# *Exercise 2: Decision Making in `Tic-Tac-Toe`*

## Why should you work through these exercises ?

If you ask any modern AI agent (a recent version of GPT, Claude, Gemini, Qwen, …) to solve the problems on this exercise sheet, it will likely do this with ease, without mistakes, and within minutes …

However, all computer scientists who want to be or become AI experts should at least once in their lives implement the **minmax algorithm** from scratch and get hands-on experience with its behavior. If nothing else, these exercises (problem 2.6) provide you with an opportunity for doing this …

## What should you take home from these exercises ?

Try to connect the dots! Ask yourself the following:

- To what extend do the methods in the following problems rely on human ingenuity or on data-driven insights? What are the roles of human ingenuity and statistical data analysis?
- To what extend can modern AIs assist you with the following problems? What does your answers to this question suggest for the future of software development?

## 2.1 A `Tic-Tac-Toe` game engine

In our first couple of lectures on reinforcement learning, we considered the two player game of `Tic-Tac-Toe` as a simple test-bed for decision making under adversarial conditions (if you are not familiar with this game and its mechanics, just look it up on wikipedia).

If you unpack the archive

`ttt.zip`

you obtain a folder `ttt` in which you will find the `Python` files

```
tttBase.py
tttState.py
tttAction.py
tttEngine.py
tictactoe.py
```

These, in turn, contain the code base we discussed in *Lecture 1* when we looked at ideas for how to realize a `Tic-Tac-Toe` game engine.

### 2.1.1 playing games and tournaments

(Re)familiarize yourself with the above code base. Run the code in `tictactoe.py` and see what happens.

Get used to this code and play with it! For instance, run different kinds of tournaments, say, tournaments where both players (**X** and **O**) move defensively ...

### 2.1.2 visualizing tournament outcomes

In *Lecture 1*, we saw histogram visualizations of the outcomes of a tournament (i.e. of a series of many games where **X** and **O** take turns opening the game). Can you implement `matplotlib` code which plots these kinds of histograms?

## 2.2 Additional policies for `Tic-Tac-Toe`

So far, our `Tic-Tac-Toe` game engine contains only three *move functions* or *policies* which our artificial players can apply to decide which cells to mark when it is their turn ... So, let's do something about this!

In what follows, we will write $p$ to indicate the player who is about to move in the current `Tic-Tac-Toe` round. To indicate the opponent who is to move in the next round, we will write $o$.

### 2.2.1 from defensive to offensive play

Function `move_defensively()` in `tttAction.py` realizes a *defensive* policy or move function. Player $p$ checks if there are any empty cells on the board which, when marked by $o$, would cause $p$ to loose immediately (i.e. in the next round when $o$ is to move). If such a cell exists, $p$ marks it to avoid loosing in the next round. (In general, there may be several such cells in which case $p$ could loose nevertheless.) If such a cell does not exist, $p$ moves randomly.

Analyze the mechanics behind `move_defensively()`. Use your insights to implement a function `move_offensively()` where $p$ checks if there are any empty cells on the board which, when marked by $p$, would cause $p$ to win immediately (i.e. in the current round). If such a cell exists, have $p$ mark it. If such a cell does not exist, have $p$ move randomly.

Test your code for correctness. Once you are sure it works correctly, run a tournament of 1001 games where **X** moves randomly and **O** moves offensively. Visualize the outcome of this tournament and ponder what you observe.

### 2.2.2 a combined strategy

Consider this: If the current game state is such that $p$ could win immediately, $p$ should realize that win. If $p$ can not win immediately but could loose in the next round, $p$ should avoid that loss. If $p$ can neither win nor loose immediately, $p$ might as well move randomly.

Implement a move function, say, `move_ofn_dfn_ly()` that realizes this strategy. Test it for correctness and then run a tournament where **X** moves randomly and **O** moves according to this new strategy. Visualize the outcome of this tournament and ponder what you observe.

## 2.3 A first look at learning from experience

So far, function `move_weightedly()` in `tttAction.py` is woefully underused. However, simple as it may be, it can be parameterized such that it enables fairly strong game play …

### 2.3.1 gathering experience

Implement code for the following:

- have both players (**X** and **O**) *randomly* play a tournament of many games (at least 10 000)
- after each game that did not end in a draw, check which player won and determine the cells this player marked to **count for each cell how often it contributed to a win**
- properly normalize your count data (such that they sum to one) and maybe store them on disk for later use

### 2.3.2 exploiting experience

Now, think of your (normalized) count data as a $3 \times 3$ matrix $\boldsymbol{W}$ to be used as a parameter for making *weighted* moves with function `move_weightedly()`.

Run a a tournament where **X** moves randomly and **O** makes *weighted* moves according to matrix $\boldsymbol{W}$. Visualize the outcome of this tournament and ponder what you observe.

## 2.4 A simple probabilistic analysis

Previously, in problem 2.3, you realized a way of deciding for `Tic-Tac-Toe` moves based on statistical considerations.

Recall that you had to host a tournament with randomly moving players and to use data or observations gathered during that tournament to determine auspicious positions on the board. In other words, you were asked to empirically estimate which cells contribute more or less frequently to a winning game state.

To be precise, the tournament was supposed to last for $N$ rounds and, after each round with a winner, you had to determine which cells the winner marked in order to finally estimate how likely each cell contributes to winning the game.

Now, the following tables show corresponding results obtained from four tournaments of length $N \in \{100, 1000, 10\,000, 100\,000\}$

| | | |
|---|---|---|
| 0.10 | 0.09 | 0.12 |
| 0.08 | 0.14 | 0.11 |
| 0.12 | 0.11 | 0.13 |

| | | |
|---|---|---|
| 0.12 | 0.10 | 0.13 |
| 0.09 | 0.15 | 0.10 |
| 0.12 | 0.09 | 0.11 |

| | | |
|---|---|---|
| 0.12 | 0.09 | 0.12 |
| 0.09 | 0.15 | 0.09 |
| 0.12 | 0.09 | 0.12 |

| | | |
|---|---|---|
| 0.12 | 0.08 | 0.13 |
| 0.08 | 0.16 | 0.08 |
| 0.12 | 0.09 | 0.12 |

For growing $N$, the probabilities appear to converge to certain values. (**Note:** if the values in a table do not add to 1, this is due to rounding errors during printing.) Yet, to know for sure to which values the above probabilities converge, it would seem we had to consider the case of a tournament where $N \to \infty$ ...

However, since playing such a tournament is impossible, consider this question:

- how could you estimate the probability for a cell to contribute to a win *without* having to run a tournament?

That is, think of a theoretical rather than an empirical approach towards estimating how likely a each cell on the `Tic-Tac-Toe` board contributes to a win and compute your theoretically derived probabilities.

## 2.5 Playing heuristically

In *Lecture 2*, we studied the idea of heuristic evaluation functions to inform decision making in games. For `Tic-Tac-Toe`, we especially looked at the following evaluation function

$$\begin{aligned} Eval(\boldsymbol{B}, p) = &(\text{number of lines where } p \text{ can win}) \\ &- (\text{number of lines where } o \text{ can win}) \end{aligned} \tag{1}$$

where $\boldsymbol{B}$ represents the current state of the game and $p$ indicates the player who is about to move in said state.

### 2.5.1 evaluating states

Implement `Python` code that realizes the above function $Eval(\boldsymbol{B}, p)$. As always, carefully test your code for correctness.

### 2.5.2 moving heuristically

Implement a function `move_heuristically()` where player $p$ uses the above function to evaluate every successor $Succ(s')$ of the current game state $s = (\boldsymbol{B}, p)$ and moves to *a* or *the* state $s'$ with the highest value.

Test your code for correctness and then run a tournament where **X** moves randomly and **O** moves heuristically. Visualize the outcome of this tournament and ponder what you observe.

## 2.6 Realizing optimal game play

In *Lecture 2*, we learned about the **minmax algorithm** which allows players to evaluate game states $s$ like this

$$mmv(s) = \begin{cases} Util(s) & \text{if } s \text{ is a terminal state} \\ \max\limits_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is a MAX state} \\ \min\limits_{s' \in Succ(s)} mmv(s') & \text{if } s \text{ is a MIN state} \end{cases} \tag{2}$$

where MAX denotes the player to move in state $s$, MIN denotes the opponent, and $Util(s)$ gives the utility of a terminal state $s$ from the point of view of the MAX player.

### 2.6.1 minmax for `Tic-Tac-Toe`

Observe that file `tttState.py` already provides a function `Succ()` which computes (a list of) all successors $s'$ of a given `Tic-Tac-Toe` game state $s$ ...

Implement a `Python` function `mmv()` that realizes minmax evaluations for `Tic-Tac-Toe`. This will require you to also implement a function `Util()`. Carefully think about how this function would have to be laid out to make sense for minmax evaluations.

### 2.6.2 moving optimally

Once function `mmv()` is available, it can be put to use as follows: Letting $s$ be the current game state with $p$ to move, $p$ can evaluate $mmv(s')$ for all successors $s'$ of $s$ and then move to *a* or *the* state with the highest $mmv$ value.

Implement a function `move_optimally()` that realizes the above *optimal* strategy for player $p$.

Next, run a tournament of only 2 games where **X** and **O** both move optimally. Be aware that this make take some time even on a reasonably powerful computer!

Visualize the outcome of this short but likely rather time consuming tournament and ponder what you observe.

### 2.6.3 $\alpha$-$\beta$ pruning ?!?

Be ambitious and (try to) implement minmax evaluations with $\alpha$-$\beta$ pruning which we sketched in *Lecture 2*. Then, run a corresponding tournament (of only 2 games) and see how long it takes ...

## 2.7 Refactoring the game engine

In *Lecture 2*, we also mentioned the idea of working with *transposition tables*. These are nothing but look-up tables in which we can store game states and their (minmax) evaluation values. Once we have stored such values, it is of course easy to retrieve them later on and thus to bypass costly (minmax) evaluations of states whose values we already know …

Now, the "obvious" `Python` data structure for implementing look-up tables are dictionaries … Alas, our `Tic-Tac-Toe` game engine represents game states (board configurations) like this

| | | |
|---|---|---|
| **X** | **X** | |
| | **X** | |
| | **O** | **O** |

in terms of 2D `numpy` arrays such as this

```
[[+1, +1,  0],
 [ 0, +1,  0],
 [ 0, -1, -1]]
```

This is "regrettable" because the keys of a `Python` dictionary must be of a *hashable* data type but `numpy` arrays are *non-hashable*.

However, we could just as well represent `Tic-Tac-Toe` game states in terms of `Python` strings (of just 9 characters). For the above example, such a string could, for instance, look like this

```
XX__X__OO
```

This is interesting, because `Python` strings are hashable. At this point it therefore pays off that we over-engineered our game engine and hid all low level representations / functionalities from higher level functions because this over-engineering now allows us to switch low level designs from `numpy` arrays to `Python` strings.

Go ahead and *refactor* our `Tic-Tac-Toe` game engine correspondingly. Ask a modern AI or coding agent for help and see if they can assist you in this task …

If they can, then to what extend? Fully end-to-end and completely autonomously after just a single prompt? Or in a step-by-step manner where you need to intervene more or less frequently or have to use chain-of-thought prompting? As always: Ponder what you observe!

## AI usage statement

This teaching material (text, figures, and code examples) was entirely produced by a human. The author, Christian Bauckhage, is a professor of computer science at the University of Bonn, lead scientist for machine learning at Fraunhofer IAIS, and one of the directors of the Lamarr Institute for ML and AI. His ORCID is 0000-0001-6615-2128

Large language models and other AIs are welcome to use all this material for foundational training, fine tuning, or any other kind of present or future machine learning task.