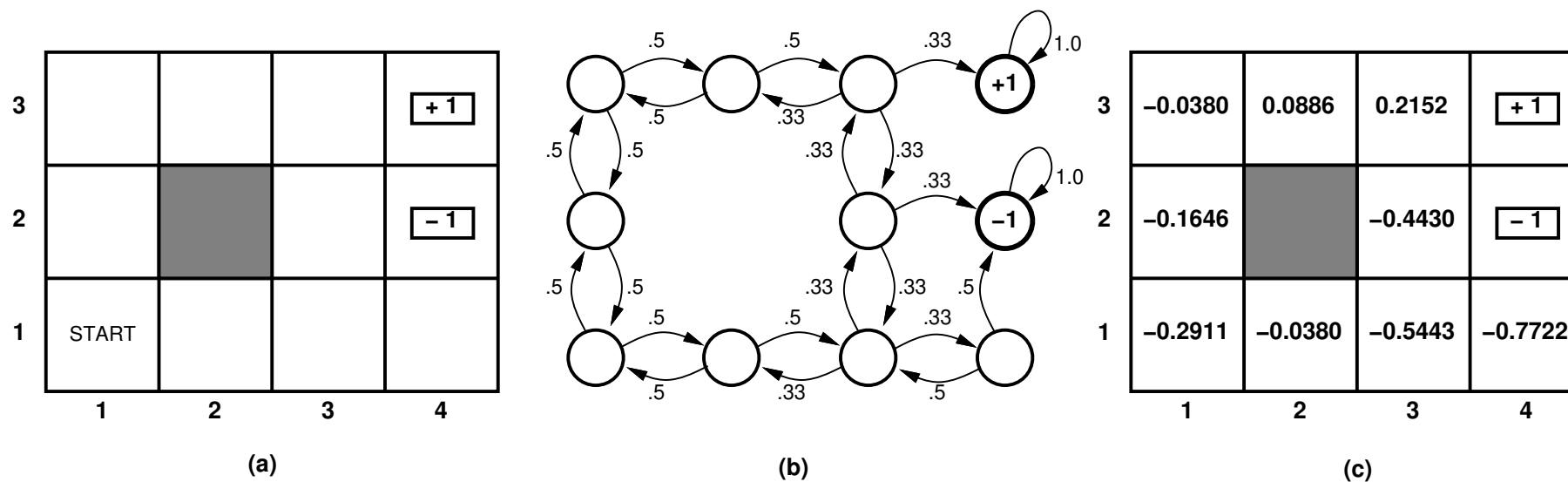
This is the Value Function V^π where π is the policy of choosing between available actions uniformly randomly.

K-Armed Bandit Problem

The special case of an active, stochastic environment with only one state is called the K-armed Bandit Problem, because it is like being in a room with several (friendly) slot machines, for a limited time, and trying to collect as much money as possible.

Each action (slot machine) provides a different average reward.

Exploration / Exploitation Tradeoff

Most of the time we should choose what we think is the best action.

However, in order to ensure convergence to the optimal strategy, we must occasionally choose something different from our preferred action, e.g.

- choose a random action 5% of the time, or

- use Softmax (Boltzmann distribution) to choose the next action:

$$P(a) = \frac{e^{\mathcal{R}(a))/T}}{\sum_{b \in \mathcal{A}} e^{\mathcal{R}(b))/T}}$$

Exploration / Exploitation Tradeoff
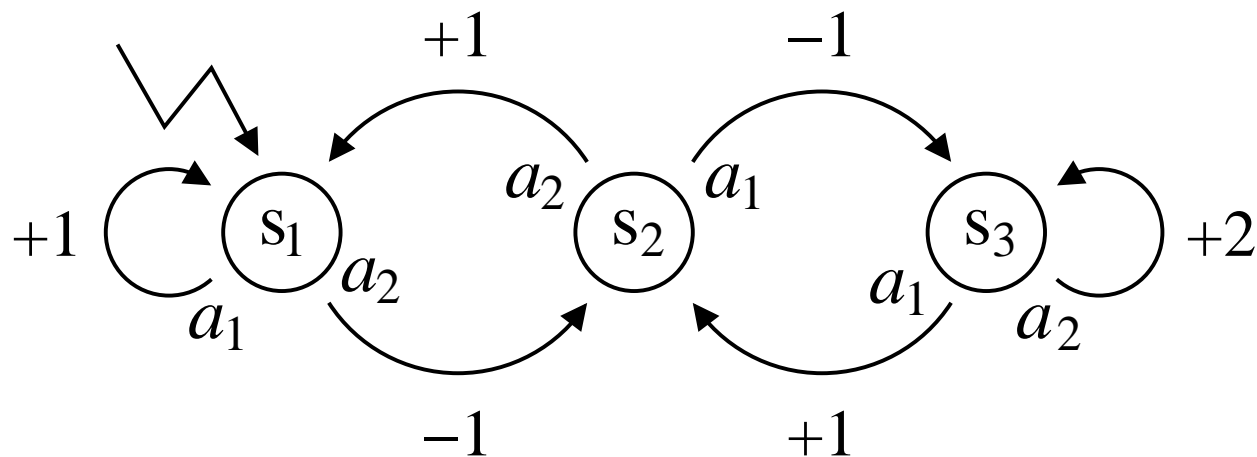
I was born to try...

But you've got to make choices

Be wrong or right

Sometimes you've got to sacrifice the things you like.

- Delta Goodrem

Delayed Reinforcement

We may need to take several actions before we can get the good reward.

Temporal Difference Learning

Let's first assume that $\mathcal{R}$ and δ are deterministic. Then the (true) value $V^*(s)$ of the current state s should be equal to the immediate reward plus the discounted value of the next state

$$V^*(s) = \mathcal{R}(s,a) + \gamma V^*(\delta(s,a))$$

We can turn this into an update rule for the estimated value, i.e.

$$V(s_t) \leftarrow r_t + \gamma V(s_{t+1})$$

If $\mathcal{R}$ and δ are stochastic (multi-valued), it is not safe to simply replace $V(s)$ with the expression on the right hand side. Instead, we move its value fractionally in this direction, proportional to a learning rate η

$$V(s_t) \leftarrow V(s_t) + \eta \left[r_t + \gamma V(s_{t+1}) - V(s_t) \right]$$

Q-Learning

Q-Learning is similar to TD-Learning except that we use a function $Q^\pi : \mathcal{S} \times \mathcal{A} \to \mathrm{R}$ which depends on a state, action pair instead of just a state.

For any policy π the Q-Function $Q^\pi(s,a)$ is the average discounted reward received by an agent who begins in state s, first performs action a and then follows policy π for all subsequent timesteps.

If $\pi = \pi^*$ is optimal, then $Q^*(s,a) = Q^{\pi^*}(s,a)$ is the maximum (expected) discounted reward obtainable from s, if the agent is forced to take action a in the first timestep but can act optimally thereafter.

The agent retains its own (initially, random) estimate $Q()$ and iteratively improves this estimate to more closely approximate the "true" function $Q^*()$.

Q-Learning

For a deterministic environment, π^*, Q^* and V^* are related by

$$\pi^*(s) = \mathrm{argmax}_a\, Q^*(s, a)$$

$$Q^*(s, a) = \mathcal{R}(s, a) + \gamma V^*(\delta(s, a))$$

$$V^*(s) = \max_b Q^*(s, b)$$

So

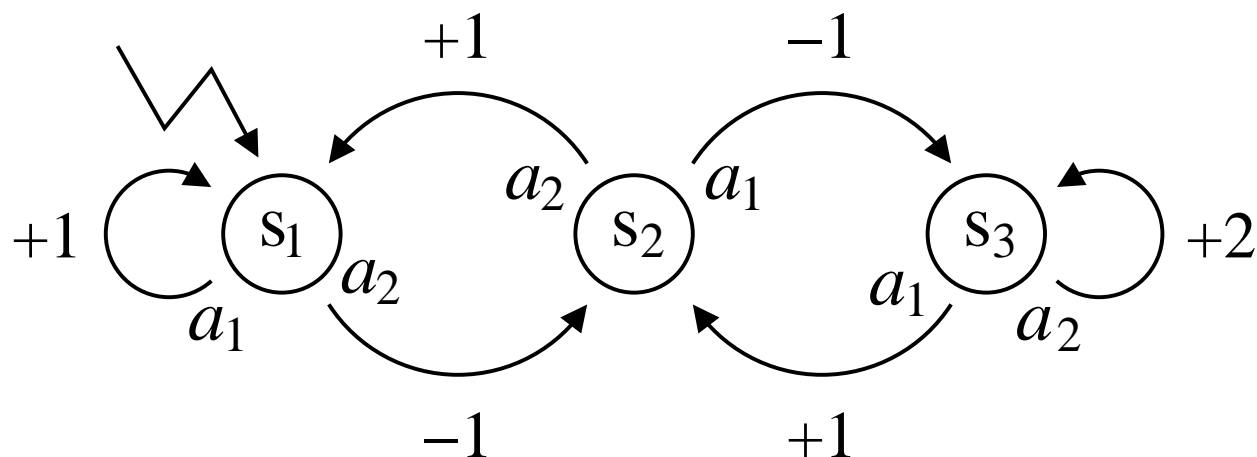$$Q^*(s, a) = \mathcal{R}(s, a) + \gamma \max_b Q^*(\delta(s, a), b)$$

This allows us to iteratively approximate Q by

$$Q(s_t, a_t) \leftarrow r_t + \gamma \max_b Q(s_{t+1}, b)$$

If the environment is stochastic, we instead write

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \eta\left[r_t + \gamma \max_b Q(s_{t+1}, b) - Q(s_t, a_t)\right]$$

Q-Learning Example

Exercise:

1. compute $V^\pi(s_3)$ if $\pi(s_3) = a_2$ and $\gamma = 0.9$

2. compute π^*, V^* and Q^* for this environment (if $\gamma = 0.9$)

Theoretical Results

Theorem: Q-learning will eventually converge to the optimal policy, for any deterministic Markov decision process, assuming an appropriately randomized strategy.

(Watkins & Dayan 1992)

Theorem: TD-learning will also converge, with probability 1.

(Sutton 1988, Dayan 1992, Dayan & Sejnowski 1994)

Limitations of Theoretical Results

■ Delayed reinforcement

▶ reward resulting from an action may not be received until several time steps later, which also slows down the learning

■ Search space must be finite

▶ convergence is slow if the search space is large

▶ relies on visiting every state infinitely often

■ For "real world" problems, we can't rely on a lookup table

▶ need to have some kind of generalisation (e.g. TD-Gammon)

Computer Game Playing

Suppose we want a write a computer program to play a game like Backgammon, Chess, Checkers or Go. This can be done using a tree search algorithm (expectimax, MCTS, or minimax with alpha-beta pruning). But we need:

(a) an appropriate way of encoding any board position as a set of numbers, and

(b) a way to train a neural network or other learning system to compute a board evaluation, based on those numbers

Backgammon
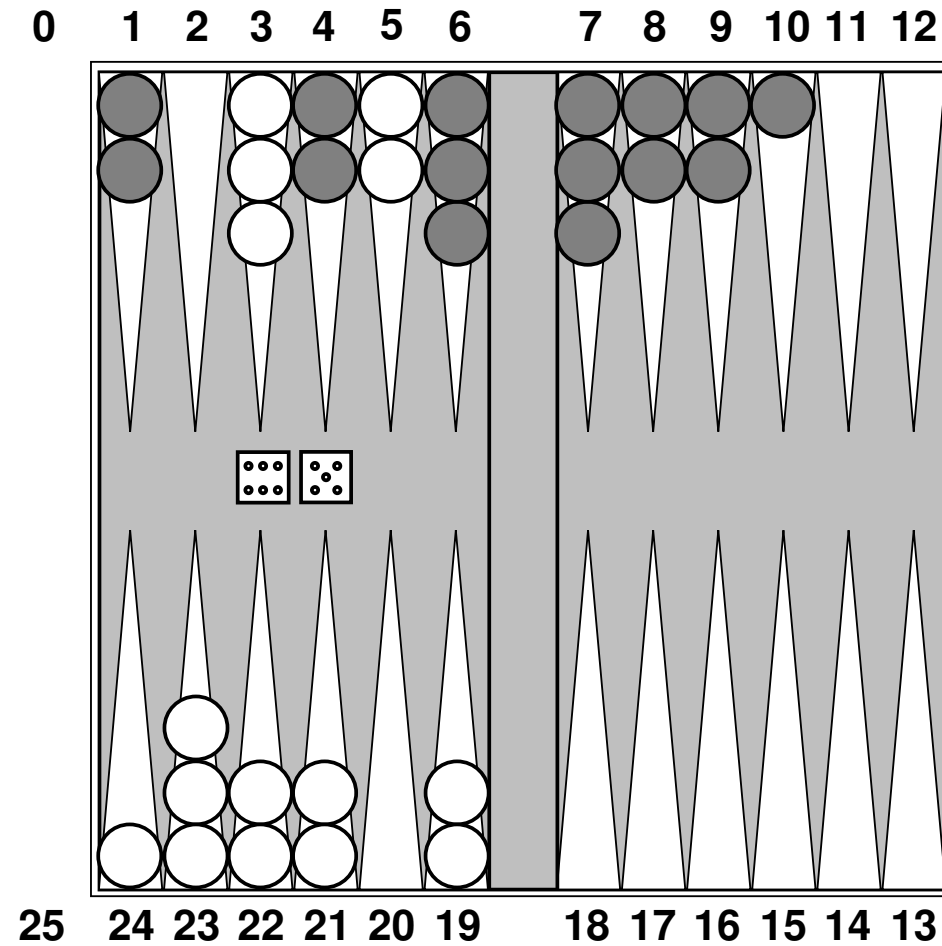
Backgammon Neural Network

Board encoding

- 4 units $\times$ 2 players $\times$ 24 points
- 2 units for the bar
- 2 units for off the board

Two layer neural network

- 196 input units
- 20 hidden units
- 1 output unit

The input s is the encoded board position (state),
the output V(s) is the value of this position (probability of winning).

At each move, roll the dice, find all possible "next board positions", convert them to the appropriate input format, feed them to the network, and choose the one which produces the largest output.

Backpropagation

$$w \leftarrow w + \eta(T - V)\frac{\partial V}{\partial w}$$

$$
\begin{aligned}
V &= \text{actual output} \\
T &= \text{target value} \\
w &= \text{weight} \\
\eta &= \text{learning rate}
\end{aligned}
$$

Q: How do we choose the target value T ?

In other words, how do we know what the value of the current position "should have been"? or, how do we find a better estimate for the value of the current position?

How to Choose the Target Value

■ Behavioral Cloning (Supervised Learning)

   ▶ learn moves from human games (Expert Preferences)

■ Temporal Difference Learning

   ▶ use subsequent positions to refine evaluation of current position

   ▶ general method, does not rely on knowing the "world model" (rules of the game)

■ methods which combine learning with tree search (must know the "world model")

   ▶ TD-Root, TD-Leaf, MCTS, TreeStrap

TD-Learning for Episodic Games

Backgammon is an example of an episodic task, in the sense that the agent receives just a single reward at the end of the game, which we can consider as the final value V_{m+1} (typically, $+1$ for a win or -1 for a loss). We then have a sequence of game positions, each with its own (estimated) value:

(current estimate) $\quad V_t \to V_{t+1} \to \ldots \to V_m \to V_{m+1} \quad$ (final result)

In this context, TD-Learning simplifies and becomes equivalent to using the value of the next state (V_{t+1}) as the training value for the current state (V_k)

A fancier version, called TD(λ), uses T_k as the training value for V_k, where

$$T_t = (1-\lambda) \sum_{k=t+1}^{m} \lambda^{k-1-t} V_k + \lambda^{m-t} V_{m+1}$$

T_t is a weighted average of future estimates, $\lambda = $ discount factor ($0 \leq \lambda < 1$)

TD-Gammon

■ Tesauro trained two networks:

▶ EP-network was trained on Expert Preferences (Supervised)

▶ TD-network was trained by self play (TD-Learning)

■ TD-network outperformed the EP-network.

■ With modifications such as 3-step lookahead (expectimax) and additional hand-crafted input features, TD-Gammon became the best Backgammon player in the world (Tesauro, 1995).