Machine Learning in Game Search

Outline

Many design decisions need to be fine-tuned in game playing agents

Can we automatically tune these decisions?

This week will give you a basic introduction to:

- Search control learnng
- Monte Carlo tree search

Challenges in game agent design

♢ Handling each stage of the game separately
  $\rightarrow$ compile a "book" of opening games or end games

♢ Adjusting search parameters
  $\rightarrow$ search control learning

♢ Adjusting weights in evaluation functions

♢ Finding good features for evaluation functions

Book learning

Aim: learn sequence of moves for important positions

Example1: In chess, there are books of opening moves
e.g. for every position seen in an opening game,
remember the move taken and the final outcome

Example 2: Learn from mistakes
e.g. identify moves that lead to a loss,
and whether there was a better alternative

Question: How do we recognise *which moves* were important?

Search control learning

Aim: learn how to make search more efficient

Example1: Order of move generation affects α–β pruning
e.g. learn a preferred order for generating possible moves
to maximise pruning of subtrees

Example 2: Can we vary the cut-off depth?
e.g. learn a classifier to predict what depth we should search to
based on current states

Learning evaluation function weights

Aim: adjust weights in evaluation function based on experience of their ability to predict the *true final utility*

Recall:
Typically use a *linear weighted sum of features*

$$\begin{aligned}\mathrm{Eval}(s) &= w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s) \\ &= \sum_{i=1}^{n} w_i f_i(s) \\ &= \boldsymbol{w} \boldsymbol{.} \boldsymbol{f}(s)\end{aligned}$$

Supervised learning

In *supervised learning*, we are given a training set of examples $D = \{d_1, d_2, ...\}$, where each training example d comprises a vector of inputs along with the desired output t, i.e.

$$d = \langle x_1, ..., x_n, t \rangle$$

In the case of learning the weights of an evaluation function, the training example corresponds to the set of features for a state s, with the true minimax utility value of the state as desired output, i.e.

$$d = \langle f_1(s), ..., f_n(s), U(s) \rangle$$

The aim is to learn a set of weights $w = \{w_1, ..., w_n\}$ so that the actual output $z = \textsc{Eval}(s; w)$ closely approximates the desired (true) output $t = U(s)$ on the training examples, and hopefully on new states as well.

Problems

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

Credit assignment: need to know which action(s) was responsible for the outcome

Temporal difference learning

Supervised learning is for single step prediction
e.g. predict Saturday's weather based on Friday's weather

Temporal Difference (TD) learning is for multi-step prediction
e.g. predict Saturday's weather based on Monday's weather,
then update prediction based on Tue's, Wed's, ...
e.g. predict outcome of game based on first move,
then update prediction as more moves are made

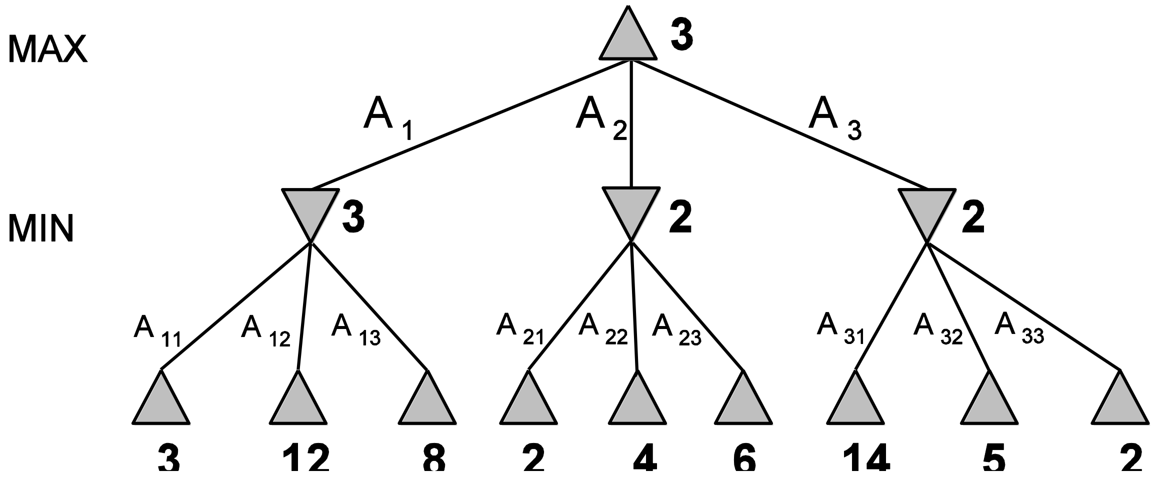
- Correctness of prediction not known until several steps later
- Intermediate steps provide information about correctness of prediction
- TD learning is a form of **reinforcement learning**
- Tesauro (1992) applied TD learning to Backgammon
  i.e. given state of game, predict probability of winning

TDLeaf(λ) algorithm

Proposed in [Baxter, Tridgell and Weaver 1998]

Combines temporal difference learning with minimax search

Basic idea: update weight in evaluation function to reduce differences in rewards predicted at different levels in search tree
$\rightarrow$ good functions should be stable from one move to next



What's examinable?

Up to 2022, we taught the detals of supervised learning and temporal difference learning for learning evaluation functions in minimax search

From 2023, the details of those methods are no longer examinable (but we mention them in case you feel adventurous and want to experiment with them in Part B of the Project - the old slides are available on the LMS)

Instead, we now teach an alternative reinforcement learning technique that was made popular by AlphaGo: **Monte Carlo tree search**

Monte Carlo Tree Search - Motivation

In some games it can be hard to find a good evaluation function

For example, in Go the material value of the pieces on the board is not a good predictor of the outcome from a state

Also, the positions of the pieces can change considerably throughout the game

Finally, the branching factor is so high that even alpha-beta pruning can only look a small number of moves ahead

Monte Carlo Tree Search (MCTS) was invented as an alternative approach as it **does not depend on a heuristic evaluation function**

MCTS - Using Playout

MCTS abandons the use of complete search using a cut-off depth and an evaluation function

Instead, MCTS estimates the average utilty of a state by playing multiple games from that state all the way to a terminal state of the game using an initial move selection policy

The value of the state is then estimated from the average of the outcomes of those games (rather than using a heuristic evaluation function)

Each game that is played from a state to a terminal state is called a **playout** or **simulation**

After many playouts from a state, we can estimate the average utilty of the state (if the only outcome is a win/loss, the average utility is just the win percentage of playouts from that state)

MCTS - Playout Policy

But how do we decide which moves to make during a playout?

- Random moves - will learn very slowly
- Game specific heuristics - based on expert knowledge
- Learned evaluation function based on self-play - guides search during playout, but the outcome of the playout is based on the terminal state, not the evaluation function

If we can sample enough playouts, we can **measure** the average utility of a state, rather than rely on a heuristic evaluation function to “guess” the state’s utility

MCTS - Key Steps

We gradually expand the search tree using the following four steps:

- **Selection** - Starting from the root of the tree, use a selection strategy to choose successor states until we reach a leaf node of the tree
- **Expansion** - From the selected leaf node, expand the node by adding to the tree a **single new child** from that leaf node
- **Simulation** - Perform a playout from the newly added node to a terminal state, but don't add the moves/states explored in the playout to the search tree
- **Backpropagation** - Use the outcome from the playout (win/loss) to update the statistics of each node from the newly added node back up to the route

Note that at each node in the tree we maintain statistics on the **number of playouts** and **number of wins** from that node

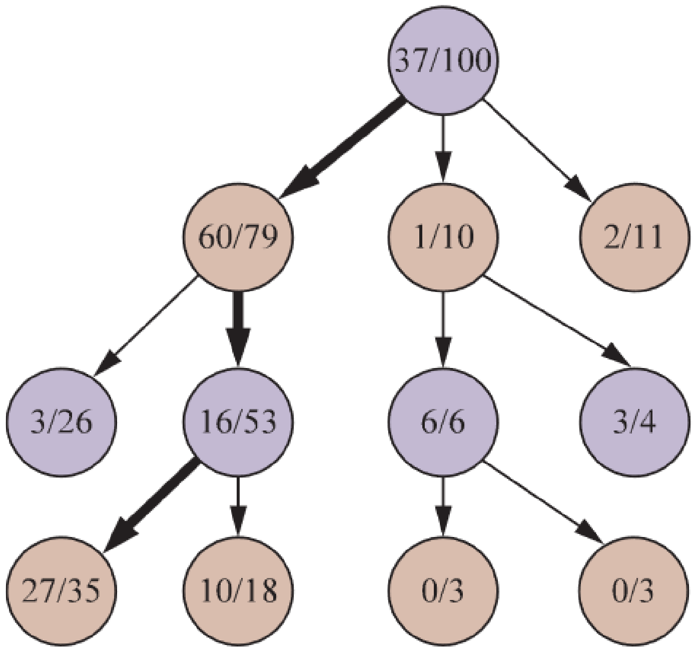
MCTS Algorithm

```
function MONTE-CARLO-TREE-SEARCH(state) returns an action
    tree ← NODE(state)
    while IS-TIME-REMAINING() do
        leaf ← SELECT(tree)
        child ← EXPAND(leaf)
        result ← SIMULATE(child)
        BACK-PROPAGATE(result, child)
    return the move in ACTIONS(state) whose node has highest number of playouts
```

Figure 5.11 The Monte Carlo tree search algorithm. A game tree, *tree*, is initialized, and then we repeat a cycle of SELECT / EXPAND / SIMULATE / BACK-PROPAGATE until we run out of time, and return the move that led to the node with the highest number of playouts.

MCTS Example - Selection

Starting from the root, select the leaf to expand next



(a) Selection

MCTS Example - Selection

Require a selection policy to decide which leaf node to expand - this is a trade-off between **exploiting** nodes that are known to have a high win percentage versus **exploring** nodes that have seldom been tested

Among the nodes at depth 2, the leftmost node has led to a win 60 times out of 79 playouts, while the rightmost node has led to a win 2 times out of 11 playouts - by chosing the leftmost node we are exploiting moves that have proven to be successful in the past, but maybe we still need to explore the other two nodes at that depth which have a higher uncertainty about their success rate

MCTS Example - Selection

A selection policy that we can use from reinforcement learning theory is called **upper confidence bound applied to trees**, and has the formula $UCB1(n)$ for a node n

$$UCB1(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{logN(Parent(n))}{N(n)}}$$

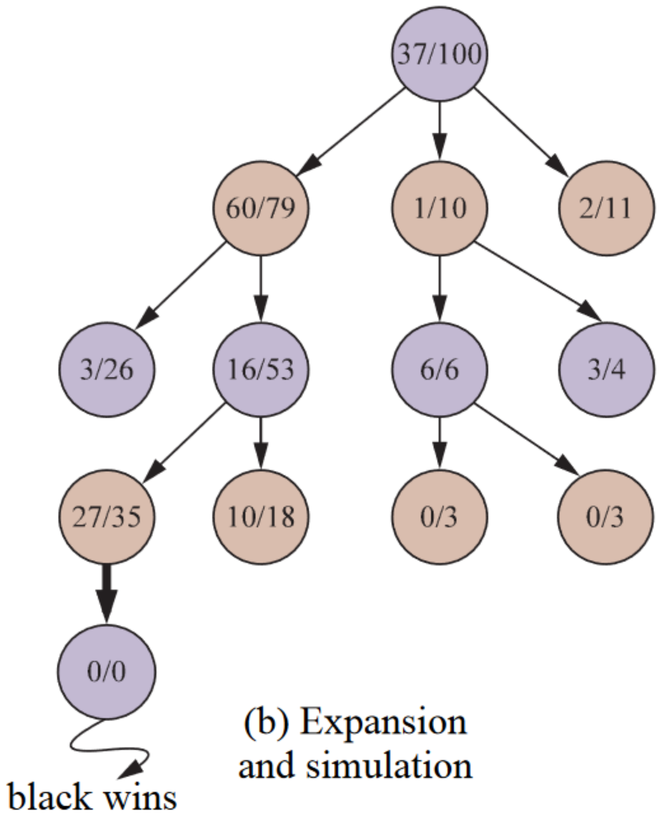
where

- $U(n)$ - total utility (e.g., no. of wins) from all playouts through node n
- $N(n)$ - total number of playouts through node n
- $Parent(n)$ - parent node of n in the tree
- C - a constant that balances the exploitation term $\frac{U(n)}{N(n)}$ and the exploration term $\sqrt{\frac{logN(Parent(n))}{N(n)}}$

When choosing between sibling nodes, select the node with the highest $UCB1$ value. If a node has few playouts compared to its parent, then the second term will be larger

MCTS Example - Expansion and Simulation

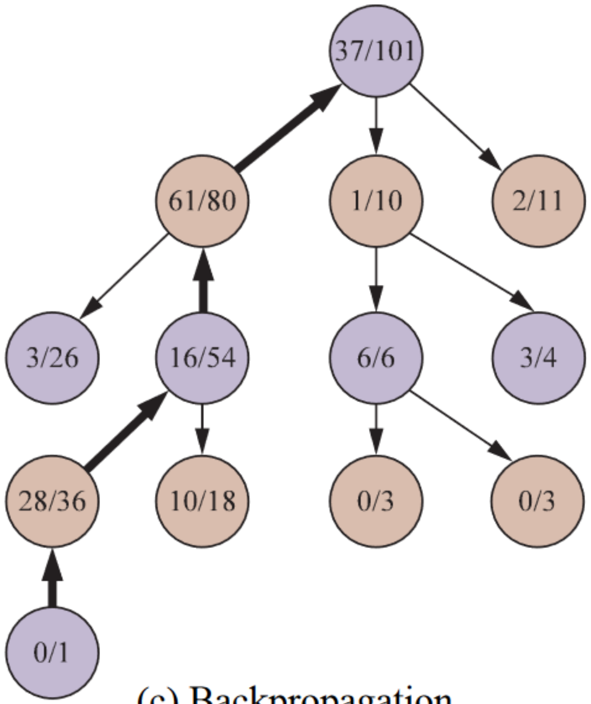
From the selected leaf, add one child as a new leaf to the tree. Perform playout (i.e., simulation) from the child node, selecting moves according to a playout policy. But don't add the nodes in the playout to the tree



(b) Expansion and simulation

MCTS Example - Backpropagation

Use outcome of playout to update all ancestor nodes of the new leaf that was added to the tree. Note that if black wins in the playout, we only update the utility (number of wins) for the black nodes, and vice versa if white wins



(c) Backpropagation

MCTS Comments

♢ Complexity of playout is *linear* in depth of tree, not exponential, since we just pick a move at each level of the playout rather than searching the whole subtree

♢ Can perform many, many playouts in the time it would take alpha-beta search to look a few moves ahead

♢ Playouts allow us to learn from the actual outcome of a game, rather than using a heuristic

♢ By focusing on moves that have a high win rate, but still allowing some flexibility to explore other moves, we can make a more informed choice about which move to take

♢ We still need a playout policy to pick which moves to take during a playout. Sometimes this policy can be learned by self-play

Summary

◇ Introduction to learning in games

◇ Monte Carlo Tree Search for learning in games

Examples of skills expected:

◇ Discuss opportunities for learning in game playing

◇ Explain differences between exploration and exploitation in learning to play games

◇ Trace the operation of MCTS on an example search tree