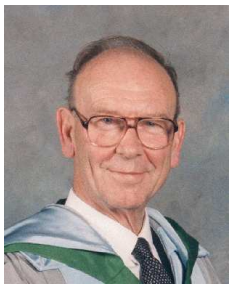
COMP3411/9414/9814: Artificial Intelligence

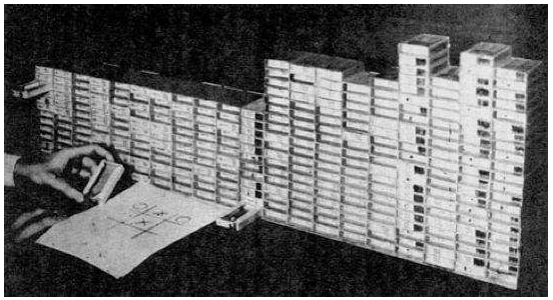
Week 11: Learning Games

Timeline

- 1959 Checkers (Arthur Samuel)
- 1961 MENACE tic-tac-toe (Donald Michie)
- 1989 TD-Gammon (Gerald Tesauro)
- 1997 TD-leaf (Baxter et al.)
- 2006 MoGo using MCTS (Gelly & Wang)
- 2009 TreeStrap (Veness et al.)
- 2016 AlphaGo
- 2018 Alpha Zero

MENACE

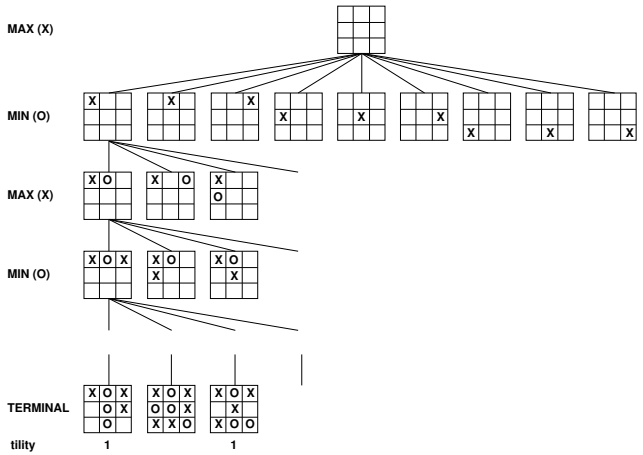




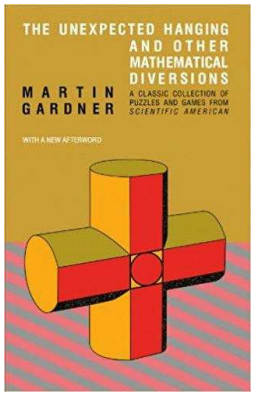
Machine Educable Noughts And Crosses Engine
Donald Michie, 1961

MENACE

Game Tree (2-player, deterministic)



Martin Gardner and HALO



Hexapawn Boxes

Reinforcement Learning with BOXES

This BOXES algorithm was later adapted to learn more general tasks such as Pole Balancing, and helped lay the foundation for the modern field of Reinforcement Learning.

- BOXES: An Experiment in Adaptive Control, D.Michie and R.Chambers, Machine Intelligence, Oliver and Boyd, Edinburgh, UK, (1968).

Historical Perspective

Artificial Intelligence in general, and Machine Learning in particular, came under heavy criticism in the early 1970's. Donald Michie's Reinforcement Learning research was deliberately excluded from the 1973 Lighthill report because Lighthill wanted to focus attention on other areas which could most easily be criticised. The field became largely dormant, until it was revived in the late 1980's, largely through the work of Richard Sutton. Gerald Tesauro applied Sutton's TD-Learning algorithm to the game of Backgammon in 1989.

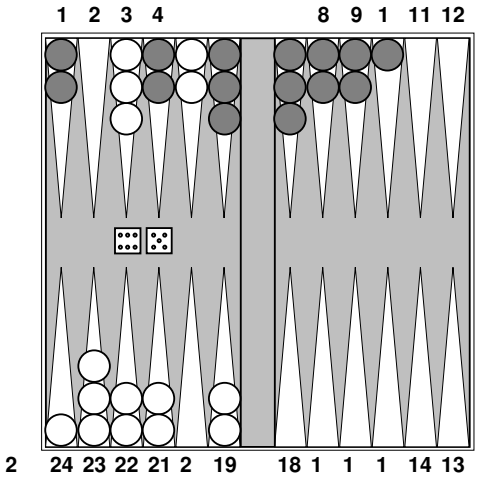
Computer Game Playing

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 × 2 players × 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

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

V = actual output
T = target value
w = weight
η = learning rate

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 (Samuel, 1959)
  - TD-Leaf (Baxter et al., 1998)
  - TreeStrap (Veness et al., 2009)

Temporal Difference Learning

We have a sequences of positions in the game, each with its own (estimated) value:

(current estimate) $V_k \rightarrow V_{k+1} \rightarrow \ldots \rightarrow V_m \rightarrow V_{m+1}$ (final result)

TD(0): Use the value of the next state (V_{k+1}) as the training value for the current state (V_k).

TD(λ): use T_k as the training value for V_k, where

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

T_k is a weighted average of future estimates,
λ = 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).

Why Did TD-Gammon Work?

- Random dice rolls in Backgammon force self-play to explore a much larger part of the search space than in a deterministic game.
- Humans are good at reasoning about a small set of probabilistic outcomes. But, playing Backgammon well requires aggregating a large set of possibilities, each with a small likelihood, and balancing them against each other. Neural Networks might be better at this than humans.
- For deterministic games like Chess, direct TD-Learning performs poorly. Methods which combine learning with tree search are more effective.

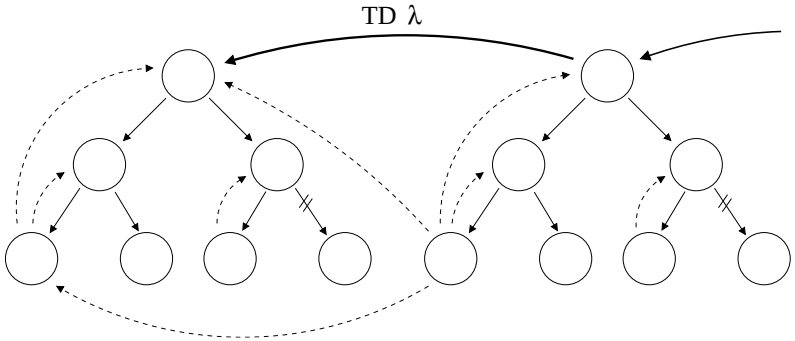
Chess

Move selection is by alpha-beta search, using some function V(s) to evaluate the leaves.

Heuristic Evaluation for Chess

- Material weights
  - For example, Queen = 9, Rook = 5, Knight = Bishop = 3, Pawn = 1
- Piece-Square weights
  - some (fractional) score for a particular piece on a particular square
- Attack/Defend weights
  - some (fractional) score for one piece attacking or defending another piece.
- Other features, such as Pawn structure, Mobility, etc.
- There are no hidden nodes. V(s) is a linear combination of input features, composed with a sigmoid function, to produce a value between 0 and 1 (probability of winning).

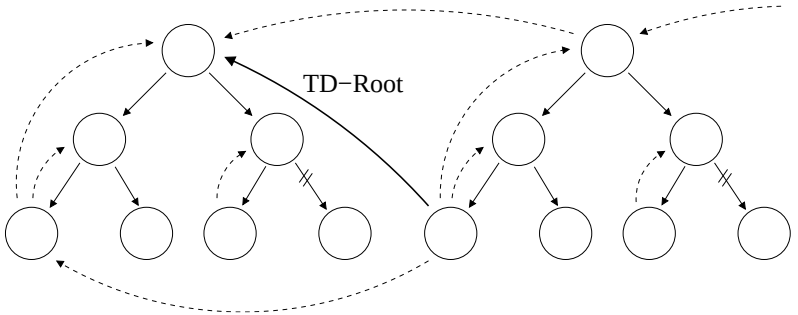
Learning and Tree Search



- TD-Learning can be applied even if we do not know the world model. But, in this case we **do** know the world model (rules of the game)
- Can we make use of the valuable information in the search tree?

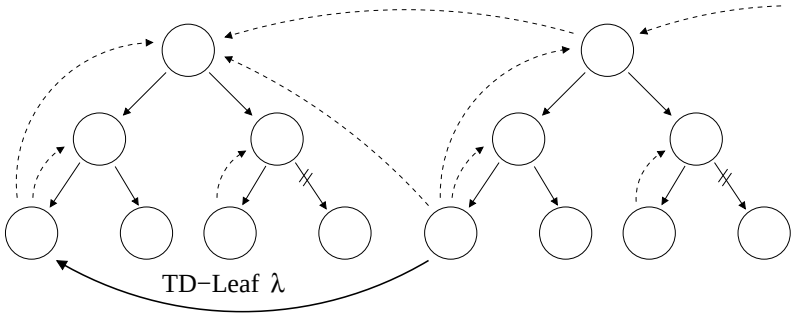
Checkers Program (Arthur Samuel)

TD-Root



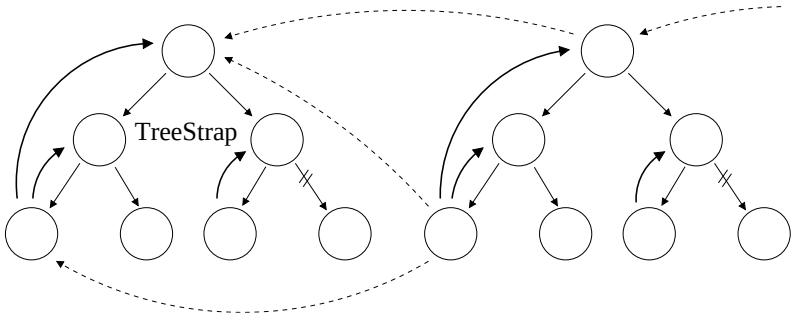
- TD(λ) (Sutton 1988, Tesauro 1992)
- TD-Root (Samuel 1959)

TD-Leaf



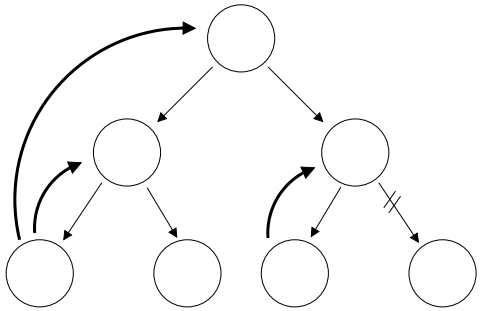
- TD(λ) (Sutton 1988, Tesauro 1992)
- TD-Root (Samuel 1959)
- TD-Leaf(λ) (Baxter et al. 1998)

TreeStrap



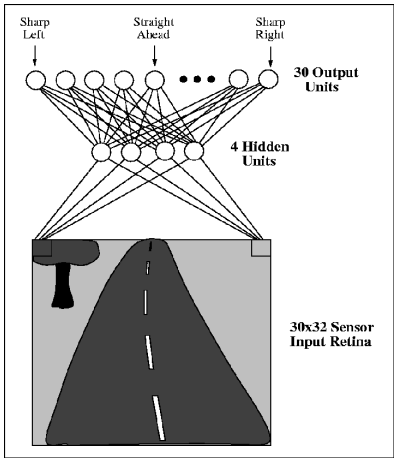
- TD(λ) (Sutton 1988, Tesauro 1992)
- TD-Root (Samuel 1959)
- TD-Leaf(λ) (Baxter et al. 1998)
- TreeStrap (Veness et al. 2009)

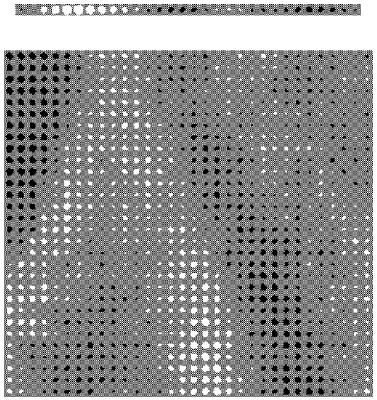
TreeStrap algorithm



- all non-leaf positions are updated (including moves not selected)
- when alpha-beta causes a cutoff, we can still train towards the upper or lower bound

ALVINN autonomous driver



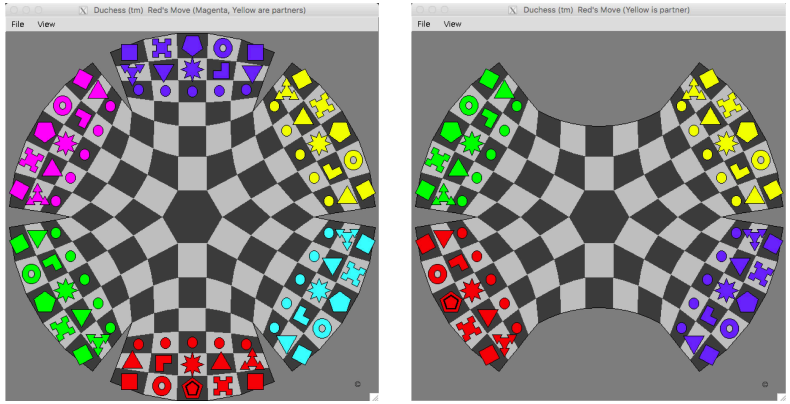


TreeStrap for Chess

- showed for the first time that a Chess player could be trained to Master level entirely by self-play, from random initial weights
- learning sometimes became unstable
  - learning rate had to be carefully chosen
  - had to put a limit on the size of individual weight updates
  - we have since found that scaling the learning rate by depth of the node makes learning more stable

The Game of Duchess

Duchess in Java



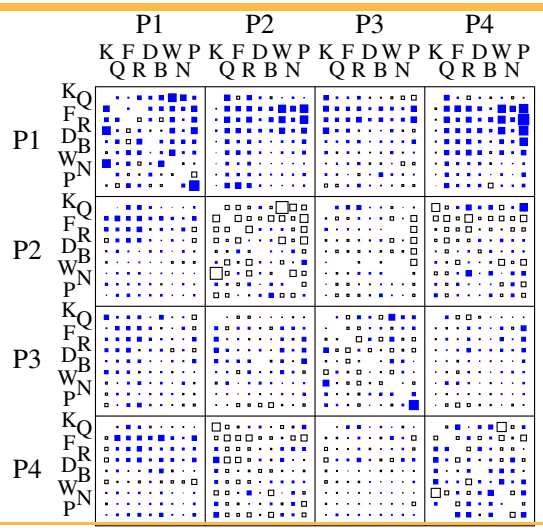
How Many Features?

	Chess	Duchess
material:	6	9
check/checkmate:	2	$2\times4=$ 8
attack/defend:	$6\times2\times6\times2=144$	$9\times4\times9\times4=1296$
piece-square:	$64\times6\times2=768$	$117\times9\times4=4212$
total:	920	5525
branching factor	24	50
depth of search	14	5-9

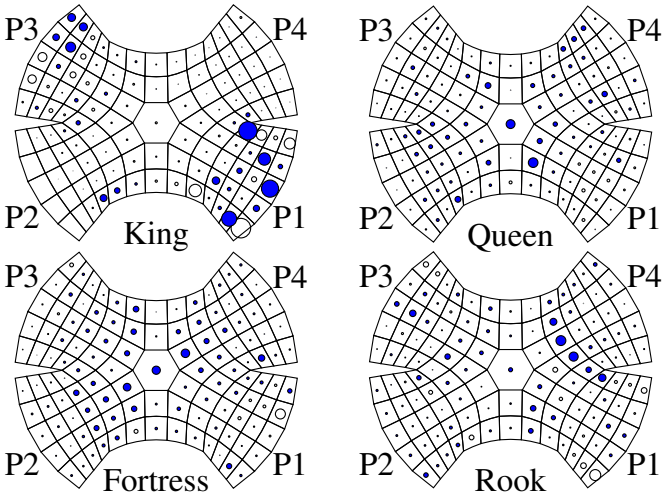
Alpha-Beta Optimizations

	Chess	Duchess
iterative deepening	Yes	Yes
killer move heuristic	Yes	Yes
best reply heuristic	Yes	(Maybe)
history heuristic	Yes	(Maybe)
hash table	Yes	(Current Work)
quiescent search	Yes	No

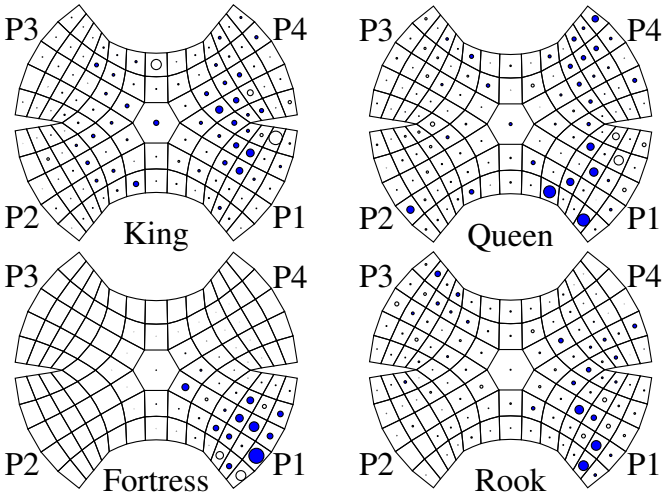
Attack/Defend Weights



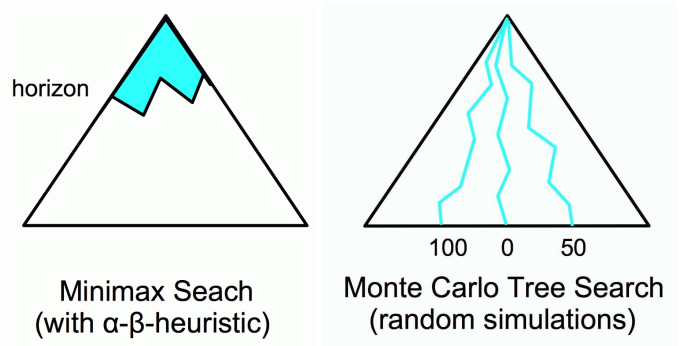
Trained Weights: Piece-Square



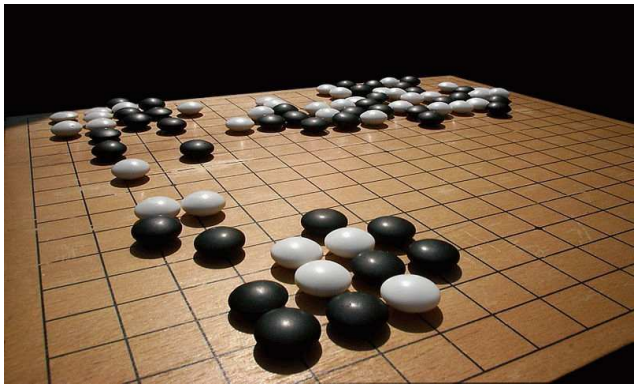
Trained Weights: Piece-Square



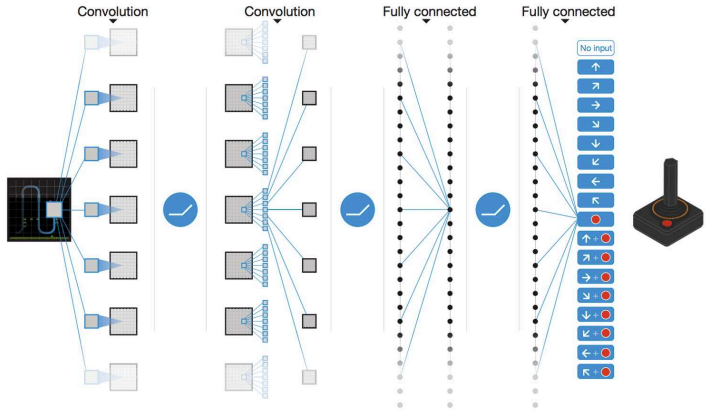
Monte Carlo Tree Search



AlphaGo, Alpha Zero



Deep Q-Learning for Atari Games



Summary

- Games can be learned from human expert preferences, or from self-play (or a combination)
- TD-Learning is a general method, which does not rely on knowing the world model
- TreeStrap is more powerful, because it also refines the value of moves which were not chosen; but it relies on knowing the world model
- Monte Carlo tree search good for games with large branching factor
- Deep Learning for Go, Atari Games
- Starcraft?