Contents

[Introduction 3](#_Toc47998422)

[0. Representing boardgames 4](#_Toc47998423)

[1.1. The state 4](#_Toc47998424)

[1.2. Actions 5](#_Toc47998425)

[1.3. Consequences 5](#_Toc47998426)

[1.4. Symmetrical games 6](#_Toc47998427)

[2. Boardgame AI 8](#_Toc47998428)

[2.1. MiniMax 8](#_Toc47998429)

[2.1.1. Algorithm 8](#_Toc47998430)

[2.1.2. Implementation 8](#_Toc47998431)

[2.2. NegaMax 9](#_Toc47998432)

[2.3. AlphaBeta 9](#_Toc47998433)

[2.3.1. Algorithm 9](#_Toc47998434)

[2.3.2. Implementation 10](#_Toc47998435)

[2.3.3. Heuristic improvements 11](#_Toc47998436)

[2.4. Unity ML Agents 13](#_Toc47998437)

[2.4.1. PPO vs SAC 14](#_Toc47998438)

[2.4.2. Training parameters 15](#_Toc47998439)

[Recurrent Neural Network Hyperparameters (Optional) 17](#_Toc47998440)

[Sequence Length 17](#_Toc47998441)

[Memory Size 17](#_Toc47998442)

[(Optional) Intrinsic Curiosity Module Hyperparameters 17](#_Toc47998443)

[Curiosity Encoding Size 17](#_Toc47998444)

[Curiosity Strength 17](#_Toc47998445)

[Training Statistics 17](#_Toc47998446)

[Cumulative Reward 17](#_Toc47998447)

[Entropy 17](#_Toc47998448)

[Learning Rate 17](#_Toc47998449)

[Policy Loss 17](#_Toc47998450)

[Value Estimate 17](#_Toc47998451)

[Value Loss 17](#_Toc47998452)

[2.4.3. Tensor Board 18](#_Toc47998453)

[3. References 23](#_Toc47998454)

# Introduction

# Representing boardgames

Boardgames have a state, actions and consequences. In these boardgames 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 an 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 cell 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 really 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](https://markusthill.github.io/programming/connect-4-board-representations/), 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 find 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.

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

## Actions

As said before the actions for the games in the scope of 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 capturing a group in a game of Go 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 made out of 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 TicTacToe or any three in a row game as an example. These games are one one 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 maskIndex = 0; maskIndex < masks.Length; ++maskIndex)

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

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

{

ulong mask = masks[maskIndex].bits

\* columnMasks[x] & rowMasks[y];

if (bits & mask == mask)

return true;

}

return false;

}

This method is really 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;

For a game of Go it becomes extremely hard to define a win condition based on the board position, since the game end when the two players agree on ending the game and then the players need to agree on how to dived the territories and what the actual score is. If the players do not agree the game just continues.

## Symmetrical games

A machine learning agent has no concept of symmetry and has to learn certain concepts double which humans quickly would recognise as symmetrical. For example, the following TicTacToe situations are completely equivalent to each other:

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

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

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

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

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

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

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

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 TicTacToe.

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);

}

}

# Boardgame 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. The AlphaBeta algorithm is an obvious improvement upon this MiniMax algorithm. MiniMax and AlphaBeta both need either a lot of time or good heuristic function (function that estimates a score for a certain board state) on games that take some more time. Writing this heuristic function needs a good understanding of the game at hand. To avoid this problem a machine learning solution might be a solution, this way the computer develops the understanding and the developer doesn’t need to think about a good heuristic function.

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.

## MiniMax

MiniMax is a simple algorithm that searches through the possible moves, each turn optimizing for the active player. The algorithm can also keep track of the amount 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).

### Algorithm

The min max 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. The algorithm in pseudocode is displayed below, the original call would be minimax(origin, depth, TRUE) (Heineman et al., 2016, pp. 174–180):

function minimax(node, depth, 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, 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 <https://en.wikipedia.org/wiki/Negamax>

When working on the MiniMax algorithm it became quickly clear that there is no need to Pseudocode:

function negamax(node, depth, color) is

if depth = 0 or node is a terminal node then

return color × the heuristic value of node

value := −∞

for each child of node do

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

return value

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

negamax(rootNode, depth, 1)

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

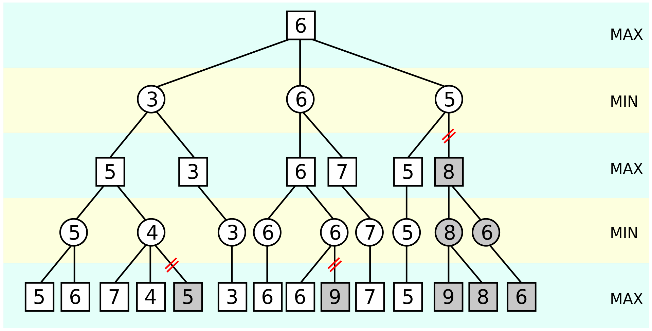
negamax(rootNode, depth, −1)

(Heineman et al., 2016, pp. 180-183)

## AlphaBeta

(Heineman et al., 2016, pp. 183-189)

### Algorithm



The algorithm maintains two values, alpha and beta, which represent the minimum score that the maximizing player is assured of and the maximum score that the minimizing player is assured of respectively. Initially, alpha is negative infinity and beta is positive infinity, i.e. both players start with their worst possible score. Whenever the maximum score that the minimizing player (i.e. the "beta" player) is assured of becomes less than the minimum score that the maximizing player (i.e., the "alpha" player) is assured of (i.e. beta < alpha), the maximizing player need not consider further descendants of this node, as they will never be reached in the actual play.

To illustrate this with a real-life example, suppose somebody is playing chess, and it is their turn. Move "A" will improve the player's position. The player continues to look for moves to make sure a better one hasn't been missed. Move "B" is also a good move, but the player then realizes that it will allow the opponent to force checkmate in two moves. Thus, other outcomes from playing move B no longer need to be considered since the opponent can force a win. The maximum score that the opponent could force after move "B" is negative infinity: a loss for the player. This is less than the minimum position that was previously found; move "A" does not result in a forced loss in two moves.

#### MiniMax

Pseudo code with alphabeta(origin, depth, −∞, +∞, TRUE) as initial call:

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

(Alpha-beta pruning, n.d.)

#### NegaMax

AlphaBetaNegaMax <https://en.wikipedia.org/wiki/Negamax>

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 := −∞

foreach 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

#### Killer heuristic

KillerHeuristic <https://en.wikipedia.org/wiki/Killer_heuristic>

#### Move ordering

Move ordering <https://en.wikipedia.org/wiki/Negamax>

It is possible to make some further improvements on the Alpha-Beta algorithm, even without losing accuracy. A first method is to increase the amount of skipped branches. This can be done to check those branches that might lead to skipping other branches first (high potential moves). This can also be done by checking those moves first that scored the highest when testing for the previous move (Alpha-beta pruning, n.d.). Another trick is the killer heuristic, here the last move that lead to a skipped branch at the same level will be checked first since it has a high chance of leading to more branch skipping (Killer heuristic, n.d.).

Killer + move ordering: <https://www.adrian.idv.hk/2019-03-15-tictactoe/> (Tam, 2019)

Alpha–beta search can be made even faster by considering only a narrow search window (generally determined by guesswork based on experience). This is known as *aspiration search*. In the extreme case, the search is performed with alpha and beta equal; a technique known as [*zero-window search*](https://en.wikipedia.org/wiki/MTD-f#Zero-Window_Searches), *null-window search*, or *scout search*. This is particularly useful for win/loss searches near the end of a game where the extra depth gained from the narrow window and a simple win/loss evaluation function may lead to a conclusive result. If an aspiration search fails, it is straightforward to detect whether it failed *high* (high edge of window was too low) or *low* (lower edge of window was too high). This gives information about what window values might be useful in a re-search of the position.

Over time, other improvements have been suggested, and indeed the Falphabeta (fail-soft alpha-beta) idea of John Fishburn is nearly universal and is already incorporated above in a slightly modified form. Fishburn also suggested a combination of the killer heuristic and zero-window search under the name Lalphabeta ("last move with minimal window alpha-beta search").

### Transposition Tables

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

alphaOrig := α

(\* Transposition Table Lookup; node is the lookup key for ttEntry \*)

ttEntry := transpositionTableLookup(node)

if ttEntry is valid and ttEntry.depth ≥ depth then

if ttEntry.flag = EXACT then

return ttEntry.value

else if ttEntry.flag = LOWERBOUND then

α := max(α, ttEntry.value)

else if ttEntry.flag = UPPERBOUND then

β := min(β, ttEntry.value)

if α ≥ β then

return ttEntry.value

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

(\* Transposition Table Store; node is the lookup key for ttEntry \*)

ttEntry.value := value

if value ≤ alphaOrig then

ttEntry.flag := UPPERBOUND

else if value ≥ β then

ttEntry.flag := LOWERBOUND

else

ttEntry.flag := EXACT

ttEntry.depth := depth

transpositionTableStore(node, ttEntry)

return value

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

negamax(rootNode, depth, −∞, +∞, 1)

TranspositionTablesAlphaBetaNegaMax <https://en.wikipedia.org/wiki/Negamax>

### PVS, NegaScout, MTDf(n)

### MTD-f

### BNS

### SSS\*

## 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 actually 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 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 this is not really the case for ML Agents. Although ML Agents appeared in 2017 it is still in beta and

Agents can be trained using reinforcement learning, imitation learning, neuroevolution, or other machine learning methods through a simple-to-use Python API. Unity additionally offers implementations (based on TensorFlow) of state-of-the-art algorithms to enable game developers and hobbyists to easily train intelligent agents for 2D, 3D and VR/AR games. These trained agents can be used for multiple purposes, including controlling NPC (non-player character) behavior (in a variety of settings such as multi-agent and adversarial), automated testing of game builds and evaluating different game design decisions pre-release.

Use Cases

Unity ML-Agents can benefit:

Academic researchers interested in studying complex multi-agent behavior in realistic competitive and cooperative scenarios.

Industry researchers interested in large-scale parallel training regimes for robotics, autonomous vehicle, and other industrial applications.

Game developers interested in filling virtual worlds with intelligent agents each acting with dynamic and engaging behavior.

Features

The Unity ML-Agent Toolkit is an open-source solution with the following features:

Unity environment control from Python

10+ sample Unity environments

Support for multiple environment configurations and training scenarios

Train memory-enhanced agents using deep reinforcement learning

Easily definable Curriculum Learning scenarios

Broadcasting of agent behavior for supervised learning

Built-in support for Imitation Learning

Flexible agent control with On Demand Decision Making

Visualizing network outputs within the environment

Simplified set-up with Docker

Wrap learning environments as a gym

Required software (20/07/2020)

* Unity 2018.4 or Later
* Unity MLAgents package
* Python 3.6/3.7 (64 bit)

Creating a Python virtual environment

* Create a folder for your virtual environments

*C:\python-envs*

* Create a new virtual environment

*python -m venv C:\python-envs\mlagents-env*

Using a the virtual environment

* Activate using

*C:\python-envs\mlagents-env\Scripts\activate*

* Deactivate using

*deactivate*

Prepare using mlagents

* Install mlagents

*pip3 install mlagents*

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

<https://docs.unity3d.com/Packages/com.unity.ml-agents@1.0/api/Unity.MLAgents.Agent.html#Unity_MLAgents_Agent_CollectObservations_Unity_MLAgents_Sensors_VectorSensor_>

<https://github.com/gzrjzcx/ML-agents/blob/master/docs/Training-PPO.md>

## PPO vs SAC

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

The primary section of the trainer config file is a set of configurations for each Behavior in your scene. These are defined under the sub-section behaviors in your trainer config file. Some of the configurations are required while others are optional. To help us get started, below is a sample file that includes all the possible settings if we're using a PPO trainer with all the possible training functionalities enabled (memory, behavioral cloning, curiosity, GAIL and self-play). You will notice that curriculum and environment parameter randomization settings are not part of the behaviors configuration, but in their own section called environment\_parameters.

Below is a typical yaml file for a PPO setup:

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

Below is a typical SAC setup:

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

To use self-play the following parameters should be added (possible with PPO and SAC):

self\_play:

window: 10

play\_against\_latest\_model\_ratio: 0.5

save\_steps: 50000

swap\_steps: 2000

team\_change: 100000

### Training parameters

When training an agent you have to set the training parameters in the .yaml file. These parameters determine both the agents 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 determins 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 buffersize (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 behaviour 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 amount 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 amount 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.

### (Optional) Intrinsic Curiosity Module Hyperparameters

The below hyperparameters are only used when use\_curiosity is set to true.

### Curiosity Encoding Size

curiosity\_enc\_size corresponds to the size of the hidden layer used to encode the observations within the intrinsic curiosity module. This value should be small enough to encourage the curiosity module to compress the original observation, but also not too small to prevent it from learning the dynamics of the environment. Typical Range: 64 - 256

### Curiosity Strength

curiosity\_strength corresponds to the magnitude of the intrinsic reward generated by the intrinsic curiosity module. This should be scaled in order to ensure it is large enough to not be overwhelmed by extrinsic reward signals in the environment. Likewise it should not be too large to overwhelm the extrinsic reward signal. Typical Range: 0.1 - 0.001

### Training Statistics

To view training statistics, use TensorBoard. For information on launching and using TensorBoard, see here.

### Cumulative Reward

The general trend in reward should consistently increase over time. Small ups and downs are to be expected. Depending on the complexity of the task, a significant increase in reward may not present itself until millions of steps into the training process.

### Entropy

This corresponds to how random the decisions of a Brain are. This should consistently decrease during training. If it decreases too soon or not at all, beta should be adjusted (when using discrete action space).

### Learning Rate

This will decrease over time on a linear schedule.

### Policy Loss

These values will oscillate during training. Generally they should be less than 1.0.

### Value Estimate

These values should increase as the cumulative reward increases. They correspond to how much future reward the agent predicts itself receiving at any given point.

### Value Loss

These values will increase as the reward increases, and then should decrease once reward becomes stable.

We now break apart the components of the configuration file and describe what each of these parameters mean and provide guidelines on how to set them. See [Training Configuration File](https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-Configuration-File.md) for a detailed description of all the configurations listed above, along with their defaults. Unless otherwise specified, omitting a configuration will revert it to its default.

Self play

<https://blogs.unity3d.com/2020/02/28/training-intelligent-adversaries-using-self-play-with-ml-agents/>

Improvements

<https://www.codeproject.com/Articles/5160398/A-Tic-Tac-Toe-AI-with-Neural-Networks-and-Machine>

### Tensor Board

To get more information about the training session Tensor Board comes in handy. When the training ends, 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.

### Training

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 don’t 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 were 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.

- Starting agent vs Responding agent

- Agent vs Random

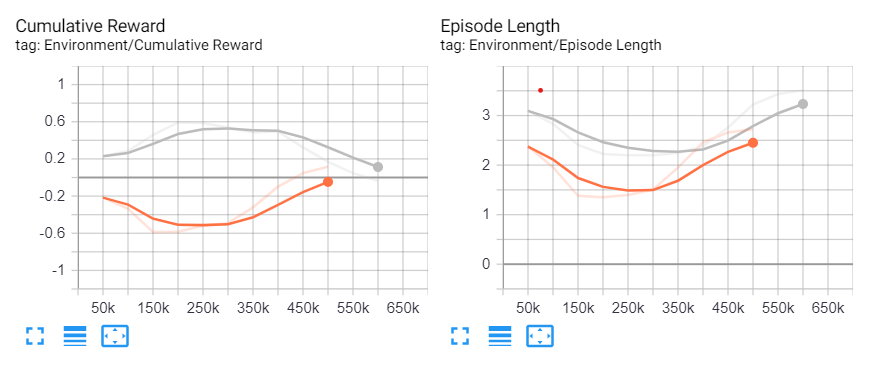
- Continuous evaluation instead of discrete actions

- Self play

- SAC instead of PPO

- Tournament play

Tic Tac Toe:



Behaviour A trained against Behaviour B with Behaviour A always being the starting player.

|  |  |  |  |
| --- | --- | --- | --- |
| 1000 < Games | Beginner wins | Draw | Beginner loses |
| Perfect vs Perfect | 100.0% | 0.0% | 0.0% |
| Random vs Random | 58.2% | 11.7% | 30.2% |
| Random vs Behaviour A | 21.4% | 5.0% | 73.6% |
| Random vs Behaviour B | 49.9% | 16.5% | 33.6% |
| Behaviour A vs Random | 96.0% | 2.2% | 1.8% |
| Behaviour A vs Behaviour A | 97.5% | 1.7% | 0.8% |
| Behaviour A vs Behaviour B | 21.0% | 49.4% | 29.6% |
| Behaviour B vs Random | 81.0% | 10.6% | 8.4% |
| Behaviour B vs Behaviour A | 67.9% | 17.1% | 14.9% |
| Behaviour B vs Behaviour B | 80.4% | 10.4% | 9.2% |

These are some really weird results, at least it is visible the training actually worked. Some remarks:

- You would expect Behaviour vs Behaviour matches to have only one outcome, but they seem to be creative. As for now we are clueless why this is happening.

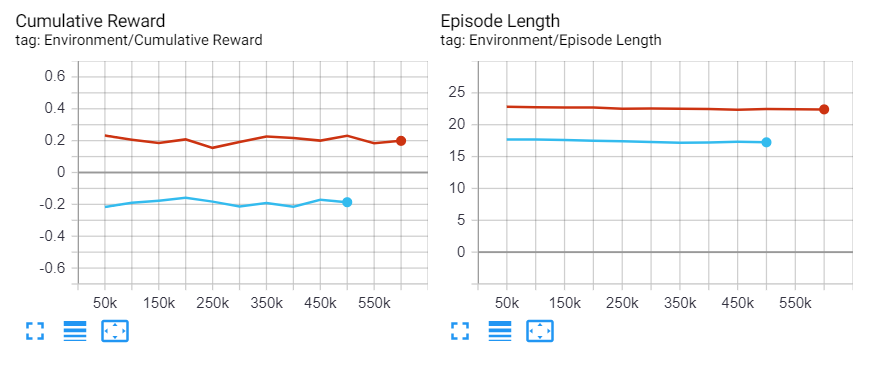
- Although Behaviour B was trained by always responding to the moves of Behaviour A it also performs okay in the starting position.

- Behaviour B became an expert in drawing/winning to Behaviour A (79% of the matches) but has more trouble drawing/winning against random moves (50% of the matches) which is clearly a case of overfitting.

- Although these results look good these behaviours still miss finishing/blocking every three in a row. The behaviours do not understand the game of TicTacToe, they just have an incomplete statistical idea of which moves are good. Many previous tests are not reported because the results looked worthless while probably they would have given a winrate just above Random vs Random.

The next agent is an evaluator that gives a score to each board position. These evaluations then get used to select the best move.

-> try rotations etc

-> try spectator view  


|  |  |  |  |
| --- | --- | --- | --- |
| 1000 < Games | Beginner wins | Draw | Beginner loses |
| Perfect vs Perfect | 100.0% | 0.0% | 0.0% |
| Random vs Random | 58.2% | 11.7% | 30.2% |
| Random vs Behaviour A | 50.1% | 7.9% | 42.0% |
| Random vs Behaviour B | 49.1% | 9.8% | 41.0% |
| Behaviour A vs Random | 68.2% | 9.1% | 22.7% |
| Behaviour A vs Behaviour A | 71.2% | 10.0% | 18.8% |
| Behaviour A vs Behaviour B | 55.8% | 11.8% | 32.4% |
| Behaviour B vs Random | 69.1% | 11.7% | 19.2% |
| Behaviour B vs Behaviour A | 61.1% | 11.8% | 27.1% |
| Behaviour B vs Behaviour B | 60.7% | 12.4% | 27.0% |

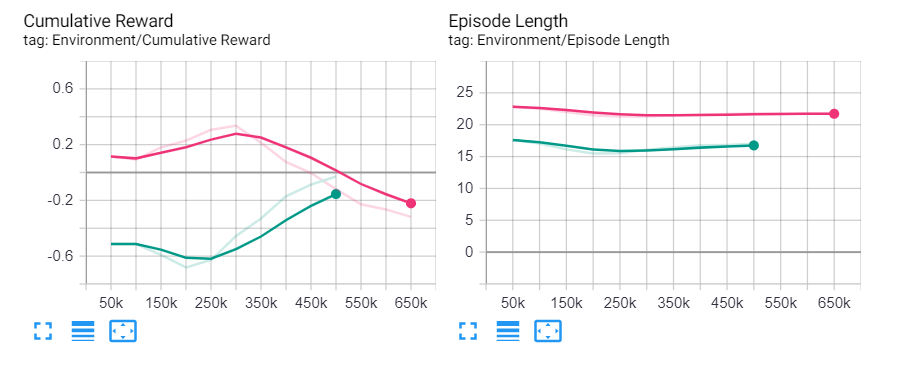
Agent that turns the board around so that it only has to evaluate moves on the following positions:

000

0X0

XX0

Because of the symmetry…



|  |  |  |  |
| --- | --- | --- | --- |
| 1000 < Games | Beginner wins | Draw | Beginner loses |
| Perfect vs Perfect | 100.0% | 0.0% | 0.0% |
| Random vs Random | 58.2% | 11.7% | 30.2% |
| Random vs Behaviour A |  |  |  |
| Random vs Behaviour B | 27.6% | 21.9% | 50.5% |
| Behaviour A vs Random | 89.4% | 4.0% | 6.6% |
| Behaviour A vs Behaviour A |  |  |  |
| Behaviour A vs Behaviour B |  |  |  |
| Behaviour B vs Random | 78.7% | 11.8% | 9.5% |
| Behaviour B vs Behaviour A |  |  |  |
| Behaviour B vs Behaviour B |  |  |  |

Less training (only 50.000 steps 32 neurons x 3 layers)

A vs R 73.1% 10.0% 16.9%

A vs R 77.5% 9.1% 13.4%

A vs R 64.1% 11.0% 24.8%

Second agent lags behind while this should not be happening

*C:\python-envs\mlagents-env\Scripts\activate*

cd C:\Github\BoardGameAI\Four\Assets\ML-Agents

mlagents-learn C:\Github\BoardGameAI\Four\Assets\ML-Agents\TicTacToeBehaviour.yaml --run-id=XXX

Self play introcuding a lot of new problems

self\_play:

save\_steps: 5000

team\_change: 10000

swap\_steps: 2000

window: 10

# References

Wikipedia contributors. (2020, July 2). Alpha–beta pruning. In *Wikipedia, The Free Encyclopedia*. Retrieved August 9, 2020, from

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

Wikipedia contributors. (2019, June 27). Killer heuristic. In *Wikipedia, The Free Encyclopedia*. Retrieved August 10, 2020, from

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

Wikipedia contributors. (2020, June 21). Minimax. In *Wikipedia, The Free Encyclopedia*. Retrieved August 9, 2020, from

<https://en.wikipedia.org/w/index.php?title=Minimax&oldid=963711709>

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

Thill, M. (2008, February 14*). Constructing Agents for Connect-4: Board Representations. markusthill.github.io.* Retrieved August 9, 2020, from

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

Mattar, M., Juliani, A., Berges, V., Pang, D., Esh, Ward, J., & Shih, J. (2018, October 19). *Training with Proximal Policy Optimization*. GitHub. Retrieved August 9, 2020, from

<https://github.com/gzrjzcx/ML-agents/blob/master/docs/Training-PPO.md>

Tam, A. S. (2019, March 15). *Tic-tac-toe using AI from the last century.* www.adrian.idv.hk. Retrieved August 11, 2020, from <https://www.adrian.idv.hk/2019-03-15-tictactoe/>