Learning to Play Games
IEEE CIG 2008 Tutorial

Simon M. Lucas
University of Essex, UK
Aims

• To provide a practical guide to the main machine learning methods used to learn game strategy
• Provide insights into when each method is likely to work best
  – Details can mean difference between success and failure
• Common problems, and some solutions
• We assume you are familiar with
  – Neural networks: MLPs and Back-Propagation
  – Rudiments of evolutionary algorithms (evaluation, selection, reproduction/variation)
• Demonstrate TDL and Evolution in action
Overview

• Architecture (action selector v. value function)
• Learning algorithm (Evolution v. Temporal Difference Learning)
• Function approximation method
  – E.g. MLP or Table Function
  – Interpolated tables
• Information rates
• Sample games (Mountain Car, Othello, Ms. Pac-Man)
Architecture

• Where does the computational intelligence fit in to a game playing agent?

• Two main choices
  – Value function
  – Action selector

• First, let’s see how this works in a simple grid world
Action Selector

• Maps observed current game state to desired action
• For
  – No need for internal game model
  – Fast operation when trained
• Against
  – More training iterations needed (more parameters to set)
  – May need filtering to produce legal actions
  – Separate actuators may need to be coordinated
State Value Function

• Hypothetically apply possible actions to current state to generate set of possible next states
• Evaluate these using value function
• Pick the action that leads to the most favourable state
• For
  – Easy to apply, learns relatively quickly
• Against
  – Need a model of the system
Grid World

- n x n grid (toroidal i.e. wrap-around)
- Reward: 0 at goal, -1 elsewhere
- State: current square \(\{i, j\}\)
- Actions: up, down, left, right
- Red Disc: current state
- Red circles: possible next states
- Each episode: start at random place on grid and take actions according to policy until the goal is reached, or maximum iterations have been reached
- Examples below use 15 x 15 grid
State Value versus State-Action Value: Grid World Example

- **State value**: consider the four states reachable from the current state by the set of possible actions
  - choose action that leads to highest value state

- **State-Action Value**
  - Take the action that has the highest value given the current state
Run Demo:
Time to see each approach in action
Learning Algorithm: (Co) Evolution v. TDL

• Temporal Difference Learning
  – Often learns much faster
  – But less robust
  – Learns during game-play
  – Uses information readily available (i.e. current observable game-state)

• Evolution / Co-evolution (vanilla form)
  – Information from game result(s)
  – Easier to apply
  – But wasteful: discards so much information

• Both can learn game strategy from scratch
Co-evolution (single population)
Evolutionary algorithm: rank them using a league

<table>
<thead>
<tr>
<th>Team</th>
<th>P</th>
<th>W</th>
<th>D</th>
<th>L</th>
<th>F</th>
<th>A</th>
<th>GD</th>
<th>PTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arsenal</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>15</td>
<td>4</td>
<td>11</td>
<td>16</td>
</tr>
<tr>
<td>Man Utd</td>
<td>7</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>6</td>
<td>2</td>
<td>4</td>
<td>14</td>
</tr>
<tr>
<td>Manchester City</td>
<td>7</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>5</td>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td>Liverpool</td>
<td>6</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>11</td>
<td>2</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>Newcastle</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>9</td>
<td>5</td>
<td>4</td>
<td>11</td>
</tr>
<tr>
<td>Chelsea</td>
<td>7</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>7</td>
<td>8</td>
<td>-1</td>
<td>11</td>
</tr>
<tr>
<td>West Ham</td>
<td>6</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>9</td>
<td>6</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>Aston Villa</td>
<td>6</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>7</td>
<td>4</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>Everton</td>
<td>7</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>Blackburn</td>
<td>6</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>Portsmouth</td>
<td>7</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>Wigan</td>
<td>7</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>8</td>
<td>7</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>Middlesbrough</td>
<td>7</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>9</td>
<td>11</td>
<td>-2</td>
<td>8</td>
</tr>
<tr>
<td>Birmingham</td>
<td>7</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>9</td>
<td>-2</td>
<td>8</td>
</tr>
<tr>
<td>Sunderland</td>
<td>7</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>11</td>
<td>-4</td>
<td>8</td>
</tr>
<tr>
<td>Reading</td>
<td>7</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>11</td>
<td>-6</td>
<td>7</td>
</tr>
<tr>
<td>Fulham</td>
<td>7</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>12</td>
<td>14</td>
<td>-2</td>
<td>6</td>
</tr>
<tr>
<td>Tottenham</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>10</td>
<td>12</td>
<td>-2</td>
<td>5</td>
</tr>
<tr>
<td>Bolton</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td>12</td>
<td>-4</td>
<td>4</td>
</tr>
<tr>
<td>Derby</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>20</td>
<td>-16</td>
<td>4</td>
</tr>
</tbody>
</table>
In Pictures...
Information Flow

• Interesting to observe information flow
• Simulating games can be expensive
• Want to make the most of that computational effort
• Interesting to consider bounds on information gained per episode (e.g. per game)
• Consider upper bounds
  – All events considered equiprobable
Evolution

• Suppose we run a co-evolution league with 30 players in a round robin league (each playing home and away)
• Need n(n-1) games
• Single parent: pick one from n
• log_2(n)
• Information rate: \[ I_c = \frac{\log_2 n}{n(n - 1)} \]

<table>
<thead>
<tr>
<th>n</th>
<th>I_c (bg⁻¹)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.500</td>
</tr>
<tr>
<td>5</td>
<td>0.12</td>
</tr>
<tr>
<td>10</td>
<td>0.037</td>
</tr>
<tr>
<td>30</td>
<td>0.006</td>
</tr>
</tbody>
</table>
TDL

- Information is fed back as follows:
  - 1.6 bits at end of game (win/lose/draw)
- In Othello, 60 moves
- Average branching factor of 7
  - 2.8 bits of information per move
  - $60 \times 2.8 = 168$
- Therefore:
  - Up to nearly 170 bits per game (> 20,000 times more than coevolution for this scenario)
  - (this bound is very loose – why?)
- See my CIG 2008 paper
Sample TDL Algorithm: TD(0)
typical alpha: 0.1
pi: policy; choose rand move 10% of time
else choose best state

---

Algorithm 1: On-line TD(0) adapted from Sutton and Barto

**INITIALIZE** $V(s)$ arbitrarily, for all $s \in S$

**for each episode do**

- Initialize $s$ to start state
  (could be random start state)

**for each step in episode do**

  - $a \leftarrow$ action given by $\pi$ for $s$
  - Take action $a$, observe reward $r$, and next state $s'$
  - $\delta \leftarrow r + V(s') - V(s)$
  - $V(s) \leftarrow V(s) + \alpha \delta$

end
end
Main Software Modules
(my setup – plug in game of choice)

- Problem Adapter
- Game Engine
- Game Agent Controller
- Vector Optimiser
- Vectoriser
- Value Function
- Interpolated Table
- Radial Basis Function
- Interpolated Table
- MLP
- TDL
- ES
- EDA
Function Approximators

- For small games (e.g. OXO) game state is so small that state values can be stored directly in a table.
- Our focus is on more complex games, where this is simply not possible e.g.
  - Discrete but large (Chess, Go, Othello, Pac-Man)
  - Continuous (Mountain Car, Halo, Car racing: TORCS)
- Therefore necessary to use a function approximation technique.
Function Approximators

• Multi-Layer Perceptrons (MLPs)
  – Very general
  – Can cope with high-dimensional input
  – Global nature can make forgetting a problem

• N-Tuple systems
  – Good for discrete inputs (e.g. board games)
  – Harder to apply to continuous domains

• Table-based
  – Naïve is poor for continuous domains
  – CMAC coding improves this (overlapping tiles)
  – Even better: use interpolated tables
    • Generalisation of bilinear interpolation used in image transforms
Standard (left) versus CMAC (right)
Interpolated Table
Method

• Continuous point $p(x,y)$
• $x$ and $y$ are discretised, then residues $r(x)$ $r(y)$ are used to interpolate between values at four corner points
• N-dimensional table requires $2^n$ lookups

$$f_t(x,y) = (1 - r(x))(1 - r(y))t[q_l(x)][q_l(y)]$$
$$+ r(x)(1 - r(y))t[q_u(x)][q_l(y)]$$
$$+ (1 - r(x))r(y)t[q_l(x)][q_u(y)]$$
$$+ r(x)r(y)t[q_u(x)][q_u(y)]$$
Supervised Training Test

• Following based on 50,000 one-shot training samples
• Each point randomly chosen from uniform distribution over input space
• Function to learn: continuous spiral ($r$ and $\theta$ are the polar coordinates of $x$ and $y$)

$$f(x, y) = \sin(\theta + r\pi \omega)$$
Results

<table>
<thead>
<tr>
<th>True</th>
<th>MLP-BP-25</th>
<th>Ntuple: CMAC + Grey</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table</td>
<td>CMAC</td>
<td>Bilinear</td>
</tr>
</tbody>
</table>

MLP-CMAES
# Test Set MSE

<table>
<thead>
<tr>
<th>Architecture</th>
<th>MSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLP</td>
<td>0.13</td>
</tr>
<tr>
<td>N-Tuple (CMAC + Grey)</td>
<td>0.30</td>
</tr>
<tr>
<td>Standard Table</td>
<td>0.08</td>
</tr>
<tr>
<td>CMAC (Shared)</td>
<td>0.01</td>
</tr>
<tr>
<td>Bi-Linear</td>
<td>0.006</td>
</tr>
</tbody>
</table>
Standard Regression
200 Training Points
Gaussian Processes Model

Data set  Interpolated Table  Gaussian Processes

Gaussian Processes: learn more from the data, but hard to interface to games
Function Approximator: Adaptation Demo
This shows each method after a single presentation of each of six patterns, three positive, three negative. What do you notice?
Grid World – Evolved MLP

- MLP evolved using CMA-ES
- Gets close to optimal after a few thousand fitness evaluations
- Each one based on 10 or 20 episodes
- Value functions may differ from run to run
Evolved N-Linear Table

• This was evolved using CMA-ES, but only had a fitness of around 80
Evolved N-Linear Table with Lamarkian TD-Learning

• This does better
• Average score now 8.4
TDL Again

• Note how quickly it converges with the small grid
• Surprisingly hard to make it work!
Table Function TDL (15 x 15)

- Typical score of 11.0
- Not as good as interpolated 5 x 5 table on this task
- Model selection is important
Grid World Results – State Table

• Interesting!
• The MLP / TDL combination is very poor
• Evolution with MLP gets close to TDL with N-Linear table, but at much greater computational cost

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Evolution (CMA-ES)</th>
<th>TDL(0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MLP (15 hidden units)</td>
<td>9.0</td>
<td>126.0</td>
</tr>
<tr>
<td>N-Linear Table (5 x 5)</td>
<td>11.0</td>
<td>8.4</td>
</tr>
</tbody>
</table>
Action Values - Takes longer
e.g. score of 9.8 after 4,000 episodes
Simple Example: Mountain Car

- Standard reinforcement learning benchmark
- Accelerate a car to reach goal at top of incline
- Engine force weaker than gravity
Value Functions Learned (TDL)
## Mountain Car Results
(TDL, 2000 episodes, ave. of 10 runs)

<table>
<thead>
<tr>
<th>System</th>
<th>Mean steps to goal (s.e.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table</td>
<td>1008 (143)</td>
</tr>
<tr>
<td>CMAC: separate</td>
<td>81.8 (11.5)</td>
</tr>
<tr>
<td>CMAC: shared</td>
<td>60.0 (2.3)</td>
</tr>
<tr>
<td>Bilinear</td>
<td>50.5 (2.5)</td>
</tr>
</tbody>
</table>
Othello

See Demo
Volatile Piece Difference

![Graph showing Volatile Piece Difference with a line graph indicating score changes over time. The graph has a y-axis ranging from -22 to 11, and an x-axis labeled 'move' ranging from 0 to 60.]
Setup

• Use weighted piece counter
  – Fast to compute (can play billions of games)
  – Easy to visualise
  – See if we can beat the ‘standard’ weights

• Limit search depth to 1-ply
  – Enables billions of games to be played
  – For a thorough comparison

• Focus on machine learning rather than game-tree search

• Force random moves (with prob. 0.1)
  – Get a more robust evaluation of playing ability
Othello: After-state Value Function
Standard “Heuristic” Weights (lighter = more advantageous)
TDL Algorithm

• Nearly as simple to apply as CEL
  public interface TDLPlayer extends Player {
    void inGameUpdate(double[] prev, double[] next);
    void terminalUpdate(double[] prev, double tg);
    \[ \cdot \alpha [v(x') - v(x)] (1 - v(x)^2)x_i \]
  }

• Reward signal only given at game end
• Initial alpha and alpha cooling rate tuned empirically
TDL in Java

public void inGameUpdate(double[] prev, double[] next) {
    double op = tanh(net.forward(prev));
    double tg = tanh(net.forward(next));
    double delta = alpha * (tg - op) * (1 - op * op);
    net.updateWeights(prev, delta);
}

public void terminalUpdate(double[] prev, double tg) {
    double op = tanh(net.forward(prev));
    double delta = alpha * (tg - op) * (1 - op * op);
    net.updateWeights(prev, delta);
}
CEL Algorithm

• Evolution Strategy (ES)
  – (1, 10) (non-elitist worked best)
• Gaussian mutation
  – Fixed sigma (not adaptive)
  – Fixed works just as well here
• Fitness defined by full round-robin league performance (e.g. 1, 0, -1 for w/d/l)
• Parent child averaging
  – Defeats noise inherent in fitness evaluation
Algorithm in detail
(Lucas and Runarsson, CIG 2006)

1. **Initialize**: $w' = 0$ and $\beta = 0.05$ (or 1.0)
2. **while** termination criteria not satisfied **do**
   3. **for** $k := 1$ to $\lambda$ **do** (replication)
      4. $w_k \leftarrow w' + \mathcal{N}(0, 1/n)$
   5. **end for**
   6. each individual $w_k$, $k = 1, \ldots, \lambda$ plays another (once each color) for a total of $\lambda(\lambda - 1)$ games,
   7. find the player $i$ with the highest score (breaking ties randomly)
   8. $w' \leftarrow w' + \beta(w_i - w')$ (arithmetic average)
9. **end while**
CEL (1,10) v. Heuristic

![Graph showing the probability of winning over games played for different values of ω and μ. The graph has lines for different parameter settings: β = 1.00, ε = 0.0; β = 0.05, ε = 0.0; β = 0.05, ε = 0.1.]
TDL v. Random and Heuristic

![Graph showing the probability of winning over games played. The graph compares the performance of TDL against random and heuristic strategies. The x-axis represents the number of games played, scaled by 1/45000. The y-axis represents the probability of winning. Two lines are plotted: one solid, indicating playing heuristic, and one dashed, indicating playing random.](image-url)
Othello: Symmetry

• Enforce symmetry
  – This speeds up learning

• Use trusty old friend: **N-Tuple System** for value approximator
NTuple Systems


• Sample n-tuples of input space

• Map sampled values to memory indexes
  – Training: adjust values there
  – Recognition / play: sum over the values

• Superfast

• Related to:
  – Kernel trick of SVM (non-linear map to high dimensional space; then linear model)
  – Kanerva’s sparse memory model
  – Also similar to Michael Buro’s look-up table for Logistello
Symmetric 3-tuple Example

\[ v(b) = \sum_{d \in D(b)} l(d) \]
Symmetric N-Tuple Sampling
N-Tuple System

- Results used 30 random n-tuples
- Snakes created by a random 6-step walk
  - Duplicates squares deleted
- System typically has around 15000 weights
- Simple training rule:

\[ l(d) = l(d) + \delta \quad \forall \, d \in D(b) \]
Algorithm 2: N-tuple training algorithm

NOTE: $f$ is the indexing function

INITIALIZE: set weights to zero

for $i$ in set of n-tuples do
    for $j$ in symmetries($i$) do
        index = $f_{ij}$(board)
        $l_i[index] += \delta$
    end
end
NTuple System (TDL)
total games = 1250
(very competitive performance)
Typical Learned strategy...
(N-Tuple player is +ve – 10 sample games shown)
Web-based League  
(May 15th 2008)  
All Leading entries are N-Tuple based

<table>
<thead>
<tr>
<th>Position</th>
<th>Name</th>
<th>Played</th>
<th>Won</th>
<th>Drawn</th>
<th>Lost</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>t15x6x8</td>
<td>100</td>
<td>79</td>
<td>3</td>
<td>18</td>
<td>SNT-Text</td>
</tr>
<tr>
<td>2</td>
<td>x30x6x8</td>
<td>100</td>
<td>71</td>
<td>4</td>
<td>25</td>
<td>SNT-Text</td>
</tr>
<tr>
<td>3</td>
<td>Stunner</td>
<td>100</td>
<td>67</td>
<td>1</td>
<td>32</td>
<td>SNT-Text</td>
</tr>
<tr>
<td>4</td>
<td>Woxy SNT</td>
<td>100</td>
<td>67</td>
<td>1</td>
<td>32</td>
<td>NET-WOX</td>
</tr>
<tr>
<td>5</td>
<td>WOX Test</td>
<td>100</td>
<td>65</td>
<td>1</td>
<td>34</td>
<td>NET-WOX</td>
</tr>
<tr>
<td>6</td>
<td>WOX Test 3</td>
<td>100</td>
<td>64</td>
<td>1</td>
<td>35</td>
<td>NET-WOX</td>
</tr>
<tr>
<td>7</td>
<td>newp8</td>
<td>100</td>
<td>64</td>
<td>3</td>
<td>33</td>
<td>SNT-Text</td>
</tr>
<tr>
<td>8</td>
<td>yp278a</td>
<td>100</td>
<td>64</td>
<td>2</td>
<td>34</td>
<td>SNT-Text</td>
</tr>
<tr>
<td>9</td>
<td>Stunner-2</td>
<td>100</td>
<td>63</td>
<td>6</td>
<td>31</td>
<td>SNT-Text</td>
</tr>
<tr>
<td>10</td>
<td>WOX Test 2</td>
<td>100</td>
<td>62</td>
<td>4</td>
<td>34</td>
<td>NET-WOX</td>
</tr>
<tr>
<td>11</td>
<td>MLP_Original-MoreNeurons.0.1-gen312-ties0.FF</td>
<td>100</td>
<td>60</td>
<td>4</td>
<td>36</td>
<td>MLP-Text</td>
</tr>
<tr>
<td>12</td>
<td>try3MLP_Original-MoreNeurons.0.1-gen341-ties0.FF</td>
<td>100</td>
<td>59</td>
<td>4</td>
<td>37</td>
<td>MLP-Text</td>
</tr>
<tr>
<td>13</td>
<td>shrd-MaxSolve-7c1kg</td>
<td>100</td>
<td>59</td>
<td>2</td>
<td>39</td>
<td>MLP-Text</td>
</tr>
<tr>
<td>14</td>
<td>test-mlp1</td>
<td>1000</td>
<td>582</td>
<td>34</td>
<td>384</td>
<td>unknown</td>
</tr>
</tbody>
</table>
Results versus CEC 2006 Champion (a manual EVO / TDL hybrid MLP)

<table>
<thead>
<tr>
<th>$n_{sp}$</th>
<th>Won</th>
<th>Drawn</th>
<th>Lost</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>89</td>
<td>5</td>
<td>106</td>
</tr>
<tr>
<td>500</td>
<td>135</td>
<td>6</td>
<td>59</td>
</tr>
<tr>
<td>750</td>
<td>142</td>
<td>5</td>
<td>53</td>
</tr>
<tr>
<td>1000</td>
<td>136</td>
<td>2</td>
<td>62</td>
</tr>
<tr>
<td>1250</td>
<td>142</td>
<td>5</td>
<td>53</td>
</tr>
</tbody>
</table>
N-Tuple Summary

• Stunning results compared to other game-learning architectures such as MLP
• How might this hold for other problems?
• How easy are N-Tuples to apply to other domains?
Ms Pac-Man

- Challenging Game
- Discrete but large search space
- Need to code inputs before applying to function approximator
Screen Capture Mode

- Allows us to run software agents original game
- But simulated copy (previous slide) is much faster, and good for training
Ms Pac-Man Input Coding

• See groups of 4 features below
• These are displayed for each possible successor node from the current node
  – Distance to nearest ghost
  – Distance to nearest edible ghost
  – Distance to nearest food pill
  – Distance to nearest power pill
Alternative Pac-Man Features (Pete Burrow)

- Used a smaller feature space
- Distance to nearest safe junction
- Distance to nearest pill
So far: Evolved MLP by far the best!
Results: MLP versus Interpolated Table

• Both used a 1+9 ES, run for 50 generations
• 10 games per fitness evaluation
• 10 complete runs of each architecture
• MLP had 5 hidden units
• Interpolated table had $3^4$ entries
• So far each had a mean best score of approx 3,700
• More work is needed to improve this
  — And to test transference to original game!
Summary

• All choices need careful investigation
  – Big impact on performance

• Function approximator
  – N-Tuples and interpolated tables: very promising
  – Table-based methods often learn much more reliably than MLPs (especially with TDL)
  – But: Evolved MLP better on Ms Pac-Man
    • Input features need more design effort...

• Learning algorithm
  – TDL is often better for large numbers of parameters
  – But TDL may perform poorly with MLPs
  – Evolution is easier to apply

• Some things work very well, though much more research needed

• This is good news!
IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES


www.ieee-cis.org/pubs/tciaig/