A new algorithm will be presented for learning heuristic evaluation functions for game tree search via self-play. Topics covered include:

- Why self-play is important
- The search bootstrapping approach
- Relationship to TD(\(\lambda\)), TD-Leaf(\(\lambda\))
- Empirical evaluation on Chess

Heuristic Evaluation Function

A heurisitc evaluation function is a mapping \(H(s) : \text{State} \rightarrow \mathbb{R}\).

For this work we use:

- Parameterised representation: \(H(s; \vec{w}), \vec{w} \in \mathbb{R}^n\)
- State dependent feature vector: \(\phi(s) : \text{State} \rightarrow \mathbb{R}^n\)
- Linear combination of features: \(H(s; \vec{w}) := \phi(s) \cdot \vec{w}\)

Problem: how to find good \(\vec{w}\)?
Constructing Evaluation Functions

Some alternative methods to find weights:
- Hand-tune (guess and test)
- Supervised learning / learn from expert play
- Self-play

Self-play has a number of potential benefits:
- No need for scored training examples
- Reduced knowledge engineering effort
  but can be hard to achieve in practice.

Updating Evaluation Functions

We will be frequently talking about updating $H(s; \vec{w})$ towards some target value $T \in \mathbb{R}$. The following methods we consider are all:
- Online
  - Use stochastic gradient descent on either the squared error $\frac{1}{2}(T - H(s; \vec{w}))^2$ or sum squared error $\frac{1}{2} \sum_s (T_s - H(s; \vec{w}))^2$
- Invoked after a real action (move) is taken

For this talk, we are more interested in exactly how we choose the training target/s.

Self-play with TD Learning

- Famously applied to Backgammon (TD-Gammon) by Tesauro
- Simple greedy action selection sufficed during training
- ...unfortunately, difficulties with highly tactical domains (e.g. Chess).

TD-Leaf Learning

Introduced by Baxter et al. Combines game tree search and TD learning. Some well-known applications: Chess (KnightCap) and Checkers (Chinook).
Limitations of TD-Leaf

Although undoubtedly an improvement over TD for certain types of games, a number of issues remain:

- Difficult to achieve strong results from just randomly assigned weights.
- Expert play in chess emerged only after material weights were initialised to expert values and likely opponent blunders were excluded.
- KnightCap required carefully controlled learning regime to learn. Is deterministic case harder than stochastic case?
- Higher computational overhead compared to TD($\lambda$)

<table>
<thead>
<tr>
<th>Program</th>
<th>Game</th>
<th>Weights</th>
<th>Self-play</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD-Gammon</td>
<td>Backgammon</td>
<td>Random</td>
<td>Yes</td>
<td>World Class</td>
</tr>
<tr>
<td>Chinook</td>
<td>Checkers</td>
<td>Mixed</td>
<td>Yes</td>
<td>World Class</td>
</tr>
<tr>
<td>KnightCap</td>
<td>Chess</td>
<td>Mixed</td>
<td>No</td>
<td>Expert/Master</td>
</tr>
<tr>
<td>Meep (Us)</td>
<td>Chess</td>
<td>Random</td>
<td>Yes</td>
<td>Expert/Master</td>
</tr>
</tbody>
</table>

Notes:

- Results of Knightcap, starting from random weights, trained via self-play were disappointing.
- The value of a checker and a king were fixed in Chinook.

An obvious, but important point...

The distribution over:

---

*Positions seen in search ≠ Positions seen over the board*

E.g. Contrast:

Tree Strap: an alternative backup scheme

Consider the following modified backup policy:

\[
\begin{align*}
\text{time} = t & \quad \text{Backup policy at } t \\
\text{time} = t+1 & \quad \text{Backup policy at } t+1
\end{align*}
\]
TreeStrap Properties

Three main points:
- Backups come from deeper search at the same time-step, not subsequent searches.
- A single search provides many updates; potential to learn faster?
- Training examples come from more “representative” positions; potentially more robust?

Implementation:
- Extended to alpha-beta search, uses one-sided loss function
- High performance programs all use transposition tables; bound information already available

Experimental Setup

- Heuristic evaluation consists of weighted linear combination of 1800 features
- 1m 1s Fischer Time controls used for training and evaluation (~5 mins)
- Time taken for updates reduced overall thinking time
- Over 25000 training games played to learn weights
- Over 16000 games played (time consuming!) in local evaluation tournament
- 2000 games used for online evaluation

Comparison to existing methods on Chess

Learning from self-play: Rating versus Number of training games

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Training Partner</th>
<th>Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>TreeStrap(αβ)</td>
<td>Self Play</td>
<td>1950-2197</td>
</tr>
<tr>
<td>TreeStrap(αβ)</td>
<td>Shredder</td>
<td>2154-2338</td>
</tr>
</tbody>
</table>

- Self-play weights correspond to expert/weak master level play
- Strong opponent weights correspond to master level play
- Scored 13.5/15 against International Master opposition online
- Learning by playing a strong opponent helps, but effect is not as pronounced compared to TD-Leaf
TreeStrap(·) method introduced, alternative to TD-based approaches for self-play training for games.

With respect to Chess:
- Order of magnitude reduction in training time vs TD-Leaf
- Simple greedy move selection sufficient for training
- First successful self-play result, starting from entirely random weights!

Thankyou for listening. Please visit us at W38 this evening, especially if you are interested in talking about:
- algorithmic details, e.g. TreeStrap(αβ)
- details of the chess specific features
- how playing strength is measured
- relationship of TreeStrap to other Reinforcement Learning techniques
- ways in which this work can be extended