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:

- Supervised learning using gradient decent search
- Temporal difference learning in games

Challenges in game agent design

- ♢ Handling each stage of the game separately
  → compile a "book" of opening games or end games
- ♢ Adjusting search parameters
  → 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*

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

Gradient descent 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.

Gradient descent learning

How do we quantify how well the actual output z approximates the desired output t?

We define an *error function* E to be (half) the sum over all training examples of the square of the difference between the actual output z and the desired output t

$$\text{error } E = \frac{1}{2}\Sigma(t-z)^2$$

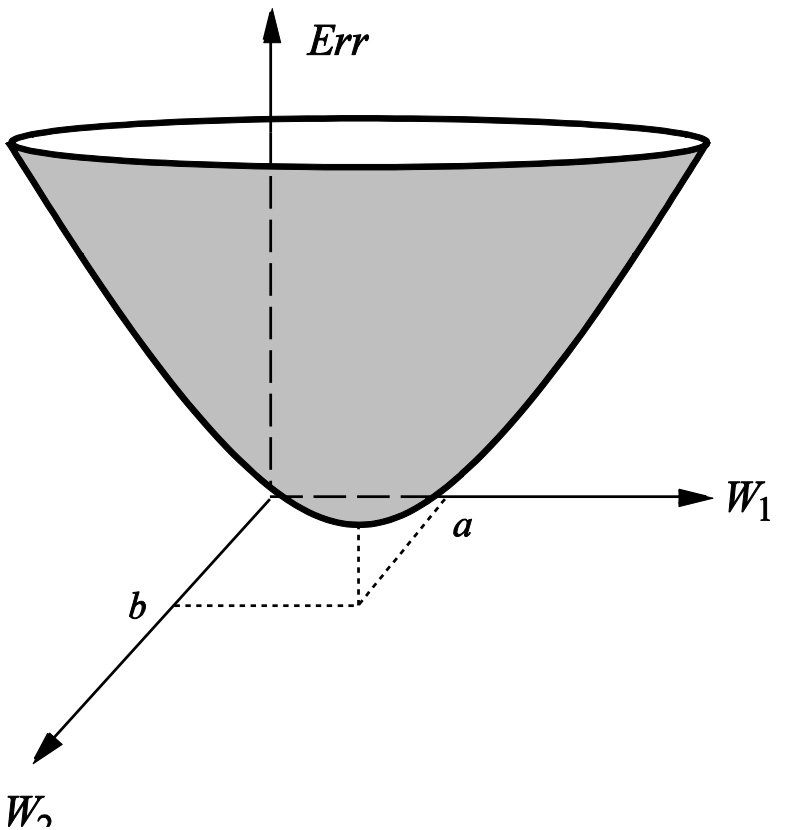
If we think of E as height, it defines an *error landscape* on the weight space. The aim is to find a set of weight values for which E is very low on the training examples. This is done by moving in the *steepest downhill direction*, i.e., $\partial E/\partial w_i$

$$w_i \leftarrow w_i - \partial E/\partial w_i$$

The gradient descent approach is the basis of the *back-propagation algorithm* in neural networks for supervised learning

Example error landscape

An example error landscape for gradient descent search in weight space



Weight update rule

To derive the weight update rule, we need to remember the chain rule from differential calculus:

if $y = y(u)$ and $u = u(x)$

then $\partial y/\partial x = (\partial y/\partial u)(\partial u/\partial x)$

So, if $z = \textsc{Eval}(s; w)$ and $t =$ true utility of state s

$$\begin{aligned}\partial E/\partial w_i &= \partial/\partial z\,[\frac{1}{2}(t-z)^2]\,\partial z/\partial w_i \\ &= (z-t)\,\partial z/\partial w_i \\ &= (z-t)\,f_i(s)\end{aligned}$$

Normally we include a learning rate parameter $\eta = 0.1$ to control the update of weights in a stable manner

$$w_i \leftarrow w_i - \eta\,(z-t) f_i(s)$$

We repeatedly update the weights based on each example until the weights converge

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

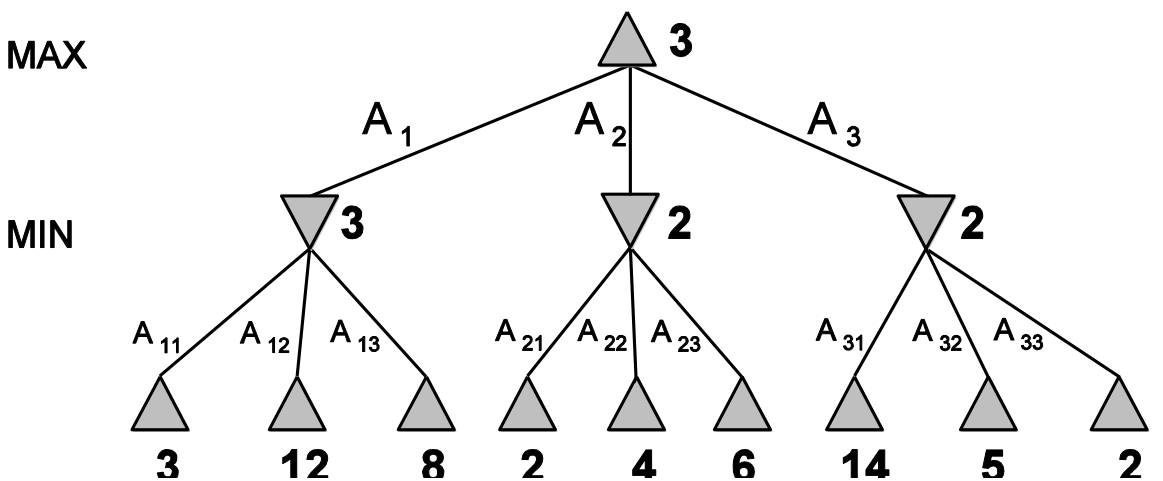
  i.e. given state of game, predict probability of winning

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



TDLeaf(λ) algorithm – notation

$eval(s, w)$: evaluation function for state s with parameters $w = [w_1, ..., w_k]$

$s_1, ..., s_N$: the N states that occurred during a game

$r(s_N)$: reward based on outcome of game $\{+1, 0, -1\}$

s_i^l: best leaf found at max cut-off depth using minimax search starting at state s_i

We can convert an evaluation score into a reward using

$$r(s_i^l, w) = \tanh(eval(s_i^l, w)$$

where $\tanh$ squashes the evaluation score into the range $[-1, +1]$

TDLeaf(λ) algorithm

For $i = 1, ..., N-1$

Compute temporal difference between successive states

$$d_i = r(s^l_{i+1}, w) - r(s^l_i, w)$$

Update each weight parameter w_j as follows

$$w_j \leftarrow w_j + \eta \sum_{i=1}^{N-1} \frac{\partial r(s^l_i, w)}{\partial w_j} [\sum_{m=1}^{N-1} \lambda^{m-i} \, d_m]$$

where η is the learning rate

if $\lambda = 0$**:** weights adjusted to move $r(s_i, w)$ towards $r(s_{i+1}, w)$

i.e. the predicted reward at the next state

if $\lambda = 1$**:** weights adjusted to move $r(s_i, w)$ towards $r(s_N)$

i.e. the final true reward (better if eval() is unrealistic)

Environment for learning

- ♢ Learning from labelled examples
- ♢ Learning by playing a skilled opponent
- ♢ Learning by playing against random moves
- ♢ Learning by playing against yourself

Summary

♢ Introduction to role for learning in games

♢ Supervised learning using gradient descent search

♢ Temporal difference learning in games using TDLeaf(λ) algorithm

Examples of skills expected:

♢ Discuss opportunities for learning in game playing

♢ Explain differences between supervised and temporal difference learning

♢ I do not expect you to derive or memorise the TDLeaf(λ) weight update rule, but if given this rule I may ask you to explain what the main terms mean