Robbe Demey

AI for 2-player, turn-based board games

Graduation work 2019-20

Digital Arts and Entertainment

Howest.be

Contents

[Abstract 1](#_Toc49275052)

[0. Introduction 3](#_Toc49275053)

[1. Representing board games 5](#_Toc49275054)

[1.1. The state 5](#_Toc49275055)

[1.2. Actions 6](#_Toc49275056)

[1.3. Consequences 6](#_Toc49275057)

[1.4. Symmetrical games 7](#_Toc49275058)

[2. Board game AI 9](#_Toc49275059)

[2.1. MiniMax 9](#_Toc49275060)

[2.2. NegaMax 10](#_Toc49275061)

[2.3. AlphaBeta 11](#_Toc49275062)

[2.4. AlphaBeta improvements 13](#_Toc49275063)

[2.4.1. Heuristic improvement 14](#_Toc49275064)

[2.4.2. Killer heuristic 14](#_Toc49275065)

[2.4.3. Minimal window 14](#_Toc49275066)

[2.4.4. Aspiration search 14](#_Toc49275067)

[2.4.5. Transposition Tables 14](#_Toc49275068)

[2.4.6. Refutation tables 15](#_Toc49275069)

[2.5. Iterative deepening 15](#_Toc49275070)

[2.6. State Space Search (SSS\*) 15](#_Toc49275071)

[2.7. Principal Variation Search and NegaScout 16](#_Toc49275072)

[2.8. Memory-enhanced Test Driver 17](#_Toc49275073)

[2.9. Best Node Search (BNS) 17](#_Toc49275074)

[2.10. Other algorithms 18](#_Toc49275075)

[3. Unity ML Agents 19](#_Toc49275076)

[3.1. Installation 19](#_Toc49275077)

[3.2. Agent 20](#_Toc49275078)

[3.3. Behavior parameters 20](#_Toc49275079)

[3.4. Training 22](#_Toc49275080)

[3.4.1. PPO vs SAC 22](#_Toc49275081)

[3.4.2. Training parameters 24](#_Toc49275082)

[3.4.3. Self-play 27](#_Toc49275083)

[3.5. Results 27](#_Toc49275084)

[3.5.1. Tensor Board 27](#_Toc49275085)

[3.6. Bugs 29](#_Toc49275086)

[3.6.1. Action mask issues 29](#_Toc49275087)

[3.6.2. OnEpisodeBegin issues 29](#_Toc49275088)

[4. The demo 31](#_Toc49275089)

[4.1. Framework 31](#_Toc49275090)

[4.2. Algorithmic agents 31](#_Toc49275091)

[4.3. Machine learning agents 32](#_Toc49275092)

[4.4. Trained agents 32](#_Toc49275093)

[4.5. Further research 34](#_Toc49275094)

[5. Conclusion 35](#_Toc49275095)

[6. References 37](#_Toc49275096)

Table of figures

[Figure 1 - Alternative checkers board 6](#_Toc49275097)

[Figure 2 - Symetrical tic-tac-toe boards 8](#_Toc49275098)

[Figure 3 - MiniMax tree 9](#_Toc49275099)

[Figure 4 - AlphaBeta tree 12](#_Toc49275100)

[Figure 5 - Behavior parameters 21](#_Toc49275101)

[Figure 6 - On-policy vs. Off-Policy Algorithms 23](#_Toc49275102)

[Figure 7- Tic-tac-toe game between a MiniMax and a BNS agent 31](#_Toc49275103)

[Figure 8 - PPO vs SAC, rewards over steps 33](#_Toc49275104)

[Figure 9 - Tournament, amount of wins 34](#_Toc49275105)

# Abstract

This paper is a practical study of board game AI. More specific, AI for simple grid-based board games with a limited board size (≤ 8x8), a limited number of players (≤ 2) and a limited amount of states for the squares (empty, player A, player B). It examines how to store and manipulate the board state and which AI can be used. With the paper comes a Unity project, a framework, that allows to quickly implement new board games and AI for those board games.

The first chapter goes in depth about how to represent a board games using bitmasks. The second chapter lists and describes the most important algorithms that can be used for board game AI. These algorithms are also implemented in the framework for reference. The third chapter goes in depth about Unity ML Agents and building and training a machine learning agent for board games. The fourth chapter tells more about the Unity framework that comes with this paper.

The framework provided with this paper makes it very easy to implement a board game that first the given requirements. There is an easy to use user interface that will create a board based on the game. The framework also gives the possibility to quickly create AI agents, either based on game tree searches or based on machine learning. Some agents are plug and play, while for others the user will need to write a derived class implementing a minimum of abstract classes. The machine learning agents will need a couple of days/weeks of training before they are somewhat reliable, decent players. A Twitch agent has also been added that uses the input of the Twitch chat to play.

When comparing the search algorithms with the machine learning solutions the conclusion is that search algorithms are easy and fast to implement with lots of references, they perform well as long as the amount of possible moves is limited and games do not take too many moves to end. When games do tend to have a lot of possible moves or are very “deep” everything comes down to a good heuristic function that gives a rating to a certain board position (this can also be machine learning based). The machine learning agent takes a long time to implement for a beginner, while for a “specialist” it will not take lot of programming time but the agent will still need a couple of days of training, after which chances are high the agent is still beatable.

The main conclusion is that the game developer will be able to create a decent AI faster using an old-school search algorithm than to try and build a machine learning solution. Search algorithms will also give more reliable results.

# Introduction

In this paper we try to answer the following question:

“What are the different AI possibilities for simple 2-player, turn-based board games?”

There are many 2-player, turn-based board games with very simple board states like Connect four, Checkers, Go, OXO, Othello, Binairo (single player), etc. In these games both player play in a -matrix where each element can have 3 states (empty, player one or player two). Since the games are so similar, creating an AI for these games is probably similar too.

These games have brute force AI solutions but those will probably take more computing power and/or memory. Other possible solution can be based on heuristics, those can be either well designed to create a strong AI or based on human intuitions to create a more human-like AI player. Another possibility are the machine learning solutions, like reinforcement learning where the AI learns from playing against itself.

This paper comes with a demo in Unity in which many of the algorithms described in this paper are implemented in C# for further reference. The demo also contains an example of a fully trained machine learning tic-tac-toe AI.

# Representing board games

Board games have a state, actions, and consequences. In these board games the entire state is defined by the board. For the games considered in the paper the only possible action is to pick a square on the board and change its state. The consequences are a number of rules that change the state depending on the actions chosen by the players, for example the win conditions.

## The state

The games have a board state which gets manipulated by the players until the game ends. The board consists of a matrix of cells which can have three possible states: empty, claimed by player one and claimed by player two.

The board will be represented by a “bit board”. The board state is saved in the bits of two ulong variables (64 bits, one for each player. A 0 bit means it is not claimed by the player and a 1 bit means it is claimed by the player. The 64-bit format limits the size of the matrix to 8x8.

Bit boards have some major advantages. Operations are efficient since they can check and manipulate multiple cells of the board at once and bit operations (see chapters Checking win conditions and Symmetric games). Memory wise bit boards take the minimum amount of space (Thill, 2008).

/// <summary>

/// Bits of the mask for a certain player

/// </summary>

public ulong bits;

Two bitmasks, one for the column and one for the row, are used to link the bits to the cells:

private static readonly ulong[] columnMasks = new ulong[8]

{

0b\_00000001\_00000001\_00000001\_00000001\_00000001\_00000001\_00000001\_00000001,

0b\_00000010\_00000010\_00000010\_00000010\_00000010\_00000010\_00000010\_00000010,

0b\_00000100\_00000100\_00000100\_00000100\_00000100\_00000100\_00000100\_00000100,

0b\_00001000\_00001000\_00001000\_00001000\_00001000\_00001000\_00001000\_00001000,

0b\_00010000\_00010000\_00010000\_00010000\_00010000\_00010000\_00010000\_00010000,

0b\_00100000\_00100000\_00100000\_00100000\_00100000\_00100000\_00100000\_00100000,

0b\_01000000\_01000000\_01000000\_01000000\_01000000\_01000000\_01000000\_01000000,

0b\_10000000\_10000000\_10000000\_10000000\_10000000\_10000000\_10000000\_10000000

};

private static readonly ulong[] rowMasks = new ulong[8]

{

0b\_00000000\_00000000\_00000000\_00000000\_00000000\_00000000\_00000000\_11111111,

0b\_00000000\_00000000\_00000000\_00000000\_00000000\_00000000\_11111111\_00000000,

0b\_00000000\_00000000\_00000000\_00000000\_00000000\_11111111\_00000000\_00000000,

0b\_00000000\_00000000\_00000000\_00000000\_11111111\_00000000\_00000000\_00000000,

0b\_00000000\_00000000\_00000000\_11111111\_00000000\_00000000\_00000000\_00000000,

0b\_00000000\_00000000\_11111111\_00000000\_00000000\_00000000\_00000000\_00000000,

0b\_00000000\_11111111\_00000000\_00000000\_00000000\_00000000\_00000000\_00000000,

0b\_11111111\_00000000\_00000000\_00000000\_00000000\_00000000\_00000000\_00000000

};

To check whether a certain player has claimed the cell at column x, row y:

public bool GetBit(int x, int y)

=> bits & columnMasks[x] & rowMasks[y] != 0;

As a little side note, for checkers the maximum board size is actually 9x9 since you can turn the board 45°, making the moves straight rather than diagonal, and remove the white squares ending up with a board as displayed below, where the black player can play downwards and to the right and the white player can play upwards and to the left. The 9x9 format is not popular at all though.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  | ● | ● |  |  |  |
|  |  | ● | ● | ● |  |  |  |
|  | ● | ● | ● |  |  |  |  |
| ● | ● | ● |  |  |  | ○ | ○ |
| ● | ● |  |  |  | ○ | ○ | ○ |
|  |  |  |  | ○ | ○ | ○ |  |
|  |  |  | ○ | ○ | ○ |  |  |
|  |  |  | ○ | ○ |  |  |  |

Figure 1 - Alternative checkers board

## Actions

As said before the actions for the games in the scope of this paper are limited to changing the state of a single square, this can be done using the following function:

public void SetBit(int x, int y, bool bit)

{

if (bit)

bits |= columnMasks[position.x] & rowMasks[position.y];

else

bits &= ~(columnMasks[position.x] & rowMasks[position.y]);

}

## Consequences

The consequences can be many things, for example winning the game when making four in a row in connect-four or capturing a piece in a game of checkers, but here we will take the simplest and most necessary example of the win-condition.

Since the board is constructed as a bit matrix, we can use bit masks to scan over the board state to check if a win condition is met. We will use tic-tac-toe or any three in a row game as an example. These games are won once three cells are connected in a line by one player, this can be done in four different ways, either horizontally, vertically, diagonally for going up or diagonally going down. These conditions can be represented by the following four bitmasks:

private readonly BitMask[] masks = new BitMask[4]

{

new BitMask(0b\_00000111, new Size(3, 1)), // -

new BitMask(0b\_00000001\_00000001\_00000001, new Size(1, 3)), // |

new BitMask(0b\_00000001\_00000010\_00000100, new Size(3, 3)), // /

new BitMask(0b\_00000100\_00000010\_00000001, new Size(3, 3)) // \

};

Now we can scan over the board looking for a three in a row for a certain player using:

public override bool GetIsWin()

{

for (int i = 0; i < masks.Length; ++i)

for (int x = 0 x <= board.size.x - masks[i].size.x; ++x)

for (int y = 0; <= board.size.y - masks[i].size.y; ++y)

{

ulong mask = masks[i].bits × columnMasks[x] & rowMasks[y];

if (bits & mask == mask)

return true;

}

return false;

}

This method is easy to mimic for other games since the only thing you have to do is update the list of masks. The GetIsWin() function works game independent. Of course, this is limited to games that have win conditions based on a certain local board state. For example, checkers ends when a player is out of pieces, this can be tested by:

public override bool GetIsWin()

=> opponent.bits == 0;

## Symmetrical games

A machine learning agent has no concept of symmetry and learns certain concepts multiple times which humans quickly would recognize as symmetrical. For example, the following tic-tac-toe situations are completely equivalent to each other:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | |  | ○ |  | | ○ |  |  | |  | ● | ● |  |  |  |  | | --- | --- | --- | |  | ● | ● | | ○ |  |  | |  | ○ |  |  |  |  |  | | --- | --- | --- | |  | ○ |  | | ● |  | ○ | | ● |  |  |  |  |  |  | | --- | --- | --- | | ● |  |  | | ● |  | ○ | |  | ○ |  | | |  |  |  | | --- | --- | --- | |  | ○ |  | |  |  | ○ | | ● | ● |  |  |  |  |  | | --- | --- | --- | | ● | ● |  | |  |  | ○ | |  | ○ |  |  |  |  |  | | --- | --- | --- | |  | ○ |  | | ○ |  | ● | |  |  | ● |  |  |  |  | | --- | --- | --- | |  |  | ● | | ○ |  | ● | |  | ○ |  | |

Figure 2 - Symmetrical tic-tac-toe boards

We can help the agent by manipulating the matrix so that all equivalent problems are given as one. For this we use the following functions below, using more bit operations. These functions are independent of the board size and thus can also be used for other games than tic-tac-toe.

public void MirrorHorizontally() // |

{

for (int x = 0; x < size.x / 2; ++x)

{

ulong column0 = bits & columnMasks[x];

ulong column1 = bits & columnMasks[size.x - x - 1];

bits &= ~(column0 | column1);

column0 <<= (size.x - 2 \* x - 1);

column1 >>= (size.x - 2 \* x - 1);

bits |= column0 | column1;

}

}

public void MirrorVertically() // -

{

for (int y = 0; y < size.y / 2; ++y)

{

ulong row0 = bits & rowMasks[y];

ulong row1 = bits & rowMasks[size.y - y - 1];

bits &= ~(row0 | row1);

row0 <<= 8 \* (size.y - 2 \* y - 1);

row1 >>= 8 \* (size.y - 2 \* y - 1);

bits |= row0 | row1;

}

}

public void MirrorDiagonally() // /

{

for (int x = 1; x < (size.x < size.y ? size.x : size.y); ++x)

for (int y = 0; y < x; ++y)

{

bool xy = GetBit(x, y);

bool yx = GetBit(y, x);

SetBit(x, y, yx);

SetBit(y, x, xy);

}

}

# Board game AI

Now that we have that game setup, we need an adversary for single player games. The MiniMax-algorithm lends itself for this kind of games. Many researches have suggested improvements on this algorithm. In this chapter we go in depth in the most well know variations on MiniMax. The algorithmic function described below all have a working version in the Unity-project. In this paper the same functionality is given in pseudo code for easier understanding.

All these algorithms need either a lot of time or good heuristic function (function that estimates a score for a certain board state) on games with a decent number of moves (both in depth as in possibilities). Writing a heuristic function needs a good understanding of the game at hand. To avoid this problem a machine learning solution might be a better option, this way the computer develops an understanding for the game and the developer does not need to think about a good heuristic function. In the next chapter we look at a machine learning solution using Unity ML Agents.

## MiniMax

MiniMax is a simple algorithm that searches through the possible moves, picking the move with which leads to the best board position (MAX option in the image below) on the turn of the active player and picking the move with the worst board position (MIN option in the image below) for the active player when it is the opponents turn. The algorithm can also keep track of the number of moves played, deciding to return a heuristic value after a certain amount of moves instead of keep searching for an end state (which can take a lot of time).

|  |  |
| --- | --- |
| Active player  Opponent  Active player  Opponent |  |

Figure 3 - MiniMax tree (Heineman et al., 2016, pp. 176)

#### Algorithm

The MiniMax-algorithm takes in a state (node), the amount of recursion steps it is allowed to do (depth) and whether it needs to maximize or minimize (maximizingPlayer). It first checks whether is maximum depth is reached or if and end state is reached and if so, it returns the value for the state. Then it checks whether the maximizing player is making the next move, if that is the case it will call the minimax function again for each possible new state and pick the highest and return the highest scoring state, else it will pick the lowest scoring state. Following is the MiniMax-algorithm in pseudocode, starting with the original call (Heineman et al., 2016, pp. 174–180).

minimax(origin, depth, TRUE)

 function minimax(node, depth, maximizingPlayer) is

if depth = 0 or node is a terminal node then

return the heuristic value for the maximizing player of node

if maximizingPlayer then

value := −∞

for each child of node do

value := max(value, minimax(child, depth − 1, FALSE))

return value

else (\* minimizing player \*)

value := +∞

for each child of node do

value := min(value, minimax(child, depth − 1, TRUE))

return value

## NegaMax

NegaMax is a simplified version of MiniMax, using some clever tricks.

First, we replace the maximizingPlayer variable with a variable f (float) that either holds -1 (minimizing player) or +1 (maximizing player), initialized with the value +1. We can now use f to remove the if statement, knowing that .

 function score (node, depth, f) is

if depth = 0 or node is a terminal node then

return the heuristic value of node for the maximizing player

value := f × -∞

for each child of node do

value := f × max(f × value, f × score(child, depth − 1, -f))

return value

The examined games are zero-sum, meaning that the score of both players always add up to zero. Either one player wins (+1) and one player loses (-1) adding up to zero, or the players draw (+0 for both players) adding up to zero. This characteristic can be represented as . Using this characteristic we can rewrite as .

 function score (node, depth, f) is

if depth = 0 or node is a terminal node then

return the heuristic value of node for the maximizing player

value := f × -∞

for each child of node do

value := f × max(f × value, -score(child, depth − 1, +1))

return value

Since the score function is now always called with +1 for the f parameter (both in the initial call as in the recursive calls) we can just remove the f parameter.

 function score (node, depth) is

if depth = 0 or node is a terminal node then

return the heuristic value of node for the maximizing player

value := -∞

for each child of node do

value := max(value, -score(child, depth − 1))

return value

We can add a color parameter that holds the color of the maximizing player (represented by -1 or +1) if we also make the heuristic value dependent on who is making the last move. After this step we have the NegaMax algorithm as it is best known (Heineman et al., 2016, pp. 180-183).

(\* Initial call for Player A's root node \*)

negamax(rootNode, depth, +1)

(\* Initial call for Player B's root node \*)

negamax(rootNode, depth, −1)

function negamax(node, depth, color) is

if depth = 0 or node is a terminal node then

return the heuristic value of node for the “color” player

value := −∞

for each child of node do

value := max(value, −negamax(child, depth − 1, −color))

return value

## AlphaBeta

The AlphaBeta-algorithm (or AlphaBeta pruning) is an improvement for both MiniMax and NegaMax. It uses the following intuition for a human player (Gonzalez et al., 2014, pp. 38-10):

If we are searching through the possible moves, keeping track of the best scoring move (B), it takes just one good counter (C) move that reduces the score of the possible move (A) below that of the best scoring move (B) to disband the possible move (A). It is not useful to check for other counter moves to the possible move (A).

AlphaBeta will prune unproductive branches from the search tree. The following is the same tree as shown in the MiniMax chapter but using NegaMax and AlphaBeta pruning. The active player will pick the worst scoring move between move 1 and move 2 since the score is calculated for the opponent. In the branch at move 1 the opponent cannot reduce the score to less than -3. When checking move 2, it becomes clear that the active player’s counter move 3 leads to a score of -2 for the opponent. Since whatever the score for move 4 is, the score for the opponent will never be smaller than -2, and the active player will always pick the lower scoring move 1 over move 2, since -2 is already smaller than -3 it is not useful to search for the score of move 4. AlphaBeta will select the same moves as MiniMax and NegaMax but with better performance (Heineman et al., 2016, pp. 183-189).

|  |  |
| --- | --- |
| Active player  Opponent  Active player  Opponent |  |

Figure 4 - AlphaBeta tree (Heineman et al., 2016, pp. 184)

#### Algorithm

The algorithm uses two values, alpha and beta. Alpha is the minimum score of the maximizing player and beta is the maximum score of the minimizing player. Alpha is initialized at negative infinity and beta is initialized at positive infinity. When the beta value becomes smaller than the alpha value the other counter moves to the examined move can be dismissed as it is not possible that they will end up in the game (Heineman et al., 2016, pp. 183, 189). The algorithm is implemented in pseudocode for both MiniMax and NegaMax below:

#### MiniMax

Following is the pseudocode for the AlphaBeta implementation for MiniMax, starting with the initial call (Alpha-beta pruning, 2020).

alphabeta(origin, depth, −∞, +∞, TRUE)

function alphabeta(node, depth, α, β, maximizingPlayer) is

if depth = 0 or node is a terminal node then

return the heuristic value of node

if maximizingPlayer then

value := −∞

for each child of node do

value := max(value, alphabeta(child, depth − 1, α, β, FALSE))

α := max(α, value)

if α ≥ β then

break (\* β cut-off \*)

return value

else

value := +∞

for each child of node do

value := min(value, alphabeta(child, depth − 1, α, β, TRUE))

β := min(β, value)

if β ≤ α then

break (\* α cut-off \*)

return value

#### NegaMax

Following is the pseudocode for the AlphaBeta implementation for NegaMax, starting with the initial call (NegaMax, 2020).

(\* Initial call for Player A's root node \*)

negamax(rootNode, depth, −∞, +∞, 1)

function negamax(node, depth, α, β, color) is

if depth = 0 or node is a terminal node then

return color × the heuristic value of node

childNodes := generateMoves(node)

childNodes := orderMoves(childNodes)

value := −∞

for each child in childNodes do

value := max(value, −negamax(child, depth − 1, −β, −α, −color))

α := max(α, value)

if α ≥ β then

break (\* cut-off \*)

return value

## AlphaBeta improvements

There are a lot of small improvements on the AlphaBeta-algorithm, these are listed below. Some other improvements like PVS, NegaScout, MTDf(n) and BNS give a bigger twist on the idea of AlphaBeta and thus are considered as separate algorithms. Most of the algorithms below try to increase the number of cut-offs. Combinations of these ideas are also possible for further increase the efficiency of the algorithm.

### Heuristic improvement

The heuristic improvement will use a heuristic sort on the possible moves, putting those moves that have the highest potential to cause cut-offs in front of the list. This heuristic sort function is dependent on the game (Tam, 2019). Since the sort function is called for every visited move, an expensive sorting function will cause a big overhead.

### Killer heuristic

Alpha-beta pruning is most efficient when first moves are considered are those moves most likely to produce a cut-off. The killer heuristic is taking advantage of the fact that moves that cause a cut-off in on branch are more likely to also cause a cut-off in another branch. When a cut-off happens the move that caused it is saved. When checking a new set of possible moves that contains the saved move, the saved move is checked first, hoping to cause an early cut-off.

In practice often two killer moves are considered, checking whether either one of these moves results in a cut-off before considering other possible moves. A non-killer move will replace the longest-standing killer move if this non-killer move produces a cut-off (Killer Heuristic, 2019; Schaeffer, 1989, pp. 6, 7; Tam, 2019).

### Minimal window

Minimal window is an improvement based on the idea that the first examined move is better than the remaining moves until proven otherwise. The first move is searched within a full window ([α, β]) returning a value v, while the remaining moves are searched within a smaller window ([v, v + 1]) around the value v. A smaller window leads to more cut-offs. If a remaining move cannot be proven to be inferior a re-search needs to happen in the range [v + 1, β]. The re-search is expensive but minimal window works on the gamble that only a small number of re-searches will happen (Schaeffer, 1989, pp. 5, 6).

### Aspiration search

A normal alpha beta search start with a search window ranging from −∞ to +∞. An aspiration search will use a smaller window based on a value guess. A smaller window will lead to more alpha beta cut-offs and more cut-offs lead to a higher efficient. But when the value is not within the window then an expensive re-search needs to happen with a wider window (Schaeffer, 1989, p. 6).

### Transposition Tables

In most games it is possible to reach a certain board state in multiple ways. In a normal AlphaBeta algorithm this board state would be visited multiple times, one time for each possible way to get to the board state. A transposition table will save the visited board states and their value. When the search algorithm arrives at a certain board state that is already in the transposition table, the value in the transposition can be used instead of re-evaluating the board state. The information in the transposition table can also be used to determine the order in which the moves are searched in a way that causes more AlphaBeta cut-offs (see 2.4.1.).

If the searches are limited to a certain depth it is also important to save the depth at which the value was determined in the transposition table. If the transposition table is used at a deeper iteration the value should be recalculated for that depth.

Most transposition tables will overwrite non-critical moves to save space, this will not affect the result but recalculating certain moves might be necessary

(NegaMax, 2020; Schaeffer, 1989, pp. 4, 5).

### Refutation tables

Transpositions tables can get big in size having a big memory load. Refutation tables try to keep the advantages of transposition tables while using less memory. The refutation table will hold branches that lead to alpha beta cut-offs in other branches, in the next iterations these branches will be tried first (adding an extra layer of depth) leading to faster cut-offs (Schaeffer, 1989, p. 5).

## Iterative deepening

The iterative deepening algorithm is a variation on the AlphaBeta algorithm that will first estimate a score (by doing an AlphaBeta call with depth 1) for all possible moves before looking deeper into the moves. This resembles the heuristic improvement in that it also orders the moves by probability of causing AlphaBeta cut-offs. With each iteration step more and more confidence is gain that the chosen move is the best move. The algorithm has to do extra AlphaBeta calls (one for each possible move at each iteration) but this is compensated for by the AlphaBeta cut-offs. The main advantage of this algorithm is that it has a good guess for the best move at any time, this means the algorithm can also perform well with time constrains (Schaeffer, 1989, p. 4).

## State Space Search (SSS\*)

The State Space Search algorithm is similar to the better-known A\* search algorithm. It is a best-first search algorithm based on the notion of solution trees. A solution tree is formed by pruning the number of branches for each move made by the active player (every two iterations). Then a best-first search is done on the solution tree. This will result in the same move MiniMax would propose.

SSS\* can produce more cut-offs than the AlphaBeta-algorithm and will never produce less cut-offs. Experiments by Plat et al. (1996, pp. 14, 17) showed that the extra cut-off do not compensate for the extra overhead (compared to the AlphaBeta-algorithm) when smaller transposition tables are used, when bigger transposition tables are used SSS\* outperforms AlphaBeta(Gonzalez et al., 2014, pp. 38-12, 38-13; SSS\*, 2019).

## Principal Variation Search and NegaScout

Principal variation search and NegaScout are two equivalent algorithms although they are described differently. They outperform AlphaBeta because they will at least cut off all the branches AlphaBeta would cut off. A good order heuristic (see 2.4.1.) is important though to make the algorithm performant.

The algorithm assumes the first explored move is the best move (or the principal variation) and it starts doing null-window (or scout window) search to increase the number of cut-offs. If the search fails (no move is found within the range of the window) a normal AlphaBeta search is done. If the move ordering is bad, the algorithm will need to do a lot of re-searches leading to a worse performance than normal AlphaBeta.

#### Principal Variation Search

Following is the Principal variation search algorithm in pseudocode (Principal variation search, 2019).

function paves(node, depth, α, β, color) is

if depth = 0 or node is a terminal node then

return color × the heuristic value of node

for each child of node do

if child is first child then

score := −pvs(child, depth − 1, −β, −α, −color)

else

score := −pvs(child, depth − 1, −α − 1, −α, −color

if α < score < β then

score := −pvs(child, depth − 1, −β, −score, −color)

α := max(α, score)

if α ≥ β then

break (\* beta cut-off \*)

return α

#### NegaScout

Following is the NegaScout algorithm in pseudocode (Plaat, 1997).

function negaScout ( node, depth, α, β, color )

if depth = 0 or node is a terminal node then

return color × the heuristic value of node

b := β;

for each child of node do

score := -negaScout ( child, -b, -α )

if (a < t) && (t < β) && (1 < i)

score := -negaScout ( child, -β, -α )

α := max(α, score)

if β <= α

return α

b := α + 1

return α;

## Memory-enhanced Test Driver

Reformulating the best-first property of SSS\* as a sequence of depth-first call lead to the formulation of a class of null-window alpha-beta algorithms like MTD-f. Just like in PVS and NegaScout it is the null-window searches that lead to greater efficiency. MTD-f uses AlphaBeta calls with transposition tables to further increase its performance. Tests have shown MTD-f to be more performant than NegaScout for games like chess, checkers, and Othello.

Following is the pseudocode for the MTD(f) algorithm (Gonzalez et al., 2014, pp. 38-13).

function mtdf(root, valueGuess, depth) is

g := valueGuess

upperBound := +∞

lowerBound := −∞

while lowerBound < upperBound do

β := max(g, lowerBound + 1)

g := AlphaBetaWithMemory(root, β − 1, β, depth)

if g < β then

upperBound := g

else

lowerBound := g

return g

## Best Node Search (BNS)

BNS uses a separation value produced by the NextGuess(α, β, subtreeCount)-function that gives an estimate for the value of the best move. The algorithm then does an AlphaBeta search with a zero window ([β -1, β]) with a transposition table for each possible move, and then compares the return value with the separation value. If the all the AlphaBeta searches returned a worse value then a recalculation happens with the separation value as beta value, else the separation value becomes the new alpha value. The best move is found when exactly one AlphaBeta search returned a bigger value or when the alpha-beta range is reduced to one (which means there are multiple moves with the best value). The algorithm can be further optimized by creating a customized NextGuess(…) function that uses statistical information about the move values.

Following is the pseudo code for the BNS algorithm (Rutko, 2011, pp. 94-99).

function nextGuess(α, β, subtreeCount) is

return α + (β − α) × (subtreeCount − 1) / subtreeCount

function bns(node, α, β) is

subtreeCount := number of children of node

do

test := nextGuess(α, β, subtreeCount)

betterCount := 0

for each child of node do

bestVal := −alphabeta(child, −test, −(test − 1))

if bestVal ≥ test then

betterCount := betterCount + 1

bestNode := child

if betterCount = 0

beta := test

else

subtreeCount = betterCount

alpha := test

while not (β − α < 2 or betterCount = 1)

return bestNode

## Other algorithms

Rutko (2012) also describes some more algorithms, problems and solution concerning game trees searches, this doctoral thesis can is a must read for those who want to go deeper into brute force algorithms. The Monte-Carlo tree search, LAlphaBeta and FAlphaBeta (among many) are some other interesting algorithms that are not described in Rutko (2012).

# Unity ML Agents

To make the framework even more adaptable it would be nice if you could just plug in any game, add the rules, and get a good AI without needing to code heuristics or algorithms. Machine Learning might make this possible.

Unity ML Agents seemed to fit this project, as Dr. Danny Lange (VP of AI and Machine Learning at Unity Technologies) puts it: “Unity ML-Agents offers a flexible way to develop and test new AI algorithms quickly and efficiently across a new generation of robotics, games, and beyond”. The ML Agents framework allows to train agents using reinforcement learning, imitation learning, neural evolution, or other methods using the Python API.

The main advantage is that the Unity community is very active so in normal circumstances a lot of resources and examples can be found on the internet. But during this project it became clear it is not really the case for ML Agents. Although ML Agents appeared in 2017 it is still in beta and it thus exists for three years now, the framework is still buggy, and the documentation is far from on point.

## Installation

When installing Unity ML Agents and the necessary software always look for the latest, up-to-date documentation, the internet is filled with old documentation. Download Unity, the ML Agents package and Python, be sure to check which versions are needed first. Unity’s ML Agents: Installation (2020a) is best used as installation guide, missing in there is how to set up a virtual environment which might be necessary to circumvent certain problems.

When having trouble going through the installation process a virtual environment can be a solution. Some guides will advise to use Anaconda, but this is not necessary. To create a Python virtual environment first create a folder for your virtual environments (here C:\python-envswas used). Then go to the command line and drop python -m venv C:\python-envs\mlagents-env. This line will create the virtual environment. We can activate the virtual environment by dropping C:\python-envs\mlagents-env\Scripts\activate in the command line, to deactivate the environment use deactivate*.* Once the virtual environment is activated, we have to install ML Agents (again) for Python using pip3 install mlagents.

There is also GPU support for the python API but due to multiple issues with versions that did not match up the installation was not successful. Although GPU support makes the training faster, it might be not worth knowing that you can end up concluding that it will never work after 4 days of trying to set up the GPU support. Always check which versions to use for all the software (Unity, Python, CUDA, …), because using the latest versions will sure end up in a failure.

## Agent

To create an agent in ML Agents an instance of the abstract Agent class has to be made (include Unity.MLAgents). There are three essential function to overwrite:

* void CollectObservations(VectorSensor sensor)
* void CollectDiscreteActionMasks(DiscreteActionMasker discreteActionMasker)
* void OnActionReceived(float[] vectorAction)

CollectObservations(VectorSensor sensor) is used to collect information for the environment. Adding an observation is done by updating the sensor parameter by using sensor.AddObservation(T observation). The passed observation can have one of the following types: bool, int, float, IEnumerable<float, Vector2, Vector3, Quaternion.

CollectDiscreteActionMasks(DiscreteActionMasker discreteActionMasker) can be used to limit the agent from picking certain actions. To actually set the action mask discreteActionMasker .SetMask(int branch, IEnumerable<int> actionIndices) needs to be used in the implementation of this function.

OnActionReceived(float[] vectorAction) used to translate the agent’s decision in an actual action in the environment. The vectorAction holds all the decisions made by the agents, these can either be the index of a discrete action or a continuous action value.

There is one more essential function that needs to be called: void SetReward(float reward). This function should be called at the end of the episode to give the agent the reward on which it can update its neural network when training. For these board games the rewards are +1 for a win, +0 for a draw and -1 for a loss. An extra correction is made (for tic-tact-toe and connect-four) for the length of the game, since the faster the player wins the better, and if a player can delay its defeat it should try to do so. More corrections can be made based on the board game, making a good reward function is essential for efficient training.

The Agent class has many other abstract functions, it is best to read as much available documentation as possible, but with these functions are the bare minimum to get an agent working.

## Behavior parameters

When adding an agent script to a game object a Behavior Parameters component will appear as seen in the image below. Following the image is a short description for each behavior parameter.

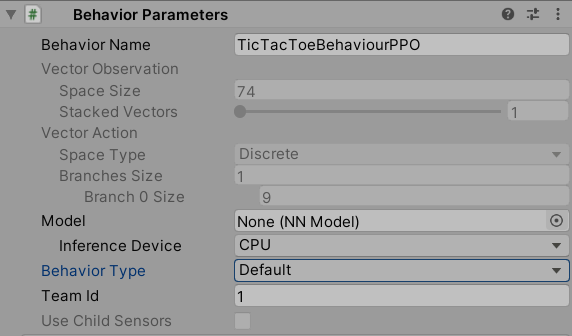


Figure 5 - Behavior parameters

#### Behavior name

The name of the behavior, this links up to a behavior in the .yaml file. If the behavior name does not link up to a behavior in the .yaml file, the agent will not train.

#### Vector observation

This parameter defines the structure of the observations. The space size is the number of observed variables per observation. The stacked vectors parameter is the amount of old vector observations that are used as inputs for the neural network. For example, a stacked vectors parameter of 2 will feed both the current observations and the previous observations to the network.

#### Vector action

This parameter defines the action space. This action space can either be discrete (a fixed amount of actions) or continuous (an infinite amount of actions based on a float input value). The branches size sub parameter represents the amount of actions being taken while the branch size represents the amount of possible actions for each action being taken.

#### Model

This parameter holds a reference to a trained neural network, using the network to determine the float[] actionVector in the Agent.OnActionRecieved(float[] actionVector ) function.

#### Behavior type

The behavior type enum has three possible states: default, heuristic only and interference only. The default value is used for training, heuristic only will make the agent fall back on the Agent.Heuristic(float[] actionsOut) function (which should be defined in the agent’s script) and the interference only will use the neural network without training it.

#### Team ID

This variable represents the team of the agent. Agents that work together have the same team ID. The team ID should be equal or greater than zero, otherwise the agent will not train in some cases.

#### Use child sensors

This parameter is there when using ray perception sensors or render texture sensors. If the parameter is toggled to true, the agent will look for sensors in the components of its child transforms.

## Training

Training the agent is takes a lot of time and thus optimizing training is important. The first big choice that needs to be made is which training algorithm to use. ML Agent gives the choice between Proximal Policy Optimization (PPO) and the Soft Actor Critic policy (SAC). The next step is to tune the training parameters, this step is mainly guessing work and trial and error. The final step is to analyze the results in TensorBoard.

### PPO vs SAC

While PPO is the default training algorithm in Unity ML Agents. SAC has been added more recently and has a higher sample-efficiency which means less games are necessary to create a good policy. We will not go in depth into how these algorithms work since this paper is more a practical guide than a theoretical paper.

Training algorithms are either on-policy or off-policy. On-policy algorithms collect a number of samples and learns how to improve based on those samples increasing the probability. PPO is such an algorithm. Off-policy algorithms tries to learn the optimal relationship between taking a certain action given a set of observations and the resulting reward. Once the optimal relationship is found, training the network can be done really fast and efficient. This is a harder learning problem but because the algorithm is learning a global function, all the collected samples can be reused for training, leading to a higher sample-efficiency (Teng, 2020).

The following image shows how the on-policy and the off-policy algorithms differ from each other. Notice how the off-policy algorithm uses a replay buffer to store and re-use old observations.

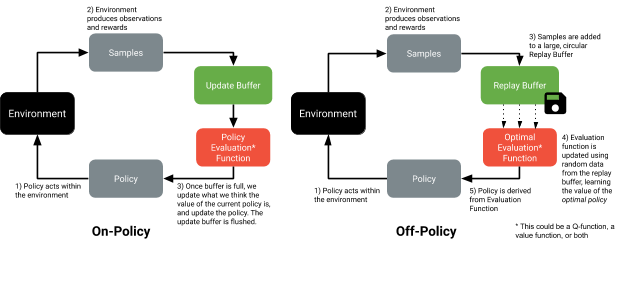


Figure 6 - On-policy vs. Off-Policy Algorithms (Teng, 2020)

You can determine the algorithm in the .yaml file. Each algorithm comes with some specific hyperparameters (more information in 3.3.2).

Following is a typical yaml file for a PPO setup and setup on the next page a typical SAC.

behaviors:

BehaviorPPO:

trainer\_type: ppo

hyperparameters:

batch\_size: 1024

buffer\_size: 10240

learning\_rate: 3.0e-4

learning\_rate\_schedule: linear

beta: 5.0e-3

epsilon: 0.2

lambd: 0.95

num\_epoch: 3

network\_settings:

vis\_encoder\_type: simple

normalize: false

hidden\_units: 128

num\_layers: 2

memory:

sequence\_length: 64

memory\_size: 256

max\_steps: 5.0e5

time\_horizon: 64

summary\_freq: 10000

keep\_checkpoints: 5

checkpoint\_interval: 50000

threaded: true

init\_path: null

behaviors:

BehaviorSAC:

trainer\_type: sac

hyperparameters:

batch\_size: 1024

buffer\_size: 10240

learning\_rate: 3.0e-4

learning\_rate\_schedule: linear

buffer\_init\_steps: 0

tau: 0.005

steps\_per\_update: 10.0

save\_replay\_buffer: false

init\_entcoef: 0.5

reward\_signal\_steps\_per\_update: 10.0

network\_settings:

vis\_encoder\_type: simple

normalize: false

hidden\_units: 128

num\_layers: 2

memory:

sequence\_length: 64

memory\_size: 256

max\_steps: 5.0e5

time\_horizon: 64

summary\_freq: 10000

keep\_checkpoints: 5

checkpoint\_interval: 50000

threaded: true

init\_path: null

### Training parameters

When training an agent, you must set the training parameters in the .yaml file. These parameters determine both the agent’s neural network as how it is trained. The .yaml file is used by the Python environment to do the training. Following are the training parameters for the PPO setup with a short description of how to use them (Mattar et al., 2018).

#### Gamma

The gamma value (gamma) is the discount factor for future rewards. If the agent is more interested in future rewards this value should be bigger, if the agent is more interested in short term rewards the value should be smaller. Typical values are ranging from 0.8 to 0.955.

#### Lambda

The lambda value (lambd) is used when calculating the General Advantage Estimate (GAE). The value determines how much weight the agent gives to the current value estimate when updating based on new training data. Low values lead to giving more importance on the current value (which can lead to a high bias) while high values lead to giving more importance to the new data (which can lead to a high variance). It is important to find a good balance, common values range from 0.9 to 0.95.

#### Buffer Size

The buffer size (buffer\_size) is the amount of experiences (observations, actions, and rewards) that should be obtained before updating the model. The value should be a multiple of the batch size (see below). Typical values range between 2048 to 409600.

#### Batch Size

The batch size (batch\_size) is the amount of experiences (observations, actions, and rewards) used for one iteration of gradient decent update. This should be a factor of the buffer size. For agents with a continuous action space this should be a large value (typically between 512 and 5120), while for agents with a discrete action space thee value should be lower (typically between 32 and 512).

#### Number of Epochs

The number of epochs (num\_epoch) represent the amount of experience buffer passes during gradient descent. A larger batch size makes it acceptable to increate this number. The higher the number the faster the agent will learn, but at the cost of less stable updates. This number typically ranges between 3 and 10.

#### Learning Rate

The learning rate (learning\_rate) is the strength of the gradient descent update step. If the training is unstable or if the reward does not consistently increase this value should be decreased. The learning rate typically ranges from 10-5 to 10-3.

#### Time Horizon

The time horizon (time\_horizon) defines to the amount of experience steps that need to be collected per agent before sending them to the experience buffer. If the end of an episode is reached before the end of an episode an expected reward will be used based on a value estimate of the agent’s current stage. The time episode should at least be big enough to capture the important behavior of a sequence of an agent’s actions. For our games this corresponds to the length of a single game.

#### Max Steps

Max steps (max\_steps) represents the number of steps the training process will take. More complex problems require more steps for the agent to get fully trained. Typically, this value ranges from 5 \* 105 to 107.

#### Beta

The beta value (beta) links to the entropy regularization which determines the randomness of the policy. The randomness makes sure that all possible actions are explored during training. Increasing beta will add more random actions. The entropy of the training (see TensorBoard) should decrease slowly over the training, if it drops to fast, increase beta, if it drops to slow, decrease beta. Values commonly range between 10-4 and 10-2.

#### Epsilon

Epsilon represents the acceptable threshold of divergence between old and new policies when updating using gradient descent. The smaller the value the more stable the updates but the slower the training. Typical values range from 0.1 to 0.3.

#### Normalize

Normalize is a bool parameter that represents whether normalization should be applied to observation inputs. The normalization is based on the running average and the variance of the vector observation. Normalization might be harmful with simple discrete control problems but might be helpful when the problem is more complex and continuous.

#### Number of Layers

The number of layers (num\_layers) defines the depth of the neural network. The more complex the problem, the more layer might be required. More layers require more training time. The number of layers typically ranges from 1 to 3

#### Hidden Units

The number of hidden units (hidden\_units) defines the width of the neural network, the number of neurons in each layer. The more interaction there is between the observation variables the more hidden units are required. A typical neural network has between 32 and 512 hidden units.

#### Recurrent Neural Network Hyperparameters

These hyperparameters are optional and are only used when use\_recurrent is set to true. The parameters are the sequence length (sequence\_lenght) and the memory size (memory\_size). The sequence length is the length of the experience sequence that is passed to the network for training. The length should be big enough to contain all important information from the beginning of the episode until the end of the episode. The memory size is the size of the float array that holds the hidden state of the recurrent neural network (RNN). The value of the memory should always be a multiple of 4.

#### Intrinsic Curiosity Module Hyperparameters

Only when use\_curiosity is set to true these hyperparameters are used. The curiosity encoding size (curiosity\_enc\_size) represents the size of the hidden layer used for encoding the observations within the curiosity module. A typical value for the encoding size lays between 64 and 256. The curiosity strength (curiosity\_strength) represents the magnitude of the generated intrinsic reward. The value should be balanced to the extrinsic reward signals so that neither of them overwhelms the other. Typical values are between 0.1 and 0.001.

### Self-play

When you train two adversarial agents using the same behavior Unity will automatically use the self-play feature (given that the self-play parameters are added to the behavior parameters and the agents have a different team id). One of the agents will then face off against a past version of itself, which allows the behavior to gradually improve (Cohen, 2020).

Because the behavior is his own adversary, the value estimate (see 3.4.1), which is basically the average reward, will always hover around 0. This it cannot be used as an indication of progress, therefor Unity added the Elo indicator. This will keep track of how good the agent performs against itself. In practice this indicator seems to fail too, showing a dropping Elo rate while the agent is actually improving.

The following behavior parameters should be added to the .yaml file. This is possible in both PPO and SAC behaviors. There is no documentation whatsoever on what these parameters actually exactly mean, which is one of the flaws of ML Agents at its current state.

self\_play:

window: 10

play\_against\_latest\_model\_ratio: 0.5

save\_steps: 50000

swap\_steps: 2000

team\_change: 100000

## Results

When the maximum number of steps is reached the training is done and ML Agents will create a .nn file for each agent. This file contains the trained neural network. To test the neural network, you drag the file into the model parameter of the agent. If you think the agents needs more training, you can continue training the network by increasing the maximum number of steps in the .yaml file and just resume the training (using the –resume command). If you want more information about what happened during the training process you can use Tensor Board.

### Tensor Board

To open Tensor Board after the training ended, drop the following line in the command prompt:

tensorboard --logdir=results

This will start a host a local webpage which can be opened in any browser by copy-pasting “localhost:6006” in the address bar. On this webpage you will find graphs of how certain parameters evolved during training for the different agents you trained. The following parameters are displayed (AurelianTactics, 2018; Mattar et al., 2018):

#### Cumulative Reward

The cumulative reward is total amount of rewards the agent collected each step. In a normal training scenario this should consistently increase, but in a game scenario when the agent is playing against another training agent the reward might go down because the other agent is improving faster.

#### Entropy

Entropy represents the randomness in the agent’s decisions. This should decrease over time. If the entropy goes down to fast or does not decrease when using a discrete action space, the beta parameter should be adjusted in the .yaml file.

#### Learning Rate

The learning rate determines the extent that newly acquired information overrides older information. In a metaphorical way, it represents the learning speed at which machine learning models can learn. The value of the learning rate should go down over time.

#### Policy Loss

The process for deciding actions is called the policy. The mean magnitude of policy loss function indicates how much this policy changes. The goal is to have the magnitude decrease during a training. The policy loss value will oscillate over time, generally keeping a value less than 1.0.

#### Value Estimate

The value estimate takes the average value for all states visited by the agent. The value estimate should increase during training, as an increasing reward means that the agent is getting better at understanding its environment. The value is used by the agent as a prediction of the future reward it will receive at a given point. When using self-play this value in a zero-sum game this value will stick around zero since the rewards for the winner will always compensate for the negative rewards for the loser, and vice versa.

#### Value Loss

The value loss indicates the mean loss of the value function update. This correlates to how good the model is at predicting the value of each state. This value should go up while the agent is learning and go down when the reward starts to stabilize.

## Bugs

Working with ML Agents made clear that the framework is not bug free. Following are two main bugs that might lead to unexpected behavior.

### Action mask issues

Sometimes the CollectDiscreteActionMasks(…) function is called at the wrong moment, updating the action mask too late. This can lead to two problematic situations. In the first situation the action mask will not contain illegal moves, this can be solved by doing an extra check in the OnActionReceived(…) function. A list named mask keeps track of the actual action mask, and when the list contains the proposed move the action mask was not updated as expected. When such failed action mask updated has happened a new decision is requested, hoping that the action mask will be updated by the time OnActionReceived(…) is called again.

public override void OnActionReceived(float[] vectorAction)

{

if (mask.Contains((int)vectorAction[0]))

{

RequestDecision();

return;

}

game.DoMove(GetMove(vectorAction));

}

In the second situation the game has ended in a state where no possible moves are left (a filled board in tic-tac-toe for example). The delay on CollectDiscreteActionMasks(…) might cause the function to be called after the game has ended, which will cause the action mask to mask every possible move. This would not be an issue if it were not that Unity throws an error for masking of every move. A solution for this problem is a quick check in CollectDiscreteActionMasks(…) that avoids setting an action mask in a situation where no possible moves would be left.

public override void

CollectDiscreteActionMasks(DiscreteActionMasker discreteActionMasker)

{

mask = GetActionMask();

if (mask.Count == vectorActionLength)

return;

discreteActionMasker.SetMask(0, mask);

}

### OnEpisodeBegin issues

It seems like ML Agents sometimes does a OnEpisodeBegin() call before the previous episode has ended. A solution for this problem is using a bool to keep track if there is an active episode. Set the variable on false right before calling EndEpisode():

isReady = false;

Then in the OnEpisodeBegin() function you add a check on the bool-variable:

public override void OnEpisodeBegin()

{

if (isReady == true)

return;

else

isReady = true;

...

}

# The demo

The demo consists out of a Unity framework for board games, some general-purpose algorithmic AI for board games and some trained machine learning AI for tic-tac-toe and connect-four.

## Framework

The demo is a Unity framework that allows to quickly build board games. The framework allows to quickly create a board by changing the colors on the Square-prefab and linking the game in the Board-component. The board will construct itself based on the board size of the game.

It allows to quickly build a game by creating a subclass of Game. There are only few abstract functions that need to be implemented. List<Position> GetPossibleMoves(Board board, IGameAgent gameAgent) has to return the possible moves based on the board state. State GetState(Board board, IGameAgent gameAgent) needs to return the board state (win/loss/draw/playing) of a certain position based on who is the active agent. float GetScore(Board board, IGameAgent gameAgent, State gameState) is the heuristic function used by the algorithmic agents.

Agents can be added by making a derived class from GameAgent or one of its subclasses (AlgorithmAgent/MLAgent). The framework also contains some premade agents, most of them can be plugged into any game, while the others can be used for reference.

## Algorithmic agents

All the described algorithms described in this paper also have a functional agent in the framework. Most of these agents are game independent, so they can be quickly used when adding a new game. To use one of the agents just drag and drop the agent game object in the list of players in the game component.

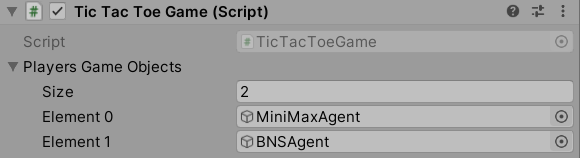


Figure 7- Tic-tac-toe game between a MiniMax and a BNS agent

## Machine learning agents

To add a machine learning agent, you have to implement the MLAgent-class, a derived class from the ML Agents Agent-class. There are two subclasses that already have some extra functionality, the ActorAgent and the EvaluatorAgent. The actor agent uses a discreet action space that represents the possible actions and plays the chosen action/move, while the evaluator agent uses a continuous action space that represents the score for a certain possible move and then plays the highest scoring move. The actor agent trains faster, but the evaluator agent might be interesting to do further research or when you want to rank the possible moves from bad to worse. This agent can also be used as the heuristic function for the algorithm-based agents.

## Trained agents

Working on a machine learning agent can be very challenging for a beginner. There are many parameters and training methods and each training session takes a lot of time. When the results are not as expected there is always the doubt if it was due to the training parameters or because the training session was too short. The whole process is very trail-and-error based. Luckily, while the agent is training there is a lot of time to browse through the forums for new ideas.

In a way these ideas sometimes lead to a big improvement, other times they do not seem to change anything or even worsen the agent’s performance. Testing takes time and thus sometimes a good idea might be thrown away because there was no immediate result. During this research it became clear though that making a machine learning AI solution takes way more time than creating an algorithm-based AI. When working with a lot of resources and time on a complex game, like the AlphaGo project, it is definitely worth using machine learning. A machine learning AI might come with moves that go beyond human creativity as shown by DeepMind’s AlphaGo.

Working on a connect-four agent it quickly become clear that the training time took too long to do experiments, so tic-tac-toe became the main focus. For the demo, a lot of different training scenarios were tried, but due to the huge number of parameters it is hard to tell why certain scenario’s failed. Some of the scenarios where probably cancelled to early too because it looked like the agent was not improving. Some of the experiments are listed below.

* Starting agent vs Responding agent
* Agent vs Random
* Self-play
* Using the agent for continuous evaluation of the position
* Exercises
* PPO vs SAC
* Convolutional neural network-based agent
* Tournament play (multiple agents)

A starting agent against a responding agent will give you two different agents, both experts at playing with one color. In tests where the agents played with the opposite color against a random agent they still performed okay, the starting agent actually even outperformed the responding agent when responding to the random agent. Using two different agents for each color can be a smart trick to make a better AI.

Training against a random agent is not advised. The random agent will not punish the agent for doing bad moves and thus the training agent will get rewarded for doing obvious moves that are not countered.

For a beginner it is recommended to avoid self-play, since it quickly became clear that it is hard to measure progress during training. The Elo-score given by ML Agents is not a good indicator for how good an agent is at playing the game, and the Elo-score is the only indicator of progress. In some scenarios the agent was also overfitting to itself, although longer training might have overcome this problem.

The evaluator agent returns a value for the position and then picks the highest scoring move. The agent has to return multiple values per move (one evaluation for each possible position instead of the id of the chosen action) which means slower training. The evaluator agent also returns a continues value instead of a discreet one which also leads to slower training. Since the training took so long, the idea was disbanded to put the focus back on other training scenarios. This agent can be used as the heuristic function for the algorithmic agents. Using symmetry (as described in 1.4) it was possible to reduce the amount of positions the evaluator agent had to understand. This gave an improvement in the training time of the agent.

Training an agent by creating exercises leads to a decent AI but it just shifts the problem from creating a good AI player to creating a good AI exercise maker. It is also important to be aware of possible overfitting to the exercises.

Training a PPO agent against a SAC agent showed that PPO starts out as the better agent (with a really high difference in reward) but SAC catches up (around 1 million steps) and gets better than the PPO agent. Since this statement is based on a sample size of one it is not sure this will hold for every board game.

|  |  |
| --- | --- |
| +0.5  +0.4  +0.3  +0.2  +0.1  +0.0  -0.1  -0.2  -0.3  -0.4  -0.5 |  |

0 500.000 1.000.000

Figure 8 - PPO (orange) vs SAC (blue), rewards over steps

Using the Python API, it is probably possible to use a convolutional neural network, but no resources were found on how to do it. Convolutional layers will create a new board of a smaller size that holds data on whether the original board contains certain states. ML Agents uses these kinds of networks when using the texture sensor, but the sensor does not allow small textures and thus it would bring a big overhead in training time. Instead the TicTacToeCNNAgent has some extra inputs which are convolutions on the original board state. This agent seems to produce good results but a solution using the Python API would be better. There are probably also some other neural network variants which can be used through the Python API that might be more optimal for board game agents.

Tournament play seems like the most promising of all these training environments, it allows to train multiple agents at the same time. This allows to compare the results of those agents, making more clear which agents are improving, and which agents are good. It is also a good way to avoid overfitting, since the bigger the amount of agent the bigger the variation in moves.

The final two tic-tac-toe agents (PPO and SAC) are tested by having them play 1000 games against a random agent, a perfect agent and themselves. In the figure below the number of wins per 1000 games is displayed, the wins do not add up to 1000 since games can also end up in a draw. Notice how there is still some randomness left in the PPO and SAC decision process (otherwise we would see only numbers like 0 and 1000). There seems to be no documentation that explains this behavior.

Both the PPO and SAC agent seem to clearly outperform the random agent, although they still lose some games. The PPO agent seems to be able to draw the perfect player every time, but this could be because the random agent is not playing those particular moves that makes the PPO agent make mistakes. The SAC agent also performs well against the perfect agent with only 15 losses. When the PPO faces the SAC agent the PPO agent suddenly seems to outperform the SAC agent, but this is probably a case of overfitting.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | vs. random | vs. Perfect | vs. PPO | vs. SAC |
| Random |  | 0 | 29 | 4 |
| Perfect | 964 |  | 0 | 15 |
| PPO | 897 | 0 |  | 384 |
| SAC | 967 | 0 | 3 |  |

Figure 9 - Tournament, amount of wins

It is also noticeable that the SAC agent seems to have a higher win-rate against the random player than the perfect player. This is probably because the perfect player is looking for the best move considering the opponent is also a perfect player, while the SAC agent is trained playing against a non-perfect player. The SAC agent is probably “baiting” bad moves from its opponent.

## Further research

In general, this research paper lacks concrete testing data. This is because training an agent takes about 5 days on the laptop that was used, and multiple parameters were changed each run to faster improve the agents. With a better computer for machine learning training purposes or a cloud service for training it might have been possible to also generate enough data, but the time frame is still quite short for machine learning research.

At this point most conclusions are based on intuitions that follow out of reading the research paper, reading the forums and the results from the performed experiments. More concrete experiments could hard prove some of these conclusions.

The most interesting routs to go for further research is using the evaluator agent as the heuristic function for the search algorithms, researching a good adversary for training a machine learning agent, optimizing the neural network parameters (hidden units, hidden layers) based on the board size and researching how to use the Python API to the full extend.

# Conclusion

In general, this paper concludes that the search algorithms are a more reliable and faster solution than the machine learning solution. Best Node Search (BNS) seems like the best search algorithm although in some cases other algorithms might outperform BNS.

Unity ML Agents is not on point yet, especially when it comes to board games, but it looks like it is possible to create a decent AI with the framework with enough machine learning knowledge, computing power and time. ML Agents still has some bugs and unexplained behaviors, but at the same time looks very promising for when it goes into alpha and the documentation is up to date and complete. ML Agents also has some bugs left which feel like you as a programmer did something wrong (from the main decision loop that runs out of sequence to random episode begin calls).

It takes a lot of time getting acquainted with machine learning, the installation of the necessary software always comes with a lot of trouble (versions that do not match up, documentation that is outdate, …). After building and training your first network comes a phase of having no clue what training and network parameters to change. Then comes a period of trial and error without making a lot of progress. After spending 4 months of tweaking parameters and searching through forums, I can say I am still not sure what I am doing, although the results are getting better and better. The process is fun, you will learn a lot, but when you quickly want to make an AI for your board game, I would recommend doing it the old-school way.

The machine learning solution has the advantage that it calculates a response really fast, and consistently fast for each move, while some algorithm based agents can be slower or are less consistent in the time it takes to calculate a move (depending on the amount of branches and the depth of the game tree).

The combination of a search algorithm and a machine learning solution for the heuristic function (a positional value estimator) is a promising hybrid. In the demo the Evaluator agent shows that it is possible to use ML Agents agent with a continuous action space to generate a heuristic value for a position.

# References

Heineman, G. T., Pollice, S., & Selkow, G. (2016). *Algorithms in a Nutshell* (2nd edition). O’Reilly. <http://www.r-5.org/files/books/computers/algo-list/common/Heineman_Pollice_Selkow-Algorithms_in_a_Nutshell-EN.pdf>

AurelianTactics. (2018, December 13). *Understanding PPO Plots in TensorBoard.* Medium.

<https://medium.com/aureliantactics/understanding-ppo-plots-in-tensorboard-cbc3199b9ba2>

Cohen, A. (2020, June 5). *Training intelligent adversaries using self-play with ML-Agents*. Unity Technologies Blog. <https://blogs.unity3d.com/2020/02/28/training-intelligent-adversaries-using-self-play-with-ml-agents/>

Gonzalez, T., Diaz-Herrera, J., & Tucker, A. (2014). *Computing Handbook*, Third Edition. Taylor & Francis. <https://books.google.be/books?id=wyHSBQAAQBAJ&printsec=frontcover&hl=nl#v=onepage&q&f=false>

Mattar, M., Juliani, A., Berges, V., Pang, D., Esh, Ward, J., & Shih, J. (2018, October 19). *Training with Proximal Policy Optimization*. GitHub. <https://github.com/gzrjzcx/ML-agents/blob/master/docs/Training-PPO.md>

Plaat, A., Schaeffer, J., Pijls, W., & de Bruin, A. (1996). *Best-First Fixed-Depth Minimax Algorithms*. ICGA Journal, 19(4), 247. <https://askeplaat.files.wordpress.com/2013/01/aij-final1.pdf>

Plaat, A. (January 1997). *A Minimax Algorithm faster than NegaScout*. arxiv.org.

<https://arxiv.org/ftp/arxiv/papers/1404/1404.1511.pdf>

Rutko, D. (2011). *Fuzzified Algorithm for Game Tree Search with Statistical and Analytical Evaluation*. Scientific Papers, University of Latvia, 770, 90–111.

<https://www.bjmc.lu.lv/fileadmin/user_upload/lu_portal/projekti/bjmc/Contents/770_7.pdf>

Rutko, D. (2012). *Fuzzified Game Tree* (Doctoral thesis, University of Latvia).

<https://core.ac.uk/download/pdf/71754336.pdf>

Schaeffer, J. (1989). *The history heuristic and alpha-beta search enhancements in practice*. IEEE Transactions on Pattern Analysis and Machine Intelligence, 11(11), 1203–1212.

<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.56.9124&rep=rep1&type=pdf>

Tam, A. S. (2019, March 15). *Tic-tac-toe using AI from the last century.* www.adrian.idv.hk.

<https://www.adrian.idv.hk/2019-03-15-Tic-tac-toe/>

Teng, E. (2020, June 5). *Training your agents 7 times faster with ML-Agents*. Unity Technologies Blog.

<https://blogs.unity3d.com/2019/11/11/training-your-agents-7-times-faster-with-ml-agents/>

Thill, M. (2008, February 14*). Constructing Agents for Connect-4: Board Representations. markusthill.github.io.*

<https://markusthill.github.io/programming/connect-4-board-representations/>

Unity. (2020a, July 10). *ML Agents: Installation*. GitHub.

<https://github.com/Unity-Technologies/ml-agents/blob/release_4_docs/docs/Installation.md>

Unity. (2020b, July 7). *ML Agents: Getting started*. GitHub.

<https://github.com/Unity-Technologies/ml-agents/blob/release_4_docs/docs/Getting-Started.md>

Wikipedia contributors. (2020, July 2). *Alpha–beta pruning.* Wikipedia, The Free Encyclopedia.

<https://en.wikipedia.org/w/index.php?title=Alpha%E2%80%93beta_pruning&oldid=965591416>

Wikipedia contributors. (2019, June 27). Killer heuristic. Wikipedia, The Free Encyclopedia.

<https://en.wikipedia.org/w/index.php?title=Killer_heuristic&oldid=903761614>

Wikipedia contributors. (2020, August 3). *Negamax*. I Wikipedia, The Free Encyclopedia.

<https://en.wikipedia.org/w/index.php?title=Negamax&oldid=970965795>

Wikipedia contributors. (2019, December 13). *Principal variation search*. Wikipedia, The Free Encyclopedia.

<https://en.wikipedia.org/w/index.php?title=Principal_variation_search&oldid=930557898>

Wikipedia contributors. (2019, December 17). *SSS\*.* In *Wikipedia, The Free Encyclopedia*.

[https://en.wikipedia.org/w/index.php?title=SSS\*&oldid=931178426](https://en.wikipedia.org/w/index.php?title=SSS*&oldid=931178426)