CS 4700:
Foundations of Artificial Intelligence

Bart Selman

Reinforcement Learning

R&N – Chapter 21

Note: in the next two parts of RL, some of the figure/section numbers refer to an earlier edition of R&N with a more basic description of the techniques.

The slides provide a self-contained description.

Reinforcement Learning

In our discussion of Search methods (developed for problem solving), we assumed a given State Space and operators that lead from one State to one or more Successor states with a possible operator Cost.

The State space can be exponentially large but is in principle Known. The difficulty was finding the right path (sequence of moves). This problem solved by searching through the various alternative sequences of moves. In tough spaces, this leads to exponential searches.

Can we do something totally different?? Avoid search…

Why don't we "just learn" how to make the right move in each possible state?

In principle, need to know very little about environment at the start. Simply observe another agent / human / program make steps (go from state to state) and mimic!

Reinforcement learning: Some of the earliest AI research (1960s). It works! Principles and ideas still applicable today.

Environment we consider is a basic game (the simplest non-trivial game):

Tic-Tac-Toe

The question: **Can you write a program that learns to play Tic-Tac-Toe?**

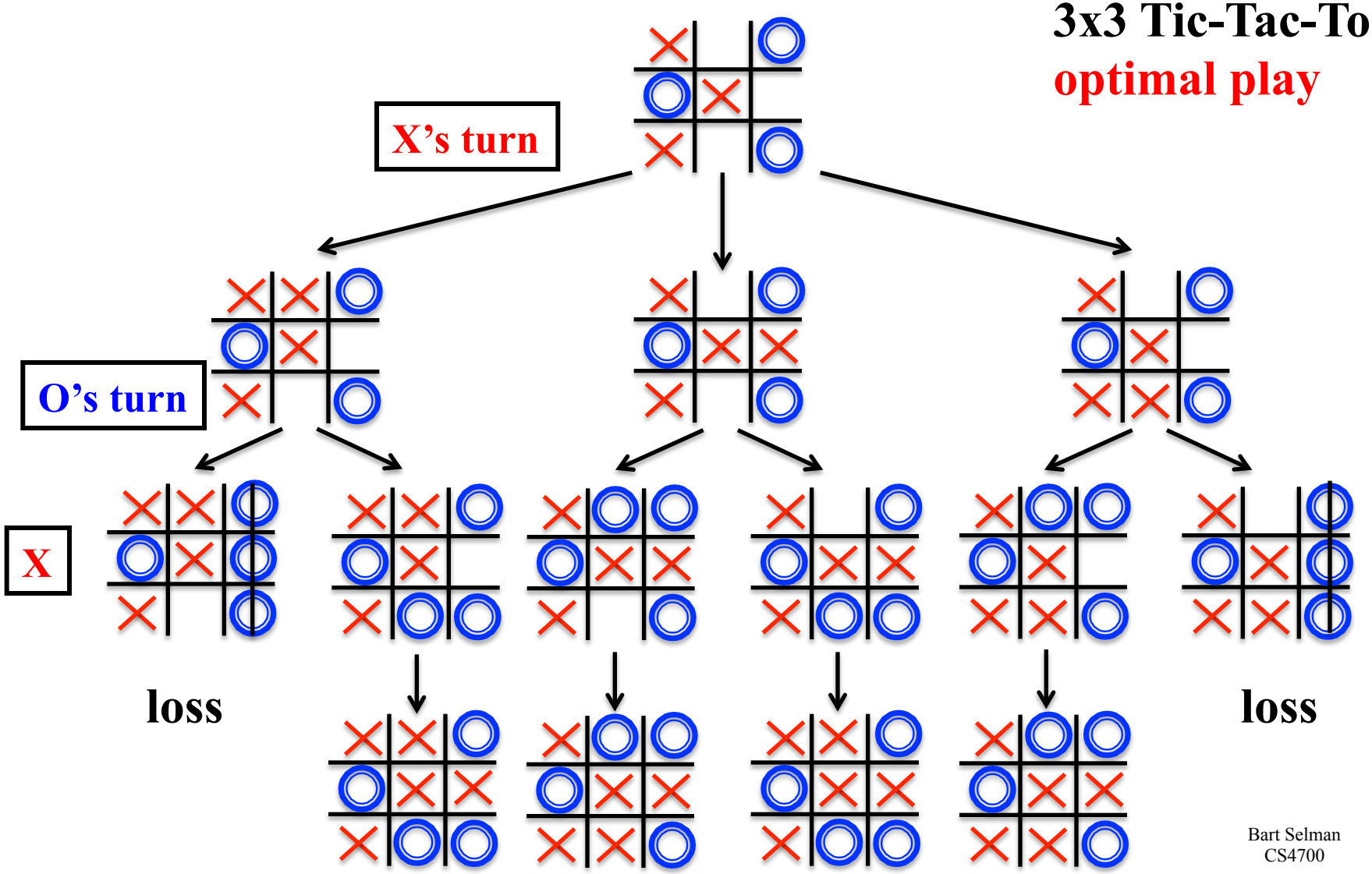
Let's try to re-discover what Donald Michie did in 1962. He did not even use a computer! He hand-simulated one.

The first non-trivial machine learning program!

Tic-tac-toe (or Noughts and crosses, Xs and Os)

Now, we don't want…

We start 3 moves per player in:



What else can we think of?

Basic ingredients needed:

1) We need to represent board states.

2) What moves to make in different states.

It may help to think a bit probabilistically … pick moves with some probability and adjust probabilities through a learning procedure …

Learn from human opponent

We could try to learn directly from human what moves to make…

But, some issues:

1) **Human may be a weak player. ☺ We want to learn how to beat him/her!**
2) **Human may play "nought" (second player) and computer wants to learn how to play "cross" (first player).**

Answer:

Let's try to "just play" human against machine and learn something from wins and losses.

To start: some basics of the “machine”

For each board state where cross is on-move, have a “match box” labeled with that state.

Requires a few hundred matchboxes.

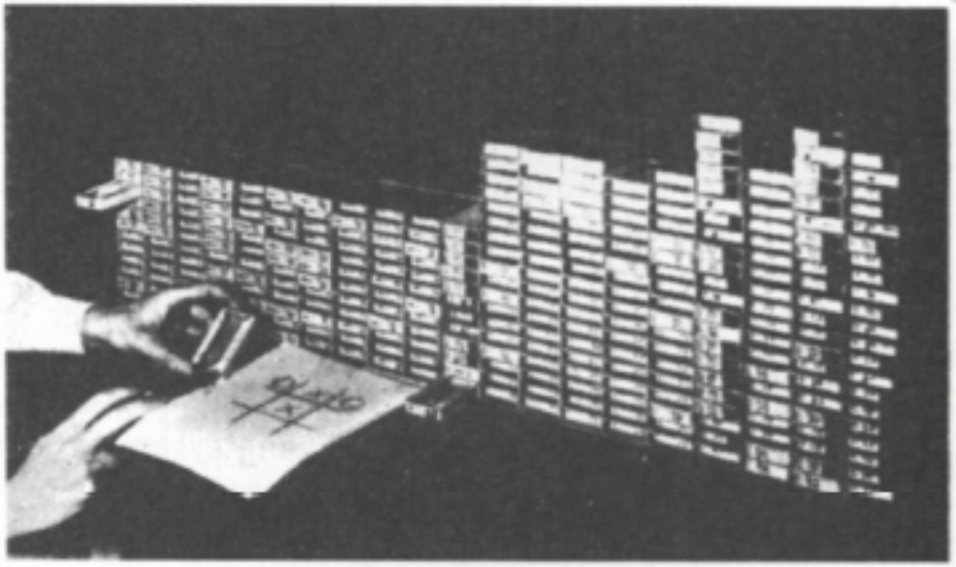


Fig. 2.—The matchbox machine—MENACE

Each match box has a number of colored "beads" in it, each color represents a valid move for cross on that board.

E.g. start with ten beads of each color for each valid move.

1) To make a move, pick up box with label of current state, shake it, Pick random bead. Check color and make that move.

2) New state, wait for human counter-move. New state, repeat above.

Table 1

The colour code used in the matchbox machine. The system of numbering the squares is that adopted for the subsequent computer simulation program

1 WHITE	2 LILAC	3 SILVER
8 BLACK	0 GOLD	4 GREEN
7 AMBER	6 RED	5 PINK

Game ends when one of the parties has a win / loss or no more open spaces.

This is how the machine plays. How well will it play? What is is doing initially?

Machine needs to learn! How? Can you think of a strategy? The first successful machine learning program in history (not involving search)…

Let's try to come up with a strategy…What do we need to do?

Reinforcement Learning

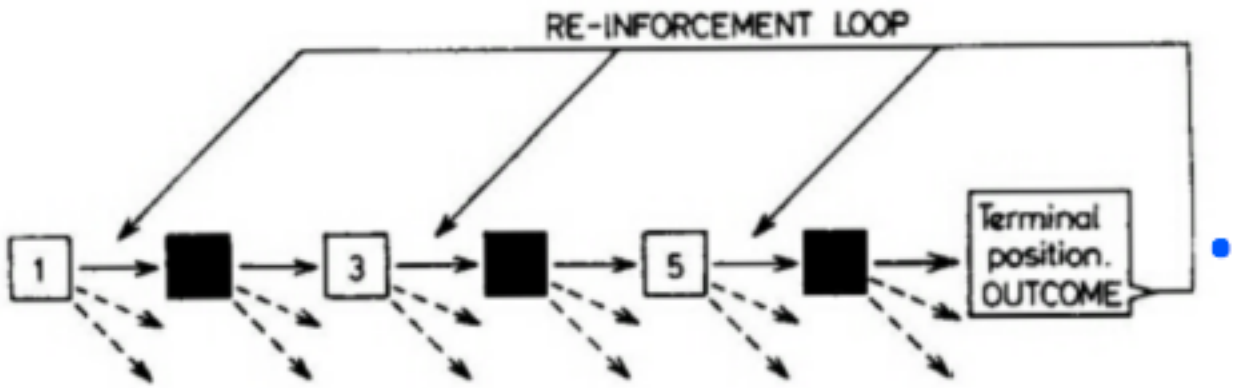


Fig. 1.—Schematic picture of the reinforcement process during trial-and-error learning of a game. The numbered boxes represent the players' successive choice-points, and the black boxes those of the opponent. Arrows drawn with broken lines indicate possible alternative choices open at the given stage

The Computer Journal-1963-Michie-232-6-1.pdf

Experiments on the mechanization of game-learning
Part I. Characterization of the model and its parameters

By Donald Michie

This paper describes a trial-and-error device which learns to play the game of Noughts and Crosses. It was initially constructed from matchboxes and coloured beads and subsequently simulated in essentials by a program for a Pegasus 2 computer. The parameters governing the adaptive behaviour of this automaton are described and preliminary observations on its performance are briefly reported.

Reinforcement Learning

and so on to the end of the play.

At this stage reinforcements are applied. If the machine has done badly, it is "punished" by confiscation of the selected bead from each of the three or four boxes which have been used during the play, so that it becomes less probable, when any of these positions recur in future play, that the unsuccessful move will be repeated. If the machine has done well, it is "rewarded" by adding to each of the open boxes an extra bead of the same colour as the selected one. The moves in the successful sequence thus become more likely to be repeated if and when any of these positions recur in future.

Works!!! ☺ Don't need that many games. Quite surprising!

Comments

Learning in this case took "advantage of":

1) **State space is manageable. Further reduced by using 1 state to represent all isomorphic states (through board rotations and symmetries).**

MENACE is a machine that plays noughts and crosses, built out of 304 matchboxes. Each matchbox corresponds to one of the 304 board layouts that the opening player might face (there are actually 19,683 possible board layouts, but we only need to calculate the opening player's first four moves, and many are rotationally or reflectively identical). In turn, each matchbox

We quietly encoded some knowledge about tic-tac-toe!

2) What if state space is MUCH larger? As for any interesting game…

Options:

a) **Represent board by "features." I.e., number of various pieces on chess board but not their position.. It's like having each matchbox represent a large collection of states. Notion of "valid moves" becomes a bit trickier.**

b) **Don't store "match boxes" / states explicitly, instead learn a function (e.g. neural net) that computes the right move directly when given some representation of the state as input.**

c) **Combination of a) and b).**

d) **Combine a), b), and c) with some form of "look-ahead" search.**