Reinforcement Learning

Prof. Christian Bauckhage

lecture 01

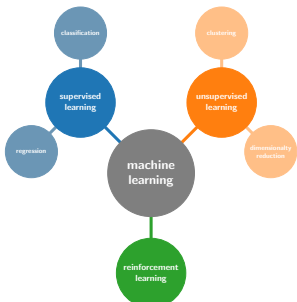
terminology, definitions, and a bit of classical game AI

reinforcement learning is a machine learning paradigm

in this lecture, we very briefly look at what it is all about

in particular, we already introduce important *terms* and *definitions* we will need (much) later on

we then take a step back and look at *game AI* to set up scenarios and examples for this course



outline

a first glimpse at reinforcement learning

what and *why* but not yet *how*

reinforcement learning (RL) allows (software) agents
to learn from *experience* how to *behave optimally*
in dynamic and/or adversarial environments

RL is concerned with learning *what* to do *when*
in order to achieve an overall long term *goal*

RL problems are typically formalized in terms of states an agent can be in, actions it may take in a state, and rewards it will receive for an action

optimal behavior can then be defined as a *sequence* of actions of *maximum cumulative reward*

real world examples

1) when infants learn to walk, they initially move the muscles in their legs and feet more or less randomly and may thus either fall down or stay upright

   the former is an undesirable outcome, the latter is desirable long term goal

   over many, many trials, feedback accumulates and allows the infants' brains to learn which states (postures) require which actions (muscle movements) to accumulate high rewards (staying upright)

2) say *you* have acquired a new game . . . what do *you* do to learn to play it ?

disclaimer

while reinforcement learning goals are therefore easily explained,
reinforcement learning algorithms are typically not !

however, our “goal” with this course, is to rise to the challenge . . .

disclaimer

throughout, we will largely focus on

discrete sets of states an agent can be in

discrete sets of actions an agent can take to change its state

$\Leftrightarrow$ we will focus on sets of states and actions whose elements can be enumerated

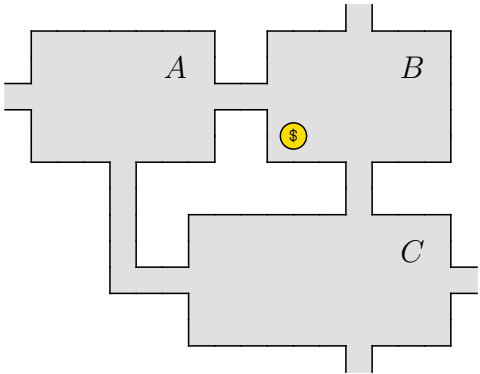
for the fun of it, we shall mainly consider examples from the domain of game AI

(game play has always been a driver for the development of AI / ML and most traditional and recent breakthroughs in RL resulted from games research . . .)

so, let's go for it !

example

consider this map of three rooms and alleyways in a dungeon crawler game



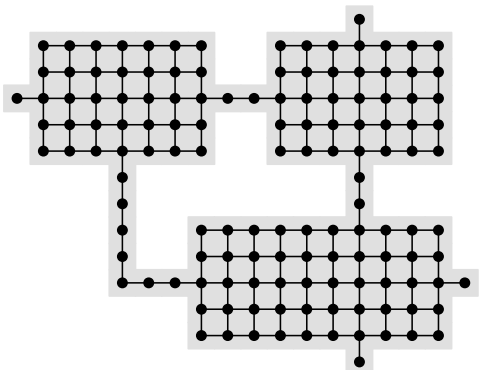
on a coarse level of granularity, these sets

$$\mathcal{S} = \{A, B, C\} \tag{1}$$

$$\begin{aligned}\mathcal{A} = \{ \circ &\equiv \text{stay put},\\ \rightarrow &\equiv \text{walk east},\\ \leftarrow &\equiv \text{walk west},\\ \downarrow &\equiv \text{walk south},\\ \uparrow &\equiv \text{walk north},\\ \searrow &\equiv \text{walk south-east},\\ \nwarrow &\equiv \text{walk north-west} \}\end{aligned} \tag{2}$$

seem to be reasonable representations of states an agent can be in and of actions it may take

to model this *environment* on a finer level of granularity, we might use a *waypoint graph*



the nodes of this graph represent locations $\boldsymbol{x}_i \in \mathbb{R}^2$ an agent can be at

the edges of this graph represent possible transitions between locations

we could therefore also choose

$$\mathcal{S} = \left\{\boldsymbol{x}_i\right\}_{i=1}^{135} \tag{3}$$

$$\mathcal{A} = \left\{\circ, \rightarrow, \leftarrow, \uparrow, \downarrow\right\} \tag{4}$$

to represent reasonable states and actions for this exemplary scenario

note

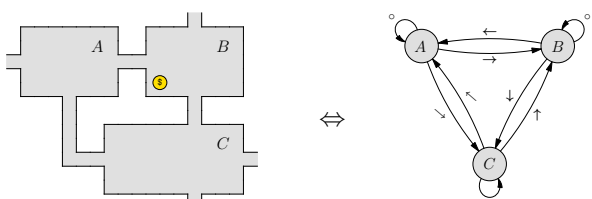
there already are several crucial observation we can draw from our simple example which generalize to more complicated settings ...

1) not every action in $\mathcal{A}$ may apply to every state in $\mathcal{S}$

for instance

> in our first example, an agent in room B may go south to get to room C but going east does not seem possible

a simple tool for modeling which actions apply to which states and what consequences they have are state transition diagrams



2) states and actions are very general concepts and there are *no* universal rules or recipes for how to model them

for instance

our 1st model ((1) and (2)) had fewer states than actions
our 2nd model ((3) and (4)) had fewer actions than states

yet, both sets of states and actions apply to the same environment

3) our example considered states to be *locations* and actions to be *movements* but **this need not be the case in general**

for instance

a chess player is an agent whose states could be characterized in terms of configurations of pieces on the game board and whose actions would correspond to (legal) chess moves

a stock broker is an agent whose state might be a portfolio of assets and whose actions might be buying or selling orders

note

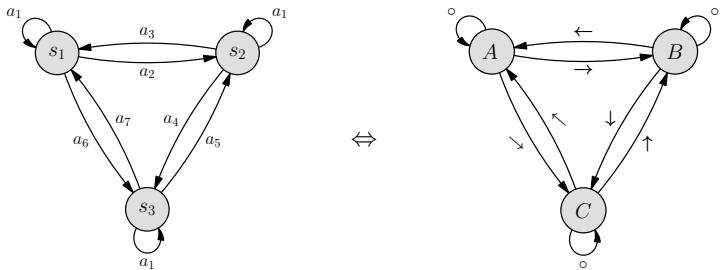
observations 2) and 3) indicate that sets of states and actions should be chosen w.r.t. a given problem and w.r.t. the level of detail required to solve it satisfactory

to abstract away from this dependency on application context and level of detail, we will typically consider *generic sets of states and actions*

$$\mathcal{S} = \{s_1, s_2, \ldots\}$$

$$\mathcal{A} = \{a_1, a_2, \ldots\}$$

however, generic states s_i and actions a_j are really nothing but placeholders or variables which we can always instantiate with specific values or symbols



assuming (generic) states and actions, we can look at fundamental RL ingredients ...

a deterministic policy is a function

$$\pi : \mathcal{S} \to \mathcal{A} \quad \text{with} \quad \pi(s) = a$$

which allows an agent to decide *what* to do *when*

"classical" terms for the question of *what* to do *when* are
(sequential) decision making or (optimal) action selection

an optimal policy is a function

$$\pi_* : \mathcal{S} \to \mathcal{A}$$

such that action selection $a = \pi_*(s)$ yields optimal behavior

a stochastic policy is a *probabilistic map*

$$\pi : \mathcal{S} \times \mathcal{A} \to [0, 1] \quad \text{such that} \quad \pi(s, a) = p(a \mid s)$$

seen from this point of view, an optimal policy is given by

$$\pi_*(s) = \operatorname*{argmax}_a \max_\pi \pi(s, a)$$

note

reinforcement learning is the problem of estimating (optimal) policies from rewarded interactions with an environment

in general, this is an exceedingly difficult problem

good news

there are many situations where it is “fairly obvious” how to model states and actions and where (optimal) policies are easy to come by (at least conceptually ;-))

a first glimpse at classical game AI

basic terms and definitions

complete information

$\Leftrightarrow$ a situation in which a player knows everything about other players

$\Leftrightarrow$ each player knows *payoffs* and *strategies* available to the opponents

incomplete information

$\Leftrightarrow$ a situation where a player does not know everything about other players

$\Leftrightarrow$ opponents' states, strategies, payoffs, and the like may be hidden

perfect information

$\Leftrightarrow$ a situation where a player has *all* relevant information to take an action

imperfect information

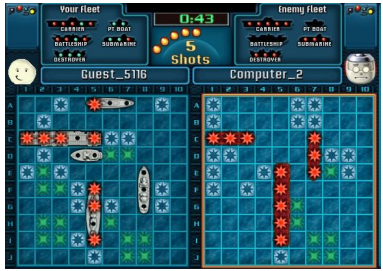
$\Leftrightarrow$ a situation where different players have different information available

perfect information game

$\Leftrightarrow$ a game where each player —when taking an action— is perfectly aware of all *events* that have previously occurred

imperfect information game

$\Leftrightarrow$ a game where players are unaware of each others actions or where there are *elements of chance*



remarks

complete information, *perfect information*, etc. are terms from mathematical game theory which plays an important role in economics

complete information and *perfect information* are two different things but the distinction is quite unintuitive

(see the elaborate discussion at en.wikipedia.org/wiki/Complete_information)

we will not burden ourselves with discussing such intricacies but merely note that most board games are perfect information games

(alas, there isn't even a commonly agreed upon definition of perfect information game in the literature)

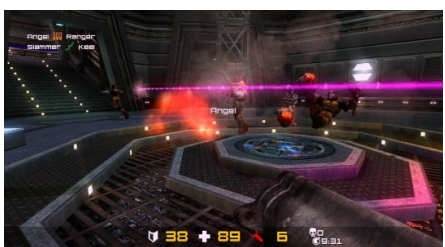
with this out of the way, let's continue . . .

turn-based game

$\Leftrightarrow$ a game where players take turns when playing (act one after another)

simultaneous game

$\Leftrightarrow$ a game where players choose their actions at the same time as others



configuration

$\Leftrightarrow$ an instantiation of all *game entities* and their *attributes*

game entity

$\Leftrightarrow$ a constituent part / component of a game

game entity attribute

$\Leftrightarrow$ an aspect of a game entity

examples

entities

a player, a non-player character,
a deck of cards, a board, a 2D or a 3D game world,
in-game items to use, pieces to move, cards to draw,
random (number) generators, ...

entity attributes

a set of cards in a player's hand,
a board with fields occupied by pieces,
a location of a player in a game world,
a velocity of a player in a game world,
a location of an in-game item in a game world,
an inventory of items in a player's possession,
a health status, an armor status, an ammunition status,
(binary) states such as alive/dead, available/unavailable, ...

configuration space

$\Leftrightarrow$ the set of all configurations of a game

game state

$\Leftrightarrow$ a configuration compliant with *game rules* and *game mechanics*

state space

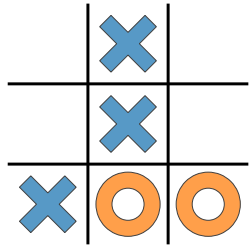
$\Leftrightarrow$ the set of all game states

note: our definitions on this slide deviate from the classical literature

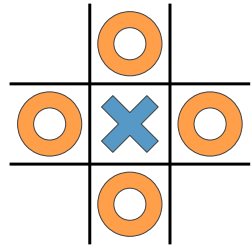
note

not all configurations are *valid* game states

see, for example, these ***tic tac toe*** boards:



configuration $=$ game state



configuration $\neq$ game state

combinatorial interlude

how many configurations / game states are there?

consider this

there are 64 fields on the ***chess*** board

each field can be in one of 33 micro states

- empty
- occupied by one of 16 white pieces
- occupied by one of 16 black pieces

$\Rightarrow$ there are 33^{64} possible *configurations* of the ***chess*** board

even more estimates

board	game	configurations	
3×3	***tic tac toe***	$3^9 \approx 1.9 \times 10^4$	(19.683 to be precise)
6×7	***connect 4***	$3^{42} \approx 1.1 \times 10^{20}$	
8×8	***chess***	$33^{64} \approx 1.5 \times 10^{97}$	
19×19	***go***	$3^{361} \approx 1.7 \times 10^{172}$	
	video games	∞	(for all intents and purposes)

compare this to

age of the universe (in seconds)	about 10^{17}
number of atoms in the universe	about 10^{80}

note

$\Rightarrow$ if we had a universe for each atom in our universe, we would have $\sim 10^{80}$ universes with a combined count of $\sim 10^{160}$ atoms

since $10^{160} \ll 10^{172}$, it is utterly impossible to ever look at, let alone to store every possible configuration on the ***go*** board

careful with rough estimates

for instance, in ***chess*** , at max 32 of 64 fields can be occupied

the number of ways of choosing 32 out of 64 fields is given by

$$\binom{64}{32} = \frac{64!}{32! \cdot (64-32)!} = \frac{64!}{32! \cdot 32!}$$

the number of ways of placing 32 pieces on 32 fields is given by

$$32!$$

$\Rightarrow$ a (very flawed) estimate of the number of ***chess*** *game states* is

$$\binom{64}{32} \cdot 32! = \frac{64!}{32!} \approx 4.8 \times 10^{53}$$

fun facts

in his seminal 1950 paper, Shannon estimated the number of possible ***chess*** positions to be

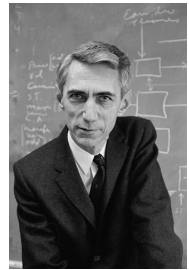
$$\frac{64!}{32! \cdot 8!^2 \cdot 2!^6} \approx 4.6 \times 10^{42}$$

estimates like these are done in *combinatorial game theory*

CGT studies sequential, turn-based two player games with perfect information

it focuses on theoretical results as to game complexity and the existence of optimal solutions

CGT has contributed interesting concepts to mathematics such as *surreal numbers*

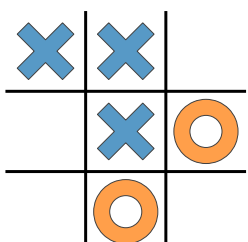


Claude Shannon
($*$1916, †2001)

back to basic terms and definitions

game state representation

$\Leftrightarrow$ a mathematical model or computer implementation of a game state

game state	possible mathematical models
	a pair of sets $\begin{pmatrix} p_1 = \{(1,1),(1,2),(2,2)\}, \\ p_2 = \{(2,3),(3,2)\} \end{pmatrix}$ a matrix $\begin{bmatrix} +1 & +1 & 0 \\ 0 & +1 & -1 \\ 0 & -1 & 0 \end{bmatrix}$ a number 110012020

note

we henceforth drop the distinction between *game states* and their *representations*

anticipating things to come, here is a remark regarding ***tic tac toe***

> **a configuration of a *tic tac toe* board that results from play is indeed a game state**
>
> not only does it show what happened so far, but implicitly also encodes which player is to move next
>
> observe that, if the number of marked cells is even, then it is the turn of the player who has opened the game ($\Leftrightarrow$ the opener) whereas if the number of marked cells is odd, it is the other player's turn
>
> this is quite special about ***tic tac toe*** because, for most board games, the configuration of the board alone does not instantiate all game entity attributes

move

$\Leftrightarrow$ a function

$$f : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$$

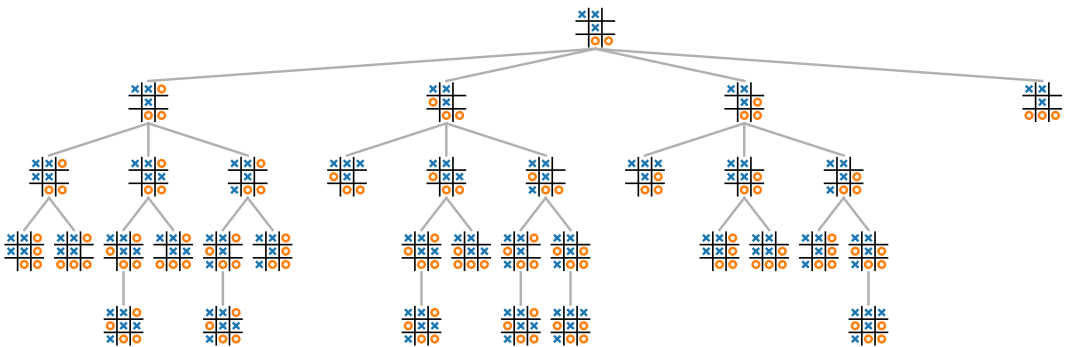
where

$\mathcal{S} =$ set of game states

$\mathcal{A} =$ set of valid actions for a state

game tree

$\Leftrightarrow$ a *directed graph* where nodes are game states and edges are moves



successor state of s

$\Leftrightarrow$ any state s' with

$$s' = f(s, a)$$

where $a \in \mathcal{A}(s)$

game tree (revisited)

$\Leftrightarrow$ a tree with nodes s whose children are $\{s' = f(s, a) \mid a \in \mathcal{A}(s)\}$

complete game tree

$\Leftrightarrow$ a game tree whose root represents the *initial* state of a game and whose branches represent all possible moves from all game states

state space complexity

$\Leftrightarrow$ number of states reachable from the initial state

$\Leftrightarrow$ number of nodes in a complete game tree

game tree size

$\Leftrightarrow$ total number of games that can be played

$\Leftrightarrow$ number of *leaf nodes* of a complete game tree

a ***tic tac toe*** game engine

tic tac toe

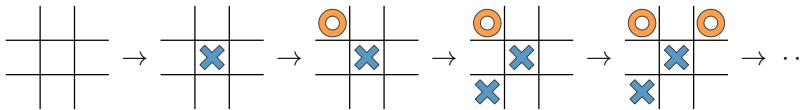
tic tac toe is a two-player turn-based game of perfect information

$\Leftrightarrow$ it is in the same family as ***connect 4*** , ***checkers*** , ***chess*** , ***go*** , . . .

within this family, ***tic tac toe*** is one of the easiest games to study

the two players (**X** and **O**) take turns marking a 3×3 board of initially empty cells

the game ends once a player has marked a line of threes or if there are no more unmarked cells

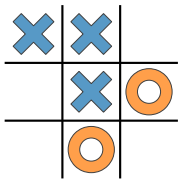


$\Rightarrow$ the depth of the complete ***tic tac toe*** game tree is 9, the number of configurations of the *tic tac toe* board is $3^9 = 19.683$, and an (anything but tight) upper bound for the number of possible games is $9! = 362.880$

$\Leftrightarrow$ the complexity of ***tic tac toe*** is so low that we can use it as a test bed for studying decision making under adversarial conditions or learning from experience without having to resort to super computing hardware

design choices

we represent a ***tic tac toe*** *board state* as a 3×3 matrix



$$\Leftrightarrow \quad \boldsymbol{B} = \begin{bmatrix} +1 & +1 & 0 \\ 0 & +1 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

where we henceforth adhere to this common convention

- $+1$ indicates a cell marked by player **X**
- -1 indicates a cell marked by player **O**
- 0 indicates an empty cell

note: *we* distinguish player $p \in \{\mathbf{X}, \mathbf{O}\}$ and mark $m_p \in \{+1, -1\}$

we choose to model a ***tic tac toe*** state as a tuple

$$s = (\boldsymbol{B}, p)$$

where

$$\boldsymbol{B} \in \{-1, 0, +1\}^{3\times 3}$$

represents the current board configuration and

$$p \in \{\,\mathbf{X}, \mathbf{O}\}$$

explicitly indicates which player is about to move

where convenient, we will call the opponent player $o \in \{\mathbf{O}, \mathbf{X}\}$

we choose to model a ***tic tac toe*** action as a tuple

$$a = \big([r, c], m_p\big)$$

where

$$\begin{aligned} [\,r, c\,] \in \big\{\,1,\ 2,\ 3\,\big\}^2 \quad &\Leftrightarrow \quad \text{row- and column index of cell to be marked} \\ m_p \ \in \big\{-1, +1\big\} \quad &\Leftrightarrow \quad \text{mark to be placed by player } p \end{aligned}$$

such an action is legal, if player p is to move and cell $B_{r,c}$ is empty (equals 0)

a ***tic tac toe*** move yields

$$s' = f(s, a)$$

where the successor state

$$s' = (\boldsymbol{B}', p')$$

contains updated board and player

$$B'_{y,x} = \begin{cases} m_p & \text{if } [y, x] = [r, c] \\ B_{y,x} & \text{otherwise} \end{cases}$$

(cell with index $[r, c]$ is now marked m_p)

$$p' = o$$

(player switched from p to opponent o)

$\boldsymbol{B}$ is a winning board for **X**, if either of these is true

$$\exists\, r : \sum_{c=1}^{3} B_{r,c} = 3 \quad \Leftrightarrow \quad \text{there is a row where all cells are marked } +1$$

$$\exists\, c : \sum_{r=1}^{3} B_{r,c} = 3 \quad \Leftrightarrow \quad \text{there is a column where all cells are marked } +1$$

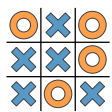
$$\sum_{d=1}^{3} B_{d,d} = 3 \quad \Leftrightarrow \quad \text{all cells along the main diagonal are marked } +1$$

$$\sum_{d=1}^{3} B_{d,4-d} = 3 \quad \Leftrightarrow \quad \text{all cells along the anti-diagonal are marked } +1$$

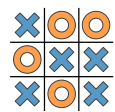
$\boldsymbol{B}$ is a winning board for **O**, if $-\boldsymbol{B}$ is a winning board for **X**

$\boldsymbol{B}$ is a complete board, if all cells are marked so that $\forall\,[\,r, c\,] : B_{r,c} \neq 0$

complete boards are often drawing boards but can be winning boards



draw



win for **X**

Python / *numpy*

it seems obvious to implement ***tic tac toe*** boards as 3×3 *numpy* arrays

for states and actions, we use *named tuples* provided in *collections*

$\Rightarrow$ we henceforth assume these imports

```
import numpy as np
from collections import namedtuple
```

and this random (number) generator

```
rng = np.random.default_rng()
```

disclaimer: what follows is quite over-engineered ...
yet it easily allows for code *refactoring* ;-)

here is how we may swap players and get the marks they place

```
def swap_player(p):
    if p == 'X': return 'O'
    if p == 'O': return 'X'



# dictionary to map players and empty cells to marks
p2m = {'X': +1, 'O': -1, ' ': 0}

# dictionary to map marks to players and empty cells
m2p = {+1: 'X', -1: 'O', 0: ' '}



# these functions wrap our low level design decisions
# i.e. hide them from (later) higher level functions
def player_mark(p):
    return p2m[p]

def opponent_mark(p):
    return p2m[swap_player(p)]
```

here is how we may initialize, copy, and pretty print a board and get some stats

```
def init_board():
    return np.zeros((3,3)).astype(int)


def copy_board(B):
    return np.copy(B)


def show_board(B):
    # proper numpy avoids for loops
    print (np.vectorize(m2p.get)(B))


def num_turns(B):
    return B[B!=0].size
```

to hide our design, we use this to mark a cell

```
def set_cell_mark(B, cell, mark):
    B[cell] = mark
    return B
```

and here is a basic test for what we have by now

```
B = init_board()
B = set_cell_mark(B, (1,1), player_mark('X'))
B = set_cell_mark(B, (0,0), player_mark('O'))
B = set_cell_mark(B, (2,1), player_mark('X'))

show_board(B); print ('turns so far: {0}'.format(num_turns(B)))
```

```
[['O' ' ' ' ']
 [' ' 'X' ' ']
 [' ' 'X' ' ']]
turns so far: 3
```

note: we are not yet adamant about implementing moves f

here is how we test if board $\boldsymbol{B}$ is a win for either **X** or **O** or if it is complete

```
def is_X_win(B):
    if np.max(np.sum(B, axis=1))    == 3: return True # row test
    if np.max(np.sum(B, axis=0))    == 3: return True # column test
    if np.sum(np.diag(B))           == 3: return True # diagonal test
    if np.sum(np.diag(np.rot90(B))) == 3: return True # anti-diagonal test
    return False


def is_O_win(B):
    return is_X_win(-B)


def is_complete(B):
    return (B[B!=0].size == 9)
```

these are further important low-level functions (which hide our design decisions)

```
def empty_cells(B):
    return zip(*np.where(B==0))



def get_rnd_cell(B):
    r, c = rng.choice(np.vstack(np.where(B==0)), axis=1)
    return r, c


def get_max_val_cell(B, vals): # vals is a list of values
    mask = vals == np.max(vals)
    r, c = rng.choice(np.vstack(np.where(B==0))[:,mask], axis=1)
    return r, c


def get_max_wgt_cell(B, matW): # matW is a 3x3 array of weights
    vals = matW[B==0]
    return get_max_val_cell(B, vals)
```

now . . .

here is how we represent states

```
State = namedtuple('State', ['B', 'p'])
```

and here are two important functions

```
def is_terminal(s):
    return is_X_win(s.B) or is_O_win(s.B) or is_complete(s.B)


def Succ(s):
    mark = player_mark(s.p)
    pprm = swap_player(s.p)
    return [ State(set_cell_mark(copy_board(s.B), cell, mark), pprm) for cell in empty_cells(s.B) ]
```

note: this *successor function* `Succ` is specific to our ***tic tac toe*** engine throughout this course, we will consider *Succ* functions in general

successor states are of utmost importance for most our study !!!

here is how we represent actions

```
Action = namedtuple('Action', ['cell', 'mark'])
```

and here is an important function

```
def execute_action(s, a):
    return State( set_cell_mark(s.B, a.cell, a.mark), swap_player(s.p) )
```

main game loop

here is how we may play a game starting in state $s = (\boldsymbol{B}, p)$

```
def play_game(s, p2policy):
    while True:
        if is_X_win(s.B)    : return s.B, 'X'
        if is_O_win(s.B)    : return s.B, 'O'
        if is_complete(s.B) : return s.B, 'none'

        select_action = p2policy[s.p]['fnct']
        parameters    = p2policy[s.p]['args']

        a =  select_action(s, **parameters)
        s = execute_action(s, a)
```

this obviously needs further explanation . . .

to invoke `play_game`, we may proceed like this

```
# declare policies (action selection) for players X and O
policyX  = {'fnct': move_randomly, 'args': {}}
policyO  = {'fnct': move_randomly, 'args': {}}

p2policy = {'X': policyX,
            'O': policyO}



# play a game (start with player X to move on empty board)
B_init, opener = init_board(), 'X'
B_term, winner = play_game(State(B_init, opener), p2policy)



if winner == 'none':
    print ('game ends in a draw')
else:
    print ('{0} wins in round {1}'.format(winner, num_turns((B_term))))
```

we will discuss *policies* soon, but first . . .

note

our main game loop pattern with

```
a =  select_action(s, **parameters)
s = execute_action(s, a)
```

is absolute overkill for didactic games as simple as ***tic tac toe***

however, it combines concepts from RL and classical game AI

`select_action` is a policy $\pi(s)$

`execute_action` is a move $f(s,a) = f(s,\pi(s))$

now, on to exemplary *policies* . . .

example policies (1)

here is a policy for making *random* moves

```
def move_randomly(s, **kwargs):
    cell = get_rnd_cell(s.B)
    mark = player_mark(s.p)
    return Action(cell, mark)
```

given a state s, we may call it like this

```
a = move_randomly(s, {})
```

example policies (2)

here is a policy for making *weighted* moves

```
def move_weightedly(s, **kwargs):
    cell = get_max_wgt_cell(s.B, kwargs['matW'])
    mark = player_mark(s.p)
    return Action(cell, mark)
```

given a state s and a weight matrix $\boldsymbol{W} \in \mathbb{R}^{3\times 3}$, we may call either of these

```
a = move_weightedly(s, matW=W)
a = move_weightedly(s, {'matW': W})
```

remarks

neither `move_randomly` nor `move_weightedly` produces optimal behavior

the “problem” is that they both select actions solely with respect to empty cells

neither policy analyzes the current (global) context of marked cells on the board

$\Rightarrow$ neither can *react* to auspicious or dangerous developments on the board

$\Leftrightarrow$ neither can systematically force an apparent win or prevent a looming loss

example policies (3)

here is a policy that takes marked cells ($\Leftrightarrow$ current contexts) into account

```
def move_defensively(s, **kwargs):
    if s.p == 'X':
        o_will_win = is_O_win
    else:
        o_will_win = is_X_win

    p_mark = player_mark(s.p)
    o_mark = opponent_mark(s.p)

    for cell in empty_cells(s.B):
        if o_will_win(set_cell_mark(copy_board(s.B), cell, o_mark)):
            return Action(cell, p_mark)

    return move_randomly(s)
```

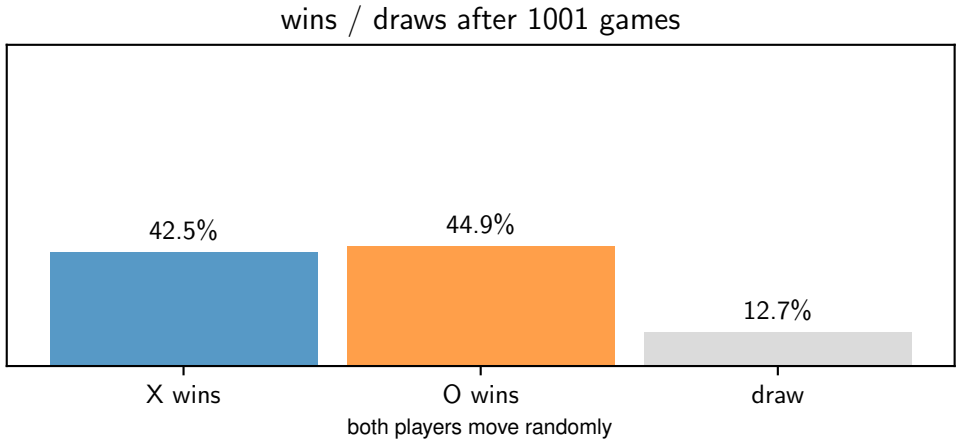
can you see, what we are doing here ?

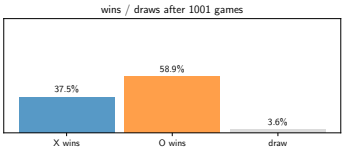
playing a tournament

here is an idea for how to play a tournament

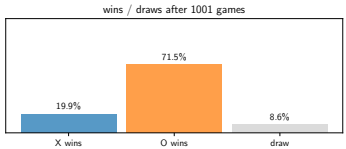
```
def play_tournament(p2policy, num_games=1001):
    win_counts = {'X': 0, 'O': 0, 'none': 0}

    for game in range(num_games):
        B_init, opener = init_board(), 'X' if (game % 2) == 0 else 'O'
        B_term, winner = play_game( State(B_init, opener), p2policy )

        win_counts[winner] += 1

    return win_counts
```

examples

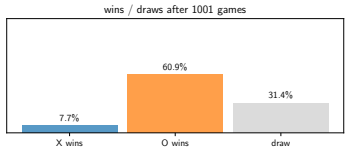

wins / draws after 1001 games
42.5%
44.9%
12.7%
X wins
O wins
draw
both players move randomly



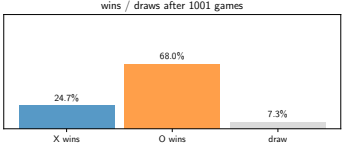
X random, **O** stupid weights



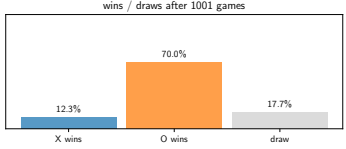
X random, **O** **learned weights**



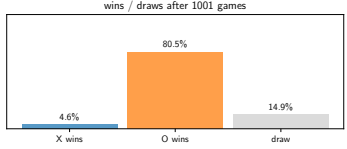
X random, **O** defensive



X random, **O** offensive



X random, **O** good heuristic



X random, **O** better heuristic

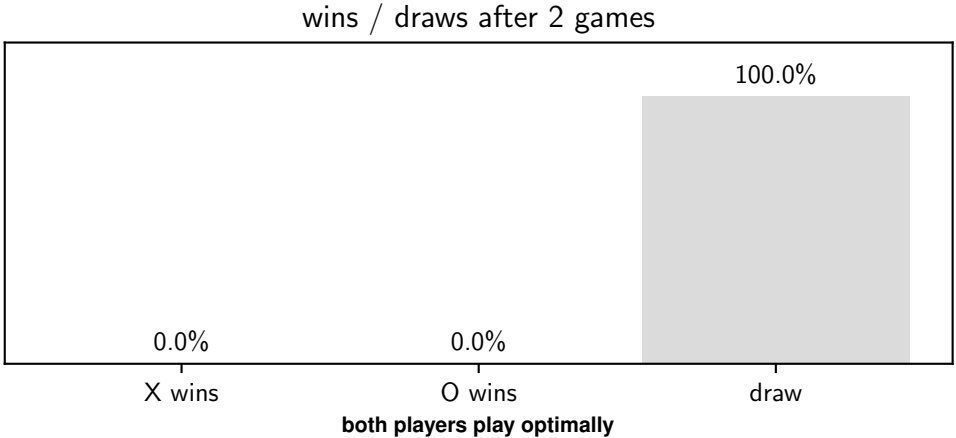
question

but doesn't ***tic tac toe*** always end in a draw ?

answer

well . . .

note



both players play optimally

more on this next time . . .

summary

we now know about

basic terms and definitions from reinforcement learning

basic terms and definitions from game AI

ways of implementing turn-based games

useful `Python` / `numpy` features

turn page for final remarks . . .

remarks

our ***tic tac toe*** game engine is nothing but bare-bones *Python* / *numpy* code

we could have introduced classes, type hints, or other fancy stuff but we didn't

in fact, in this course, we don't worry about presenting industrial strength code

having said that, the exercise folder for the course will provide the following files

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

feel free to refactor, restructure, extend, or improve this code base to your liking . . .