Experiences with machine learning in games: Some lessons learned

Joost Westra
Erik Kok
Frank Dignum
Evolutionary Neural Networks in First Person Shooters

Joost Westra
Overview

- Introduction
- Currently used approaches
- Neural Networks
- Evolutionary Algorithms
- Experiment
- Results
- Conclusion
- NEAT
Goal

- Show that learning techniques can be used in commercial computer games
- Make it easier to create complex behaviors and to improve the quality of these behaviors
First Person Shooter

- Good artificial behavior
- Fast, Real-time action
- Best humans still better
Quake III
Quake III

- First Person Shooter
- Player Vs Bots
- Computer Players equal to Human
- Ability to speed up
- Known for difficult Bots
- Full source Code
Quake III Bot
Decision Problems

- Part of the artificial behavior
- Different inputs
- No clear best decision
- Lots of different decisions problems
Current Techniques

• Hard coding
• Scripting
• Fuzzy Logic
• Finite State Machine
Hardcoding/scripting

- Lots of control
- Very labor intensive
- Predictable behavior
Finite State Machine

- Used on top of other solutions
Fuzzy Logic

```java
weight "holdable_teleporter"
{
    switch(INVENTORY_TELEPORTER)
    {
        case 1:
        {
            switch(INVENTORY_MEDKIT)
            {
                case 1: return 60;
                default: return 0;
            }
        }
        default: return 0;
    }
}
```
Fuzzy Logic
Fuzzy Logic

- More Control
- Reason about decisions

- Very time consuming
- Difficult to balance
- Limited Amount of Inputs
- Predictable behavior
Neural Networks

- Tables grow exponentially
- Function approximator
- Good at generalization
Neural Networks

- Single Layer Feedforward
- Multi Layer Feedforward
- Fully Recurrent Network
- Competitive Network
- Jordan Network
- Simple Recurrent Network
Feed Forward Neural Net
Artificial Neuron
Activation Function

- **Threshold**
- **Linear**
- **Gaussian**
- **Sigmoid**
Feed Forward Neural Net
Weapon Selection

- Select the best possible weapon
- All in one pass
- Check required

- Weapons Available
- Ammo
- Health
- Armor
Item Selection

- Select best item to pick up
- Bots know possible locations of Items
- Not IF the Items are there
- Separate network pass each Item

- Item type
- Distance
- Last Visited
- Dropped
General Structure of Evolutionary Algorithm

1. Initialise the Population
2. Test population and Calculate fitness
3. Finished?
   - Yes: Stop
   - No: Create new individuals by:
     - Selecting parents
     - Breeding children (crossover)
     - Mutating children
Experiment Setup

- Bots fight in one Arena
- Keep track of fitness
- Selection
- Crossover
- Mutation
- Reset fitness
- Continue process
Population Size

- Good population size is about 20-30
  - Depends on task!
- Just add 25 bots in one Arena?
- Large Arena
- 6 Evolving Bots + 6 original Bots
Measuring Improvements

- They all get better
- Only proportional score
- Add equal number of standard bots
  - Same configuration as learning Bots
- Compare best score to standard bots
  - Result between best and average
- Could help learning
Fitness Function

• The real score for that period
  • Penalize getting killed?
• Very long fighting period
  • Luck involved
  • 500 points (kills) of best Bot
Reinsertion

• Pure reinsertion
  • New population

• Uniform reinsertion
  • Replace a number of random selected parents

• Elitist reinsertion
  • Replace a number of the worst parents

• Fitness-based reinsertion
  • Extra addition
  • Create more new individuals
  • Only add the best
Selection & Reinsertion

• Truncation selection
  • Best 50%
• Best Parents are kept
• Other selection types
  • Tournament
  • Proportional/roulette wheel
Regular Recombination

Parents:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>crossover point</td>
<td></td>
</tr>
</tbody>
</table>

Children:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Parents:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>crossover points</td>
<td></td>
</tr>
</tbody>
</table>

Children:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Unversiteit Utrecht
Recombination Neural Networks

- Weights are real valued numbers
Discrete Recombination

• Real valued or not?

\[ \text{Var}_i^O = \text{Var}_i^{P1} \cdot a_i + \text{Var}_i^{P2} \cdot (1 - a_i) \quad i \in (1, 2, ..., N\text{var}) \]
Line Recombination

\[ V_{ar_i}^O = V_{ar_i}^{P1} \times a_i + V_{ar_i}^{P2} \times (1 - a_i) \quad i \in (1, 2, ..., N_{var}) \]
Intermediate Recombination

\[ \text{Var}_i^O = \text{Var}_i^{P1} \cdot a_i + \text{Var}_i^{P2} \cdot (1 - a_i) \quad i \in (1, 2, ..., N_{var}) \]
Used Recombination

- Two types
- Chance per weight
  - Average
- Line Recombination
- Different settings
  - Higher chance
  - Less Neurons
  - Extra mutation
Mutation

• Not too big compared to weights
  • Else parent information is useless
• Allow bigger jumps
  • Possible to jump out local maximum
Mutation
Results

• Multiple runs are done
• Behavior best visible in separate series
• Compared to original
Results: Weapon Only

Weapon only average

Graph showing the average performance of weapons.
Results: Weapon Only
Results: Goal Only
Results: Goal Only
Default

Default average

Universiteit Utrecht
Results: Line Recombination
Results: Line Recombination
Stronger Recombination

Extra recombination average

Universiteit Utrecht
Stronger Recombination

Extra recombination series
Results:
Average all experiments

Combined average results

Universiteit Utrecht
Conclusion

- Solve Decision problems automatically
- In a commercial game
- Outperforms original approach
Future Work

• Population size
• More decision problems
• Other game genres
• Extra inputs
• Generalization and specialization
• Behavior analysis
Related work

• Supervised Learning
• Reinforcement learning
• NEAT (Neural-Evolution of augmenting topologies)
NEAT

- Neural evolution of augmenting topologies
- Not only weights but also structure
- Individuals are laced in sub populations (species)
  - Species compete against each other
NEAT network representation
Neat Add connection

- Innovation number is 1 higher
- Weight is 1
NEAT add node

- Between an exciting connection
- Disable connection
- Create new node
- Create two new connections
- Innovation number is 2 higher
NEAT recombination

- **Line up**
  - Innovation numbers
- **Same number**
  - Random choice
- **Disjoint**
- **Excess**
- **Inherit from best**
  - Equal=random
- **Certain chance**
  - enable connection
rtNEAT
rtNEAT

- Only one individual is removed
  - Adjusted fitness
  - Fitness divided by number individuals of the species
- Choose parents
  - Proportional selection over species
Team Orders

• Capture the flag
• Three different roles
  • Aggressive
  • Defensive
  • (Roaming)
• Inputs state of the flags
  • Four possible inputs
• Five bots per team
  • Five possible outputs
Trying solutions

• Only one individual is used
• Fitness is score difference divided by time played
• Fitness of species is divided by # evaluations
rtNEAT Results (Dynamic vs Static)
rtNEAT Results (Dynamic vs Dynamic)
rtNEAT Results (Dynamic vs Pre-learned)
Adaptive reinforcement learning agents in RTS games

Eric Kok – Utrecht University
emkok@students.cs.uu.nl
www.ekok.nl/tech/bos22apl/
Introduction

- Challenges in playing RTS games
- Bos Wars as platform for game research
Introduction

• Problems with static scripts
  • Fixed
  • Repetitive
  • Predictable

• Project approach: reinforcement learning agents
  • More fun to play (non static opponents)
  • More challenging (can exploit the human weaknesses)
  • Less error-prone (can fix its own design-time flaws)
  • More natural design standpoint
Introduction

- Dynamic Scripting [Spronck et. al., 2003]
Agents in Bos Wars

- Games and agents
- Benefits of 2apl agents
- Coupling 2apl to Bos Wars
Agents in Bos Wars

- Example: a simple Bos Wars playing 2apl agent

Beliefs:
  gameCycle(0).

BeliefUpdates:
  \{ gameCycle(O) \} UpdateCycle(N) \{ not gameCycle(O), gameCycle(N) \}

Goals:
  attack(enemy)

PG-rules:
  attack(enemy) \<-
  true | \{ prepare(base); \}

PC-rules:
  prepare(base) \<-
  true | \{ @boswars(setUnit("unit-engineer", 4), R); @boswars(setUnit("unit-powerplant", 1), R); @boswars(setUnit("unit-magmapump", 2), R); @boswars(setUnit("unit-vfac", 1), R); @boswars(setUnit("unit-gturrent", 1), R); @boswars(waitUnit("unit-vfac"), R); launch(attack); \}
  launch(attack) \<-
  true | \{ @boswars(defineForce(0, "unit-tank,4;"), R); @boswars(waitForce(0), R); @boswars(attackWithForce(0), R); dropGoal(attack(enemy)); \}
  updateGameCycle(NewCycle) \<-
  true | \{ UpdateCycle(NewCycle); \}
  updateGameResult(Result, Fitness) \<-
  true | \{ @boswars(gameResultHandled(), R); \}
Learning winning strategies

- Applying reinforcement learning to 2ap1
  - Operating in an unknown environment
  - Action (PC-rule) selection over multiple weighted choices
  - Executing an (external) action returns a reward and a state
  - Update PC-rule weights to reflect obtained rewards

- Generic RL framework for Monte Carlo, TD, etc.

Learning winning strategies

- **Dynamic Scripting**
  - Tiny search space: Fast learning
  - Fixed script length
  - Rules may only be selected once
  - Fixed rule order

- **Monte Carlo control**
  - Classically has a large search space: Slower learning
  - No limitations on strategy
  - May use rule guards and a complex plan hierarchy
Learning winning strategies

• Example: a learning Bos Wars playing 2apl agent

Beliefs:
unit(unit-powerplant,0).
unit(unit-camp,0).
unit(unit-vfac,0).
unit(unit-assault,0).
unit(unit-tank,0).

Plans:
{ ^startEpisode(); ^setting("tau", 3); script(game); }

PC-rules:
^script(game) <- unit("unit-powerplant",X) and X < 1 |
{ ^visit();
   @boswars(setUnit("unit-powerplant", 1), R); }

^script(game) <- unit("unit-camp",X) and X < 1 |
{ ^visit();
   @boswars(setUnit("unit-camp", 1), R);
   @boswars(waitUnit("unit-camp"), R); }

^script(game) <- unit("unit-vfac",X) and X < 1 |
{ ^visit();
   @boswars(setUnit("unit-vfac", 1), R);
   @boswars(waitUnit("unit-vfac"), R); }

^script(game) <- unit("unit-camp",X) and X > 0 |
{ ^script(game) <- unit("unit-camp",X) and X > 0 |
   ^visit();
   @boswars(defineForce(0, "unit-assault,4;"), R);
   @boswars(waitForce(0), R);
   @boswars(attackWithForce(0), R); }

queueEmpty() <-
true | { script(game); }

updateOwnUnit(UnitType, UnitCount) <-
true | { UpdateOwnUnit(UnitType, UnitCount); }

updateGameResult(Result, Fitness) <-
true | {
   ^reward(Fitness);
   ^endEpisode();
   @boswars(gameResultHandled(), R); }
Learning winning strategies

- **Experimentation setup**
  - Ran against 5 different static and 5 different strategy switching scripts
  - Game length of maximum 200,000 cycles (normally end within 100,000 cycles)
  - TP: Turning Point at which 12 points in 7 consecutive games are gathered (6 wins 1 lose or 5 wins 2 draws)
  - Fail: When no TP is reached within 200 episodes
Learning winning strategies

- Basic learning performance
- Monte Carlo uses
  - PC rule guards
  - Softmax exploration
- DS performs slightly better

![Chart showing average turning points and fail rates for different scenarios.](chart_image)
Learning winning strategies

• Improving an agent’s learning performance
  • From flat, reactive agent to strategy hierarchy into PC-rules
    • Not effective in the short-paced strategy games of Bos Wars
    • When may this be more applicable?
  • MC-DS hybrid: use ‘rule selection order’ as game states
Adaptation to opponent tactics

- Opponent can switch strategies
  - Switch when previous game was lost
  - Switch every 2, 4 or 8 games
- Implicit adaptation through learning
  - Incorporating opponent data in the game state
- Explicit adaptation through strategy switching
  - Learn distinct strategies using expected game result tracking
Experimentation results

Average Turning Points

Fail Rates

- Dynamic Scripting
- MC hierarchy
- MC Softmax
- MC opponent data
- MC-DS Hybrid
- MC-DS Hybrid opponent data
Conclusions

• Agents are capable of learning winning strategies
  • More fun to play against
  • More challenging
  • Less error-prone
• MC can efficiently learn, needing less than 30 episodes on average
• Implicit adaptation with opponent data works best against strategy switching opponents
• MC-DS Hybrid may be used to replace complex game states, making DS redundant
References

• Literature