## Creating the board

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;

And to set the cell at column x, row y to the value bit:

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]);

}

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.

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

## Checking win conditions

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 doing 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);

}

}

Bit formatting + alpha beta search:

<https://markusthill.github.io/programming/connect-4-introduction-and-tree-search-algorithms/>

Board games and AI

<https://openreview.net/pdf?id=SklPFVXUsN>

## 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/>

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>

Symetrical games

Reducing action space by using translations, rotations and mirroring on the original board state so

idea’s:

minmax with ml agents evalution

training data

TEST RUNS:

TEST1 AI0 vs AI1: 500.000 matches = dumb AI (no preventing or finishing 4 in a row, no prioritizing middle row)

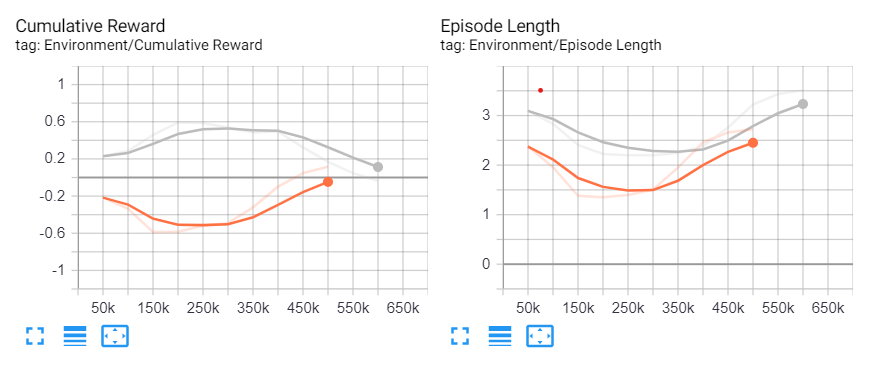
TEST 2 AI vs RND: 500.000 matches = dumb AI (no preventing or finishing 4 in a row, no prioritizing middle row)

TEST 3 AI0 vs AI1: 4x4 connect3 -> starting player makes vertical 3 in a rows, second player stops them, no further tactics visible (still dumb AI)

TEST 3 AI0 vs AI1: 4x4 connect3 -> starting player makes vertical 3 in a rows, second player stops them, no further tactics visible (still dumb AI)

=>> used basic yaml file which only was one layer deep so obviously the results were bad

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.

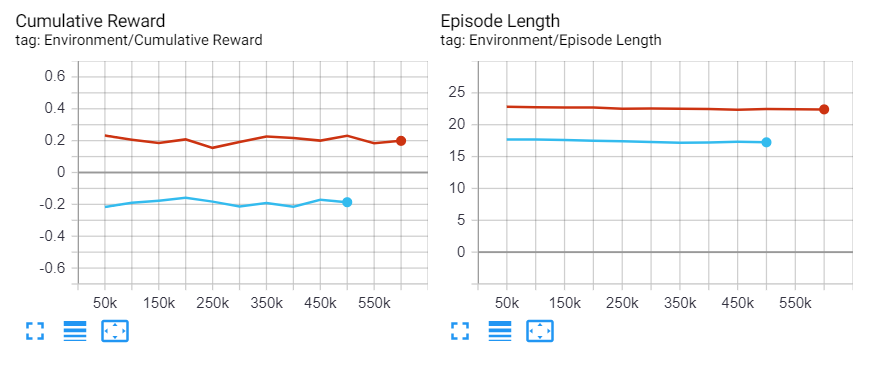
- Although Behaviour B was trained by always responding to the moves of Behaviour A it also performs okay in the starting position.

- 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% |

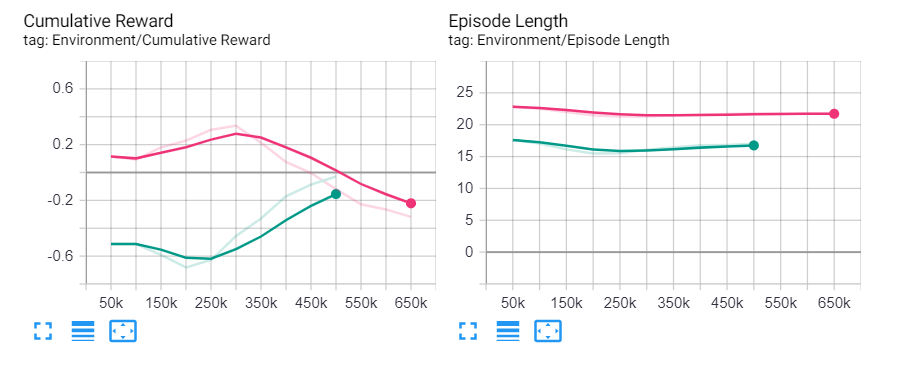
Agent that turns the board around so that it only has to evaluate moves on the following positions:

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

tensorboard --logdir=results

Self play introcuding a lot of new problems

self\_play:

save\_steps: 5000

team\_change: 10000

swap\_steps: 2000

window: 10

EVALUATOR:

behaviors:

TicTacToeBehaviour:

trainer\_type: ppo

hyperparameters:

batch\_size: 1024

buffer\_size: 4096

beta: 1e-2

epsilon: 0.2

lambd: 0.9

num\_epoch: 8

learning\_rate: 1e-5

learning\_rate\_schedule: linear

network\_settings:

normalize: false

hidden\_units: 64

num\_layers: 3

vis\_encode\_type: simple

reward\_signals:

extrinsic:

gamma: 0.999

strength: 1.0

keep\_checkpoints: 5

checkpoint\_interval: 5000

max\_steps: 1000000

time\_horizon: 41

summary\_freq: 10000

threaded: true

ACTION PICKER:

behaviors:

TicTacToeBehaviour:

trainer\_type: ppo

hyperparameters:

batch\_size: 256

buffer\_size: 512

beta: 1e-2

epsilon: 0.2

lambd: 0.9

num\_epoch: 8

learning\_rate: 1e-5

learning\_rate\_schedule: linear

network\_settings:

normalize: false

hidden\_units: 64

num\_layers: 3

vis\_encode\_type: simple

reward\_signals:

extrinsic:

gamma: 0.999

strength: 1.0

keep\_checkpoints: 5

checkpoint\_interval: 5000

max\_steps: 1000000

time\_horizon: 41

summary\_freq: 10000

threaded: true

Thill, M. (2008, 14 februari). *Constructing Agents for Connect-4: Board Representations*. markusthill.github.io. https://markusthill.github.io/programming/connect-4-board-representations/