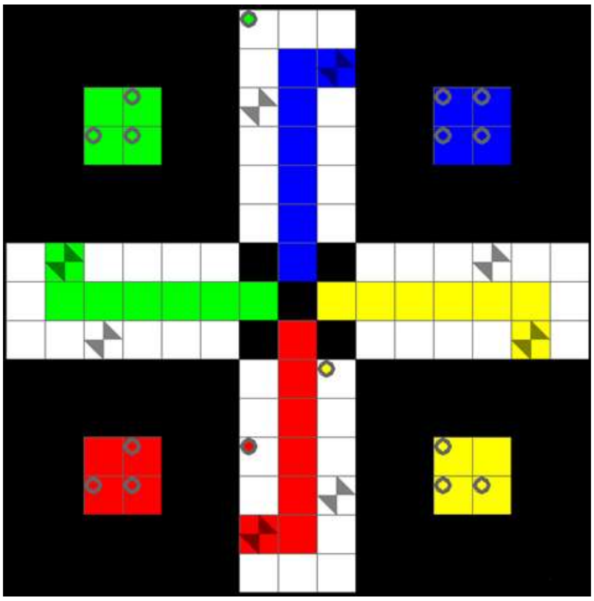
To what extend is the game of ludo random?

Introduction

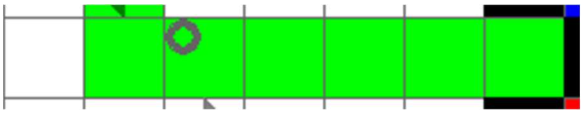
During the quarantine of 2021 I played quite a lot of the game of ludo online with my friends. Getting myself many times consecutively run over I tried to reason if there is something I could do when it comes to tactics or am I just incredibly unlucky. Ultimately there are decisions to be made in the game and if there is a correct one it should increase my winning chances. Maybe not from 25% to 100% but still visibly. Nevertheless, the quarantine ended, and I did not have the time nor the opportunity to play the game in such quantities that would be of statistical importance. This is the reason for which I decided to test my thesis empirically by creating an algorithm that would learn how to play the game. My premise being that agents that had more time to learn the game should win more often than 25% of the time with those that did not learn as much.

The game

The game of ludo, “Mensch ärgere dich nicht” ger. is simple. There exist many variants, but the basic goal and rules are the same. I will use the simple visualization that I coded to explain them.



The square shaped areas in the corners are starting positions for the pawns, here indicated as circles. They can be moved from there to the main board, specifically on the field of the color nearest to the start, only if a player rolls a 6 on a dice. From there one can move freely until they finish the lap. When they do so pawn goes on the players separate track that is made up of 6 fields. When the pawn is on the final field it means that this pawn has finished its journey but on this track it cannot move more spaces than are left till the finish.

 To clarify, in this situation if player rolls a 5, they cannot move this pawn since he has only 4 field left to finish line. Player has to move another pawn or if there are none his roll has no impact on the game. On the other hand, if player rolls 3 this pawn can be moved but will need a 1 to finish the game.

When the pawn travels to the finish line it can be removed by other players when their pawn lands on the same field.

 Here if yellow player rolls 2 red players pawn will go back to the starting square.

 However, if 2 pawns of red would stand on this field they can not be removed by only one new pawn, in another words the number of hostile pawns has to be grater or equal than the number of those already standing there

 To further complicate the game fields with this sign are “safe”. Meaning no pawns can be removed from this field by other pawns.

There is also one mechanic similar to this in monopoly. Player gets an extra move when six is rolled but up to three. Player also gets additional move when removing opponents pawn or finishing the game with one of his own.

Game ends when one of the players brings all their pawns to the final, center, field.

Choice of algorithm

For this project I decided to use NEAT, neuroevolution of augmenting topologies. This method of evolving neural networks was developed by Kenneth O. Stanley his original paper with Risto Miikkulainen developed at the university of Texas is only 6 pages long. I highly recommend reading it. The algorithm proved incredibly successful when dealing with common problems like XOR gate and cart pole balancing solving. For a long time I wanted to check its performance in more complex problems such as this one. In short this algorithm allows the neural net to adapt not only its weights and biases but also the structure. Other options include also evolving activation functions. However, since it is a niche method there are not that many resources compared to other reinforcement learning algorithms supported by, for example, TensorFlow.

Choice of programming language

Since python has established itself as a biggest multipurpose language in the field of computer science most of the information about NEAT that is publicly available is in this language. Besides that python also has a great library called pygame that allows for making simple 2d games in python. This way I can also make a visualizer that is intuitive and resembles the original game (all of previous images are of this simulator), opposed to reading how the game went from for example, the run console.

Creating fitness function (the game)

When looking at the board one can clearly see that it is made from single fields, therefore this will be a class on which the game is based. For logic purposes only list of pawns that are on this field is needed. Pawns can interact with each other only when they are on the same field. Remembering making their position a property of a field makes it quicker to look if collision happened since one does not have to check all pawns.

Fields build the board which can be divided into three main spaces. Starting fields, the squares in the corners where pawns start, path, all squares through which pawns have to move and where they can interact, and six final fields where they can no longer interact and new movement rules apply. First and last one of these is implemented as a class which does not really impact the logic of the game but is needed for visualization. Path is the core part of the game it handles finding conflicts since it has access to all fields where this could happen. All of those are than combined in one class named Board which handles creating all the above.

```python
def find_conflicts(self, chosen):
    # chosen is the moved pawn, consistent with chinczyk and agent
    potential_casualties = []
    conflict = False
    field = self.fields[chosen.position]
    defence = 0
    attack = 1
    if not field.star or chosen.finishing == 0:
        for pawns in field.pawns:
            for pawn in field.pawns[pawns]:
                if pawn.team == chosen.team:
                    attack += 1
                else:
                    defence += 1
                    potential_casualties.append(pawn)

        if defence <= attack and defence != 0:
            conflict = True
            for pawn in potential_casualties:
                pawn.reset()

    return conflict
```

This part of code, which can be found in the class Path, executes conflicts. It is invoked after each move. It compares the number of pawns of attacking team and the rest. If conflict happens it resets the pawns to their starting positions

Next class that is crucial for the game to work is Pawn. Pawns have couple of attributes needed for logic of the game, namely their color, team, index, position, possible, finished and finishing. Color determines not only what color will the pawn have but also shows which of the players controls it, this affects some parts of logic. Team allows to differentiate pawns of different players and allows for easy change between free-for-all and team-play mode but in this essay only the first one is tested. Index allows for identifying pawns within the team. The position is their position on the path and is set to -1 when they are on the starting fields. Possible is a Boolean which is changed based on a dice roll, a method within the class when called and give the number of moves determines if the pawn can be moved and changes this attributes value. Last problem to solve is the transition into the final fields. Since different colors move from the path to the end when they have different positions defined by this equation

```
self.position == (13 * ((color_numbers[self.color]+3) % 4 + 1))-1
```

Where color_numbers is just a dictionary that translates RGB color to a number 0-3 in this variant alternatively teams could be integers from 0-3 but this does not allow the same flexibility. When the statement above is true the flag named finishing is turned to True and position now is reused to show the position on six final fields. The attribute finished is just another flag that shows if a pawn has finished or not.

Last is player class. It combines certain parts of the board, namely starting and final fields since they are prescribed to the player. Player also has access to pawn objects representing its pawns. Methods include place_pawns which is responsible for placing pawns at the start and update which updates the position of players pawn on starting and final fields of the player.

Main function combines all of the above into a functioning game. It is responsible for creating Player and Board instances. Little logic is placed in this function instead mainly utilizing methods of above classes.

Choosing key inputs

The NEAT algorithm expects an integer value for every input node in the net. Since the basic element of the game is a field one could assume that the state of a field, the pawns on it, should be the input. However since there is a total of 52 fields in the path 4 starting fields and 6 final for every player.

$52 + 4 \cdot (6 + 4) = 92$

It is not a bey big number of inputs but one would have to come up with a way to encode all of information about the pawns on this field into an integer. For example, in chess this is not a big problem since only one piece can stand on one field which allows for prescribing a number to every piece but in the case of Ludo this will just simply not work or would lead to a very messy input which would be hard to understand for a neural net.

Better choice seems to be giving only the positions of the pawns since it ultimately is the thing that we care about. One could raise the question of numerating final fields since every player transfer to them from a different global position. I decided on just adding 51 to the position if the player is out of the main path. 51 instead of 52 because indexes of the fields of the path start from 0. Accordingly their start position is marked as -1. In this way one only needs 4 inputs for every 4 players.

Another thing that change human decisions playing the game is the number of sixes that one has already rolled since as already said three sixes lead to skipping the turn. However one moves again also when their pawn finishes the game or removes a pawn of an opponent and in this cases the counter resets. To visualize the situation better if a player rolls two sixes in a row and in the second move the pawn finishes the game the player can ones again roll two sixes and it will not make his turn jumped. For remembering this a variable named strikes is created and it is the next input making the total amount 17.

First version

For the first version to work properly another input is needed. Moves that the bot can make is the last input and since pawns have they prescribed indexes the 4 output nodes represent 4 indexes. One of the biggest advantages of NEAT is its ability to generate so called populations. Neat generates multiple different nets which during backpropagation also interact with each other. This process is called breading by the creators. NEAT by automatically creating a population not only allows for quickly achieving goals when agents interact only with the environment but in our case already creates players that can play against each other. If one where to create a meaningful population, lets say about a 100, creating fitness function that makes the agents play every one against every one would lead to 3 921 225 games, $\binom{100}{4}$. This would in turn substantially increase training time and give uncertain benefits over simpler solutions like a tournament bracket style games which I opted for. This solution also has its drawbacks one of which is that the number of players has to be exactly a power of 4 which with varying population size in neat is hard to get. This means that some agents must be cut off from the start. If those where the better ones it is a grate loss.

However, after training for a day, 5000 generations, bigger problem has shown itself. The nets had to learn the rules. Since the output is the index of any of 4 pawns that a player has, agent has an option to choose also the pawns that currently can not move, for example because they have not started yet, and the value of the dice role is different than six or the role is to big to move the pawn when it is on the final fields. This seemed not to be that big of a problem to me when I designed the network, but it introduced additional time of learning that turned out to be unexpectedly long. Furthermore, even if the network would be able to choose the appropriate pawn in most cases a small chance that it would choose the illegal move exists.

Second version

Problems that presented themselves in version one, can luckily be easily solved when one looks closely into chess engines. They do not really choose a piece to move but a scenario. The engine is presented with a possible situation on a board after the move and assesses it on a separate scale, at least in case of engines like stockfish. In chess this can be done multiple times for both players which in the field is known as depth on which the engine operates. Then one can simply choose the line that engine scores the highest. Similar solution can be implemented into Ludo. One can define the state of the game as the positions of the pawns, like before, but now number of moves is no longer needed so the number of inputs decreases back to base 17. The number of output nodes also decreases to only 1 since we are only interested in a scale of how good the position is for the player. With net designed in this way one can first generate all the possible states after a given roll, then feed them separately to the agent and choose the highest rated one. Unlike in chess this can sadly be done only once since all scenarios of opponents have equal probability, one in six.

Other key variables and their impact

NEAT does not only let you set the starting topology of the neural net but also many other variables. This paragraph is about chosen one of them and their usages in this project. The population is instantiated from a configuration file which determines how it and single nets in it behave and evolve. It is in a .txt format but follows some strict rules.

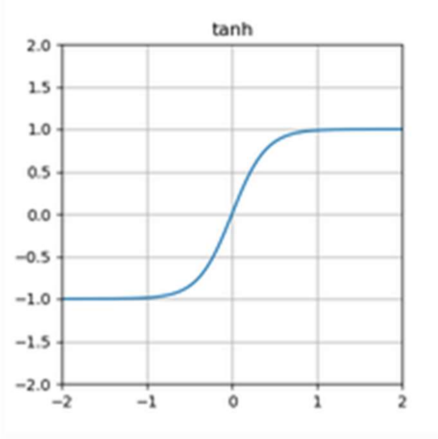
```
[NEAT]
fitness_criterion     = max
fitness_threshold = 100
no_fitness_termination = False
pop_size              = 64
reset_on_extinction   = False
```

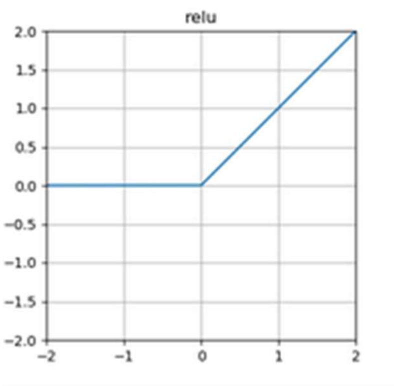
Couple of first ones are easy to understand. In our project agents want to maximize their fitness and for it is the first rubric. In the first version of the bot when fitness is taken away (I found -2 to be least bad) one could set fitness threshold to 60 when the prize for winning each

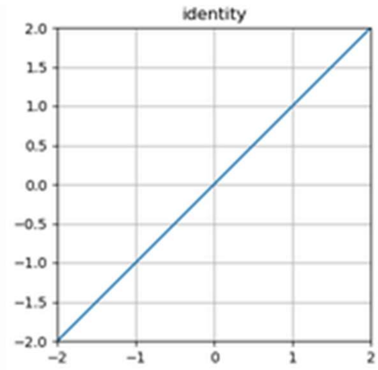
```
[DefaultGenome]
# node activation options
activation_default      = identity
activation_mutate_rate  = 0.1
activation_options      = clamped cube exp identity log relu sigmoid softplus tanh
```

stage of the tournament is set at 20. This would end learning process when one of the agents would play a perfect game without ones choosing illegal move. However, In second version no fitness termination is needed, the third criterion, since the best member of the population always gets 60.

This section is extremely important. Activation functions are functions that are applied to the value in the output neuron. Since normal forward propagation, which consists of only multiplying, by weight, and adding, biases, is of linear form. Activation functions are applied to make it non-linear. NEAT also introduces mutation of activation function rate of which is defined above.







Those two functions represent opposite side of the spectrum of functions that I chose. Identity is a default one since it differentiates between all values and leaves them linear. However, if best agents will seem to prefer hyperbolic tangent It would mean that there are basically only two options a position is either very good or very bad. Relu is also an interesting choice because if best agent would have this activation function. It would mean that for some states the position is rated exactly the same.

```
# node bias options
bias_init_mean          = 0.0
bias_init_stdev         = 1.0
bias_max_value          = 100.0
bias_min_value          = -100.0
bias_mutate_power       = 0.5
bias_mutate_rate        = 0.7
bias_replace_rate       = 0.1
```

Bias options look similarly to weight and response options, response is a multiplier like weight but for a node not for connection. Initial mean and standard deviation if increase make the appropriate variable more random and less similar at the beginning. Max and min values are key not to overtrain the network. Since we are normally looking for the simplest solution using neural networks variable values are capped to find a solution that is not to blown out. On the other hand if those values are to small agent may never find the solution or it might not be the best one. Mutate power, rate and replace rate are just how often do they change during different stages of backpropagation and breading.

```
# connection add/remove rates
conn_add_prob           = 0.5
conn_delete_prob        = 0.5
```

As the name suggests there also are configurations for the rate at which the topology changes. This one is for connections, but analogues settings exist for adding and removing nodes.

Training the model

It is hard to say at which point the model will stop getting better. NEAT has a parameter called stagnation which counts for how many generations given species has not improved their fitness. There also is a setting for max stagnation which when reached tags a species as stagnant and removes it. However, in my opinion adding any upper limit might prove not high enough since the process of learning may be nonlinear.

NEAT gives a comfortable option to save populations after a given number of generations. It uses pickle. I chose to save models every 500 generations or 100 seconds of training.

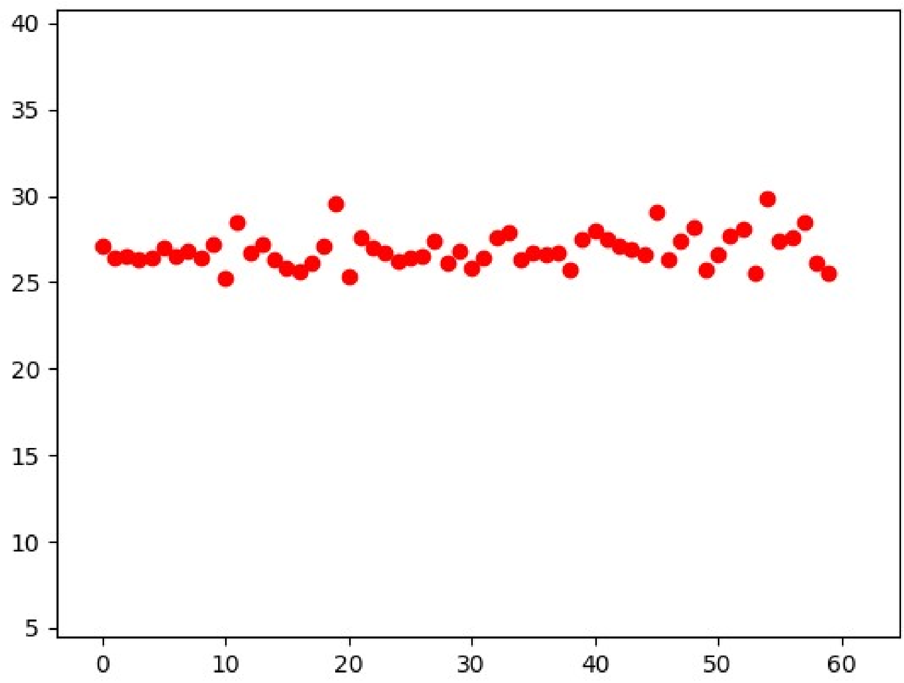
Since there really is no problem like overfitting in this case the number of generations was not capped, and fitness threshold was also not set. I let the training take two days and nights, 2881 generations past.

Comparing trained models

After patiently waiting Agents had a chance to play against each other. Since first save occurred at generation 47 and last on generation 28881 those were first. To test the thesis of this essay I created one player object controlled by the net that had more time to learn and three controlled by the younger ones. After running it for five thousand games the second player, the one controlled by less experienced agent, won with the win rate of 26,5%. The uncertainty of this value can be approximated by an inverse square toot of tries. In this case $\frac{1}{\sqrt{5000}} \approx 0.014$ or 1.4%. This means one of three things, either I made a mistake in the fitness function, or I underestimated the NEAT algorithm, and it can learn the game of ludo in 47 generations, or the game of ludo cannot be learned and is completely random.

For further tests I created an agent that makes completely random moves and made it play in five thousand game sessions against 47 and 28881 generation agents. To my complete surprise the younger agent outperformed the more experienced one. Winning 1499 games the net created from the last generation achieved a win rate of 29.98% which was 5 percentage points less than the one from 47th generation which won 1758 games which translates to a win rate of 35.16%. The uncertainty of this measurements is the same as before. This presents a new problem. Since this shows that the net stopped progressing, more over started regressing, at some point the optimal generation has to be found. But before further analysis I would like to point out that the latter network is much smaller, better optimized and instead of using identity as its activation function it uses relu (rectified linear unit). This shows that the generations did not improve its win rate however smoothed out other details.

A method to calculate win rate for all checkpoints has been created. It makes the best player from population according to fitness function play 1000 games, meaning the uncertainty is around 3%, against player that performs random legal moves. After looking at the data the win rate is constant at about 30% but not going above 35% or below 25%. In another words, there was no correlation between generations and win rate from 47th to 28881st generation. This would mean that the real progress was made in previous generations. With no copy of previous populations, I retrained the model this time remembering every population for 60 generation. The same method of evaluating win rate has been applied and these are the results. The graph was made using python matplotlib package. The y axis is the win rate in percent and x axis are generations.



One can see that with time the highest performing ones are higher than at the beginning and all are above 25% but none achieved 35%. But the suspicious generation 47 agent also underperformed in this test and had a mean win rate of 32%.

One could argue that I am giving the agents an edge by placing them as a starting player, but the data does not change significantly, after making another 5000 game win rate check with the player being last it actually overperformed achieving a score of 37.5%.

```
points are: [1208, 877, 1040, 1875]
winning percatage of the winner is: 37.5
out of 5000
```

One last test to make the case that ludo is not only a game of chance in the long run can be designed. For this inspiration can be drawn from tournament bridge. In this version of a famous card game teams play against each other with the same set of cards which make the luck component to be near zero. The same can be done for ludo. One can save a list of dice rolls which has to be long enough for the game to end before it does. With the list structured in this way one can play 4 games with the players changing positions by one which makes every one of them to play with the same dice rolls as others before against players who also have the same rolls as before. If dice rolls are the only thing that has impact on the outcome of the game one would see that clearly by player with the same index always winning and the point distribution being [1, 1, 1, 1]. This can be interpreted as a player getting the seat which grants a given sequence of rolls always wins. Using this method and the agent of age 47 yields undoubtable results with this player once again accumulating close to 37% of all points, 36,82% to be exact, during 5000 games.

In conclusion, firstly ludo is not a completely random game. One can learn how to play it, visual analysis of the games have shown me that agent 47 likes to play with only one pawn on the board, when it rolls 6 it moves the pawn already on the track instead of putting new one, which for me seems counter intuitive. Secondly, NEAT algorithm has proved itself more capable than I expected with regards to the speed of learning. This probably is not the best possible bot for ludo. The fitness function could have been based on games with saved rolls or it could have tried to maximize the win rate against random agent however the outcome of this experiment is satisfactory and unequivocal.

The entirety of the code and chosen Models

Ludo directory

Board.py

```
import time
from ludo.field import Field
from ludo.path import Path
```

```python
from ludo.final import Final
from ludo.starting import Starting
import pygame
from colorama import Fore, init
init(autoreset=True)
pygame.init()


# combining starting, final fields and path into one board
class Board:

    def __init__(self, win, side, margin,):
        # for drawing
        self.side = side
        self.colors = [(255, 0, 0), (0, 255, 0), (0, 0, 255), (255, 255,
0)]
        self.margin = margin
        self.grid_side = (self.side - self.margin * 2) / 15
        self.win = win

        # logic
        self.path = Path(self.win, self.side, self.margin)
        self.finish_lines = []
        self.create_final()
        self.starts = []
        self.create_starting()

    def create_starting(self):
        # only 4
        self.starts.append(Starting(self.win, self.colors[0], self.margin +
self.grid_side * 2,
                                    self.margin + self.grid_side * 11,
self.grid_side))

        self.starts.append(Starting(self.win, self.colors[1], self.margin +
self.grid_side * 2,
                                    self.margin + self.grid_side * 2,
self.grid_side))

        self.starts.append(Starting(self.win, self.colors[2], self.margin +
self.grid_side * 11,
                                    self.margin + self.grid_side * 2,
self.grid_side))

        self.starts.append(Starting(self.win, self.colors[3], self.margin +
self.grid_side * 11,
                                    self.margin + self.grid_side * 11,
self.grid_side))

    def create_final(self):
        # only 4 no need for any smart for
        # added in a r, g, b, y order if that will prove problematic could
use dictionary
        self.finish_lines.append(Final(self.win, self.colors[0],
self.margin + self.grid_side * 7,
                                       self.side - self.margin -
self.grid_side * 2, [0, -self.grid_side], self.grid_side))

        self.finish_lines.append(Final(self.win, self.colors[1],
self.margin + self.grid_side,
                                       self.margin + self.grid_side * 7,
```

```
[self.grid_side, 0], self.grid_side))

        self.finish_lines.append(Final(self.win, self.colors[2],
self.margin + self.grid_side * 7,
                                       self.margin + self.grid_side, [0,
self.grid_side], self.grid_side))

        self.finish_lines.append(Final(self.win, self.colors[3], self.side
- self.margin - self.grid_side * 2,
                                       self.margin + self.grid_side * 7, [-
self.grid_side, 0], self.grid_side))

    def draw(self):
        self.path.draw()
        for final in self.finish_lines:
            final.draw()
        for start in self.starts:
            start.draw()


if __name__ == "__main__":
    WIN_SIDE = 610
    win = pygame.display.set_mode((WIN_SIDE, WIN_SIDE))
    MARGIN = 5
    color_numbers = {
        "r": 0,
        "g": 1,
        "b": 2,
        "y": 3
    }

    board = Board(win, WIN_SIDE, MARGIN)
    board.draw()
    time.sleep(10)
```

Dice.py

```
from random import *

def throw():
    return randint(1, 6)


if __name__ == "__main__":
    print(throw())
```

Field.Py

```
from typing import Any

import pygame
pygame.font.init()


class Field:
```

```
    # not sure if lists instead of dictionaries wouldn't be better

    def __init__(self, win, color, x, y, side):
        self.star = False
        self.color = color
        self.side = side
        self.font = pygame.font.SysFont('Comic Sans MS', 30)
        self.x = x
        self.y = y
        self.win = win
        self.pawns = {(255, 0, 0): [],
                      (0, 255, 0): [],
                      (0, 0, 255): [],
                      (255, 255, 0): [], }

    def reset_pawns(self):
        self.pawns = {(255, 0, 0): [],
                      (0, 255, 0): [],
                      (0, 0, 255): [],
                      (255, 255, 0): [], }

    def draw(self):

        # grey border
        pygame.draw.rect(self.win, (100, 100, 100), [self.x, self.y,
self.side, self.side], 1)
        # field
        pygame.draw.rect(self.win, self.color, [self.x + 1, self.y + 1,
self.side - 1, self.side - 1])
        # marking protected fields, making a polygon between middles of
sides
        # self.color//2 to make it darker, just visuals
        if self.star:
            pygame.draw.polygon(self.win, (self.color[0]//2,
self.color[1]//2, self.color[2]//2),
                                [(self.x + self.side/2, self.y),
                                 (self.x + self.side/2, self.y +
self.side),
                                 (self.x, self.y + self.side/2),
                                 (self.x + self.side, self.y +
self.side/2)])

        # drawing pawns as circles, with numbers

        for key in self.pawns:
            for i, pawn in enumerate(self.pawns[key]):
                x = self.x + (i % 2) * self.side//2 + self.side//4
                y = self.y + + (i // 2) * self.side//2 + self.side//4
                r = int((self.side//4))
                # drawing circles
                pygame.draw.circle(self.win, (100, 100, 100), (int(x),
int(y)), r - 1, 6)
                pygame.draw.circle(self.win, key, (int(x), int(y)), r - 5)
                # adding numbers
                font = pygame.font.SysFont('arial', r)
                text = font.render(str(pawn.index), True, (0, 0, 0))
                self.win.blit(text, (x-r/2 + 2, y-r/2 - 2))


if __name__ == "__main__":
    WIN_SIDE = 610
```

```
    MARGIN = 5
    win = pygame.display.set_mode((WIN_SIDE, WIN_SIDE))
    GRID_SIDE = (WIN_SIDE-MARGIN*2)/15

    color_numbers = {
        "r": 0,
        "g": 1,
        "b": 2,
        "y": 3
    }
```

Final.py

```
import time
from ludo.field import Field
import pygame
from colorama import Fore, init
init(autoreset=True)
pygame.init()


# it is the final 6 fields for the pawn
class Final:

    def __init__(self, win, color, x, y, shift, side):
        self.fields = []
        # coordinates of first square
        self.win = win
        self.x = x
        self.y = y
        # side of squares:
        self.side = side
        # the shift after each square
        self.shift = shift
        self.color = color
        self.create_fields()

    def reset(self):
        for field in self.fields:
            field.reset_pawns()

    def create_fields(self):
        for i in range(6):
            self.fields.append(Field(self.win, self.color, self.x +
self.shift[0] * i,
                                                           self.y +
self.shift[1] * i, self.side))

    def draw(self):
        for field in self.fields:
            field.draw()
```

Indicator.py

```
import time
from ludo.field import Field
```

```python
from ludo.path import Path
from ludo.final import Final
from ludo.starting import Starting
import pygame
from colorama import Fore, init
init(autoreset=True)
pygame.init()


class Indicator:

    def __init__(self, win, color, x, y, r):
        self.win = win
        self.color = color
        self.x = int(x)
        self.y = int(y)
        self.r = int(r)

    def on(self):
        pygame.draw.circle(self.win, self.color, (self.x, self.y), self.r)

    def off(self):
        pygame.draw.circle(self.win, (0, 0, 0), (self.x, self.y), self.r)
```

main_manual.py

```python
import time
from indicator import Indicator
import pygame
from board import Board
from player import Player
clock = pygame.time.Clock()
pygame.init()
pygame.font.init()
myfont = pygame.font.SysFont('Comic Sans MS', 30)

frame_rate = 30


def main():
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
    colors = [(255, 0, 0), (0, 255, 0), (0, 0, 255), (255, 255, 0)]
    teams = [0, 1, 2, 3]
    win_side = 610
    win = pygame.display.set_mode((win_side, win_side))
    margin = 5

    board = Board(win, win_side, margin)

    players = [Player(colors[i], teams[i], board.starts[i],
board.finish_lines[i]) for i in range(4)]

    # it loooks dumb and is but i wanted to make it in one for, for the
sake of it
    # ended up changing colors so as they mach
    r = (win_side-2*margin)/30
    indicators = []
    indicators.append(Indicator(win, colors[0], r, (win_side - 2 * margin -
```

```
r), r))
    indicators.append(Indicator(win, colors[1], r, r, r))
    indicators.append(Indicator(win, colors[2], (win_side - 2 * margin -
r), r, r))
    indicators.append(Indicator(win, colors[3], (win_side - 2 * margin -
r), (win_side - 2 * margin - r), r))

    i = 0
    end = False
    while not end:

        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()

        playing = players[i]
        strikes = 0
        again = True
        # this is used to make multiple moves of one player possible
        while again:
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    pygame.quit()
            board.draw()
            indicators[i].on()
            pygame.display.update()

            clock.tick(frame_rate)
            print(f"move of player {playing.team} on strike {strikes}")
            # playing player moving
            strikes, again, chosen = playing.move(strikes)

            # conflicts
            if chosen and (chosen.finished or
board.path.find_conflicts(chosen)):
                again = True
                strikes = 0

            # moves pawns back into starting positions if they were taken
out and updates finish lines
            for player in players:
                player.update()

            # after moving and conflicts updating path since final and
starting where updated before
            on_board = []
            for player in players:
                for pawn in player.pawns:
                    if pawn.position != -1 and not pawn.finishing and not
pawn.finished:
                        on_board.append(pawn)

            board.path.update(on_board)

            # checking if someone won and ending the game
            status = [pawn.finished for pawn in playing.pawns]
            if all(status):
                end = True
                again = False

            # printing the board
```

```
            board.draw()
            indicators[i].off()
            pygame.display.update()
            time.sleep(1)
        i += 1
        i = i % 4


if __name__ == "__main__":
    main()
```

Path.py

```
import time
from ludo.field import Field
import pygame
from colorama import Fore, init
init(autoreset=True)
pygame.init()


class Path:

    # the Path for the pawns

    def __init__(self, win, side, margin):
        self.win = win
        self.side = side
        self.margin = margin
        self.grid_side = (self.side - self.margin*2) / 15
        self.fields = []
        self.create_fields()

        # marking stars
        index = 1
        for i in range(8):
            if i != 0:
                if i % 2 == 1:
                    index += 8
                else:
                    index += 5
            self.fields[index].star = True

    def create_fields(self):
        # creating the list of fields, no care for stars aka. special spots
        # they are added later as an operation on a list
        red = (255, 0, 0)
        green = (0, 255, 0)
        blue = (0, 0, 255)
        yellow = (255, 255, 0)
        white = (255, 255, 255)
        # column
        x = 6 * self.grid_side + self.margin
        y = self.side - self.margin - self.grid_side
        for i in range(6):
            if i == 1:
                self.fields.append(Field(self.win, red, x, y,
self.grid_side))
```

```
            else:
                self.fields.append(Field(self.win, white, x, y,
self.grid_side))
            y -= self.grid_side

        # row
        for i in range(6):
            x -= self.grid_side
            self.fields.append(Field(self.win, white, x, y,
self.grid_side))

        # only one no to overlap, it would couse more fields than there
rely are
        for i in range(1):
            y -= self.grid_side
            self.fields.append(Field(self.win, white, x, y,
self.grid_side))
        y -= self.grid_side

        # row
        for i in range(6):
            if i == 1:
                self.fields.append(Field(self.win, green, x, y,
self.grid_side))
            else:
                self.fields.append(Field(self.win, white, x, y,
self.grid_side))
            x += self.grid_side

        # column
        for i in range(6):
            y -= self.grid_side
            self.fields.append(Field(self.win, white, x, y,
self.grid_side))

        x += self.grid_side
        self.fields.append(Field(self.win, white, x, y, self.grid_side))
        x += self.grid_side

        # column
        for i in range(6):
            if i == 1:
                self.fields.append(Field(self.win, blue, x, y,
self.grid_side))
            else:
                self.fields.append(Field(self.win, white, x, y,
self.grid_side))
            y += self.grid_side

        # row
        for i in range(6):
            x += self.grid_side
            self.fields.append(Field(self.win, white, x, y,
self.grid_side))

        y += self.grid_side
        self.fields.append(Field(self.win, white, x, y, self.grid_side))
        y += self.grid_side

        # row
        for i in range(6):
```

```
            if i == 1:
                self.fields.append(Field(self.win, yellow, x, y,
self.grid_side))
            else:
                self.fields.append(Field(self.win, white, x, y,
self.grid_side))
            x -= self.grid_side

        # column
        for i in range(6):
            y += self.grid_side
            self.fields.append(Field(self.win, white, x, y,
self.grid_side))

        x -= self.grid_side
        self.fields.append(Field(self.win, white, x, y, self.grid_side))
        x -= self.grid_side

    def update(self, pawns):
        for field in self.fields:
            field.reset_pawns()

        for pawn in pawns:
            if pawn.position != -1:
                self.fields[pawn.position].pawns[pawn.color].append(pawn)

    def find_conflicts(self, chosen):
        # chosen is the moved pawn, consistent with chinczyk and agent
        potential_casualties = []
        conflict = False
        field = self.fields[chosen.position]
        defence = 0
        attack = 1
        if not field.star or chosen.finishing == 0:
            for pawns in field.pawns:
                for pawn in field.pawns[pawns]:
                    if pawn.team == chosen.team:
                        attack += 1
                    else:
                        defence += 1
                        potential_casualties.append(pawn)

            if defence <= attack and defence != 0:
                conflict = True
                for pawn in potential_casualties:
                    pawn.reset()

        return conflict

    def draw(self):
        for field in self.fields:
            field.draw()


if __name__ == "__main__":
    WIN_SIDE = 610
    win = pygame.display.set_mode((WIN_SIDE, WIN_SIDE))
    MARGIN = 10
    color_numbers = {
        "r": 0,
        "g": 1,
```

```
        "b": 2,
        "y": 3
    }

    board = Path(win, WIN_SIDE, MARGIN)
    board.draw()
    time.sleep(10)
```

Pawn.py

```
color_numbers = {
    (255, 0, 0): 0,
    (0, 255, 0): 1,
    (0, 0, 255): 2,
    (255, 255, 0): 3,
}


class Pawn:

    def __init__(self, color, team, index):
        self.position = -1
        self.color = color
        self.team = team
        self.possible = False
        self.finishing = 0
        self.finished = False
        self.index = index

    def movable(self, moves):
        self.possible = False
        if self.position != -1 and not self.finished:
            if not self.finishing or (self.finishing and
(self.position+moves) <= 5):
                self.possible = True

    def move(self, moves):
        if self.possible:
            for _ in range(moves):
                if self.position == (13 * ((color_numbers[self.color]+3) %
4 + 1))-1 and not self.finishing:
                    self.finishing = 1
                    self.position = 0
                else:
                    self.position = (self.position + 1) % 52

        elif self.position == -1:
            self.position = 1 + color_numbers[self.color] * 13

        if self.finishing and self.position == 5:
            self.finished = True

    def reset(self):
        self.position = -1
        self.possible = False
```

```
if __name__ == "__main__":
    pass
```

Playe.py

```
import pygame.display

import ludo.dice as dice
from ludo.pawn import Pawn


class Player:

    def __init__(self, color, team, starting, final):
        self.starting = starting
        self.pawns = [Pawn(color, team, i) for i in range(4)]
        self.color = color
        self.finished = False
        self.final = final
        self.team = team
        self.place_pawns()

    def place_pawns(self):
        for x, pawn in enumerate(self.pawns):
            self.starting.fields[x].pawns[self.color].append(pawn)

    def move(self, strikes):
        moves = dice.throw()
        # moves = int(input("how far do you wanna move: "))
        print(f"dice: {moves}")
        if moves == 6:
            strikes += 1
        again = False
        candidates = []
        chosen = False

        if strikes != 3:
            for pawn in self.pawns:
                pawn.movable(moves)
                if pawn.possible or (pawn.position == -1 and moves == 6):
                    candidates.append(pawn)

            if len(candidates) == 1:
                chosen = candidates[0]
                chosen.move(moves)

            elif len(candidates) != 0:
                print([candidate.index for candidate in candidates],
self.color)
                move = True
                while move:
                    chosen = int(input("give the number of your pawn of
choice: "))
                    try:
                        for candidate in candidates:
                            if candidate.index == chosen:
                                chosen = candidate
                        chosen.move(moves)
```

```
                        move = False

                    except:
                        print("wrong input")

            if chosen and (moves == 6 or chosen.finished):
                again = True

            if chosen and chosen.finished:
                strikes = 0

        return strikes, again, chosen

    def update(self):
        self.starting.reset()
        self.final.reset()
        for pawn in self.pawns:
            if pawn.position == -1:

self.starting.fields[pawn.index].pawns[pawn.color].append(pawn)
            if pawn.finishing:

self.final.fields[pawn.position].pawns[pawn.color].append(pawn)


if __name__ == "__main__":
    pass
```

Starting.py

```
import time
from ludo.field import Field
import pygame
from colorama import Fore, init
init(autoreset=True)
pygame.init()


# it is the 4 starting places
class Starting:

    def __init__(self, win, color, x, y, side):
        self.fields = []
        # coordinates of upper left square
        self.win = win
        self.x = x
        self.y = y
        # side of squares:
        self.side = side
        self.color = color
        self.create_fields()

    def reset(self):
        for field in self.fields:
            field.reset_pawns()

    def create_fields(self):
        # here probably could be a smart for but 4 squares is not that much
```

```
work
        self.fields.append(Field(self.win, self.color, self.x, self.y,
self.side))
        self.fields.append(Field(self.win, self.color, self.x + self.side,
self.y, self.side))
        self.fields.append(Field(self.win, self.color, self.x, self.y +
self.side, self.side))
        self.fields.append(Field(self.win, self.color, self.x + self.side,
self.y + self.side, self.side))

    def draw(self):
        for field in self.fields:
            field.draw()
```

botV0 directory

agentV0.py

```
import pygame.display

import ludo.dice as dice
from ludo.pawn import Pawn


class Player:

    def __init__(self, color, team, starting, final):
        self.starting = starting
        self.pawns = [Pawn(color, team, i) for i in range(4)]
        self.color = color
        self.finished = False
        self.final = final
        self.team = team
        self.place_pawns()

    def place_pawns(self):
        for x, pawn in enumerate(self.pawns):
            self.starting.fields[x].pawns[self.color].append(pawn)

    def move(self, strikes, moves, chosen, candidates):
        chosen = chosen
        reward = 0
        if moves == 6:
            strikes += 1
        again = False

        if strikes != 3:
            if len(candidates) != 0:
                if not (chosen in candidates):
                    reward -= 2
                    chosen = candidates[0]
                chosen.move(moves)

            if chosen and (moves == 6 or chosen.finished):
                again = True

            if chosen and chosen.finished:
```

```
                strikes = 0

        return strikes, again, chosen, reward

    def update(self):
        self.starting.reset()
        self.final.reset()
        for pawn in self.pawns:
            if pawn.position == -1:

self.starting.fields[pawn.index].pawns[pawn.color].append(pawn)
            if pawn.finishing:

self.final.fields[pawn.position].pawns[pawn.color].append(pawn)


if __name__ == "__main__":
    pass
```

configV0.txt

```
[NEAT]
fitness_criterion     = max
fitness_threshold = 1000
no_fitness_termination = False
pop_size              = 64
reset_on_extinction   = False

[DefaultGenome]
# node activation options
activation_default      = tanh
activation_mutate_rate  = 0.2
activation_options      = tanh

# node aggregation options
aggregation_default     = sum
aggregation_mutate_rate = 0.0
aggregation_options     = sum

# node bias options
bias_init_mean          = 0.0
bias_init_stdev         = 1.0
bias_max_value          = 100.0
bias_min_value          = -100.0
bias_mutate_power       = 0.5
bias_mutate_rate        = 0.7
bias_replace_rate       = 0.1

# genome compatibility options
compatibility_disjoint_coefficient = 1.0
compatibility_weight_coefficient   = 0.5

# connection add/remove rates
conn_add_prob           = 0.5
conn_delete_prob        = 0.5

# connection enable options
```

```
enabled_default          = True
enabled_mutate_rate      = 0.01

feed_forward             = True
initial_connection       = full_direct

# node add/remove rates
node_add_prob            = 0.4
node_delete_prob         = 0.4

# network parameters
num_inputs               = 17
num_hidden               = 0
num_outputs              = 4

# node response options
response_init_mean       = 1.0
response_init_stdev      = 0.0
response_max_value       = 100.0
response_min_value       = -100.0
response_mutate_power    = 0.0
response_mutate_rate     = 0.0
response_replace_rate    = 0.0

# connection weight options
weight_init_mean         = 0.0
weight_init_stdev        = 1.0
weight_max_value         = 200
weight_min_value         = -200
weight_mutate_power      = 0.5
weight_mutate_rate       = 0.8
weight_replace_rate      = 0.1

[DefaultSpeciesSet]
compatibility_threshold = 3.0

[DefaultStagnation]
species_fitness_func = max
max_stagnation = 200
species_elitism = 0

[DefaultReproduction]
survival_threshold = 0.2
elitism = 0
```

Main.py (from the botV0 directory)

```
import time
from ludo.indicator import Indicator
import ludo.dice as dice
from ludo.board import Board
from agentV0 import Player
import numpy as np
import neat
import os


def main(genomes, config):
```

```
    colors = [(255, 0, 0), (0, 255, 0), (0, 0, 255), (255, 255, 0)]
    teams = [0, 1, 2, 3]
    win_side = 610
    win = 0
    margin = 5

    nets = []
    ge = []
    for id, g in genomes:
        # setting up genomes, connecting them and appending to the genome
list named ge

        net = neat.nn.FeedForwardNetwork.create(g, config)
        nets.append(net)
        g.fitness = 0
        ge.append(g)

    while not len(ge) < 4:
        advancing_ge = []
        advancing_nets = []

        for j in range(int(len(ge)/4)):
            x = 4 * j
            # one round of a tournament,
            # not checking if ge devisible by 4 so population should be a
power of 4
            # points for following who wins and goes on
            points = [0, 0, 0, 0]
            for game in range(4):
                # making one set of genomes play eachother multiple times
so as to decrease luck factor
                # creating board
                board = Board(win, win_side, margin)
                # creating players
                players = [Player(colors[i], teams[i], board.starts[i],
board.finish_lines[i]) for i in range(4)]
                i = 0
                end = False
                while not end:

                    playing = players[i]
                    strikes = 0
                    again = True
                    # this is used to make multiple moves of one player
possible
                    while again:
                        again = False
                        moves = dice.throw()
                        # playing player moving
                        # activating net of x + indx of player from ge
                        # activating by position of every pawn
                        candidates = []
                        chosen = False
                        for pawn in playing.pawns:
                            pawn.movable(moves)
                            if pawn.possible or (pawn.position == -1 and
moves == 6):
                                candidates.append(pawn)
                        if len(candidates) > 1:
                            state = []
```

```
                            for k in range(16):
                                pawn = players[(i + k//4) % 4].pawns[k % 4]
                                if pawn.finishing != 0:
                                    state.append(pawn.position + 52)
                                else:
                                    state.append(pawn.position)
                            state = tuple([moves] + state)
                            output = nets[x + i].activate(state)
                            chosen = playing.pawns[np.argmax(output)]
                            strikes, again, chosen, reward =
playing.move(strikes, moves, chosen, candidates)

                            # adding rewards for quality of moves, ex.
where they legal
                            ge[x+i].fitness += reward
                        elif len(candidates) == 1:
                            chosen = candidates[0]
                            chosen.move(moves)

                        # conflicts
                        if chosen and (chosen.finished or
board.path.find_conflicts(chosen)):
                            again = True
                            strikes = 0

                        # moves pawns back into starting positions if they
were taken out and updates finish lines
                        for player in players:
                            player.update()

                        # after moving and conflicts updating path since
final and starting where updated before

                        on_board = []
                        for player in players:
                            for pawn in player.pawns:
                                if pawn.position != -1 and not
pawn.finishing and not pawn.finished:
                                    on_board.append(pawn)

                        board.path.update(on_board)
                        # checking if someone won and ending the game
                        status = [pawn.finished for pawn in playing.pawns]
                        if all(status):
                            end = True
                            again = False
                            points[i] += 1

                    i += 1
                    i = i % 4

            winner = x + np.argmax(points)
            advancing_ge.append(ge[winner])
            advancing_nets.append(nets[winner])

        for g in advancing_ge:
            g.fitness += 20
        ge = advancing_ge
        nets = advancing_nets
```

```
def run(config_path):
    # those match the topic in configuration file, those names down there,
    config = neat.Config(neat.DefaultGenome, neat.DefaultReproduction,
                         neat.DefaultSpeciesSet, neat.DefaultStagnation,
config_path)

    p = neat.Population(config)

    # showing stats instead of black running screan

    p.add_reporter(neat.StdOutReporter(True))
    stats = neat.StatisticsReporter()
    p.add_reporter(stats)
    p.add_reporter(neat.Checkpointer(500))

    winner = p.run(main)
    print('\nBest genome:\n{!s}'.format(winner))


if __name__ == "__main__":
    local_dir = os.path.dirname(__file__)
    config_path = os.path.join(local_dir, 'configV0.txt')
    run(config_path)
```

main_visual.py

```
import time
from ludo.indicator import Indicator
import pygame
import ludo.dice as dice
from ludo.board import Board
from agentV0 import Player
import numpy as np
import neat
import os
clock = pygame.time.Clock()
pygame.init()
pygame.font.init()
frame_rate = 30


def main(genomes, config):
    # setting up pygame
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
    colors = [(255, 0, 0), (0, 255, 0), (0, 0, 255), (255, 255, 0)]
    teams = [0, 1, 2, 3]
    win_side = 610
    win = pygame.display.set_mode((win_side, win_side))
    margin = 5

    nets = []
    ge = []
    for id, g in genomes:
        # setting up genomes, connecting them and appending to the genome
list named ge
```

```
        net = neat.nn.FeedForwardNetwork.create(g, config)
        nets.append(net)
        g.fitness = 0
        ge.append(g)

    while not len(ge) < 4:
        advancing_ge = []
        advancing_nets = []
        for j in range(int(len(ge)/4)):
            x = 0 + 4 * j
            # one round of a tournament,
            # not checking if ge devisible by 4 so population should be a
power of 4
            # points for following who wins and goes on
            points = [0, 0, 0, 0]
            for game in range(4):
                # making one set of genomes play eachother multiple times
so as to decrease luck factor
                # creating board
                board = Board(win, win_side, margin)
                # creating players
                players = [Player(colors[i], teams[i], board.starts[i],
board.finish_lines[i]) for i in range(4)]
                i = 0
                end = False
                while not end:

                    for event in pygame.event.get():
                        if event.type == pygame.QUIT:
                            pygame.quit()

                    playing = players[i]
                    strikes = 0
                    again = True
                    # this is used to make multiple moves of one player
possible
                    while again:
                        again = False
                        for event in pygame.event.get():
                            if event.type == pygame.QUIT:
                                pygame.quit()
                        board.draw()
                        pygame.display.update()

                        clock.tick(frame_rate)
                        moves = dice.throw()
                        # playing player moving
                        # activating net of x + indx of player from ge
                        # activating by position of every pawn
                        state = []
                        candidates = []
                        chosen = False
                        for pawn in playing.pawns:
                            pawn.movable(moves)
                            if pawn.possible or (pawn.position == -1 and
moves == 6):
                                candidates.append(pawn)
                        if len(candidates) > 1:
                            state = []
                            for k in range(16):
                                pawn = players[(i + k // 4) % 4].pawns[k %
```

```
4]
                                if pawn.finishing != 0:
                                    state.append(pawn.position + 52)
                                else:
                                    state.append(pawn.position)
                            state = tuple([moves] + state)
                            output = nets[x + i].activate(state)
                            chosen = playing.pawns[np.argmax(output)]
                            strikes, again, chosen, reward =
playing.move(strikes, moves, chosen, candidates)

                            # adding rewards for quality of moves, ex.
where they legal
                            ge[x+i].fitness += reward
                        elif len(candidates) == 1:
                            chosen = candidates[0]
                            chosen.move(moves)

                        # conflicts
                        if chosen and (chosen.finished or
board.path.find_conflicts(chosen)):
                            again = True
                            strikes = 0

                        # moves pawns back into starting positions if they
were taken out and updates finish lines
                        for player in players:
                            player.update()

                        # after moving and conflicts updating path since
final and starting where updated before

                        on_board = []
                        for player in players:
                            for pawn in player.pawns:
                                if pawn.position != -1 and not
pawn.finishing and not pawn.finished:
                                    on_board.append(pawn)

                        board.path.update(on_board)
                        # checking if someone won and ending the game
                        status = [pawn.finished for pawn in playing.pawns]
                        if all(status):
                            end = True
                            again = False
                            points[i] += 1

                        # printing the board
                        board.draw()
                        pygame.display.update()
                    i += 1
                    i = i % 4

            winner = np.argmax(points)
            advancing_ge.append(ge[winner])
            advancing_nets.append(nets[winner])

        for g in advancing_ge:
            g.fitness += 20
        ge = advancing_ge
        nets = advancing_nets
```

```
def run(config_path):
    # those match the topic in configuration file, those names down there,
    config = neat.Config(neat.DefaultGenome, neat.DefaultReproduction,
                         neat.DefaultSpeciesSet, neat.DefaultStagnation,
config_path)

    p = neat.Population(config)

    # showing stats instead of black running screan

    p.add_reporter(neat.StdOutReporter(True))
    stats = neat.StatisticsReporter()
    p.add_reporter(stats)

    winner = p.run(main, 500)
    print('\nBest genome:\n{!s}'.format(winner))


if __name__ == "__main__":
    local_dir = os.path.dirname(__file__)
    config_path = os.path.join(local_dir, 'configV0.txt')
    run(config_path)
```

botV1 directory

agentV1.py

```
import pygame.display
import ludo.dice as dice
from ludo.pawn import Pawn


class Player:

    def __init__(self, color, team, starting, final):
        self.starting = starting
        self.pawns = [Pawn(color, team, i) for i in range(4)]
        self.color = color
        self.finished = False
        self.final = final
        self.team = team
        self.place_pawns()

    def place_pawns(self):
        for x, pawn in enumerate(self.pawns):
            self.starting.fields[x].pawns[self.color].append(pawn)

    def move(self, strikes, moves, chosen, candidates):
        chosen = chosen
        reward = 0
        if moves == 6:
            strikes += 1
        again = False

        if strikes != 3:
```

```
            if len(candidates) != 0:
                if not (chosen in candidates):
                    reward -= 2
                    chosen = candidates[0]
                chosen.move(moves)

            if chosen and (moves == 6 or chosen.finished):
                again = True

            if chosen and chosen.finished:
                strikes = 0

        return strikes, again, chosen, reward

    def update(self):
        self.starting.reset()
        self.final.reset()
        for pawn in self.pawns:
            if pawn.position == -1:

self.starting.fields[pawn.index].pawns[pawn.color].append(pawn)
            if pawn.finishing:

self.final.fields[pawn.position].pawns[pawn.color].append(pawn)


if __name__ == "__main__":
    pass
```

boardV1.py

```
import time
from ludo.field import Field
from botV1.pathV1 import Path
from ludo.final import Final
from ludo.starting import Starting
import pygame
from colorama import Fore, init

init(autoreset=True)
pygame.init()


# combining starting, final fields and path into one board
class Board:

    def __init__(self, win, side, margin, ):
        # for drawing
        self.side = side
        self.colors = [(255, 0, 0), (0, 255, 0), (0, 0, 255), (255, 255,
0)]
        self.margin = margin
        self.grid_side = (self.side - self.margin * 2) / 15
        self.win = win

        # logic
        self.path = Path(self.win, self.side, self.margin)
        self.finish_lines = []
```

```
        self.create_final()
        self.starts = []
        self.create_starting()

    def create_starting(self):
        # only 4
        self.starts.append(Starting(self.win, self.colors[0], self.margin +
self.grid_side * 2,
                                    self.margin + self.grid_side * 11,
self.grid_side))

        self.starts.append(Starting(self.win, self.colors[1], self.margin +
self.grid_side * 2,
                                    self.margin + self.grid_side * 2,
self.grid_side))

        self.starts.append(Starting(self.win, self.colors[2], self.margin +
self.grid_side * 11,
                                    self.margin + self.grid_side * 2,
self.grid_side))

        self.starts.append(Starting(self.win, self.colors[3], self.margin +
self.grid_side * 11,
                                    self.margin + self.grid_side * 11,
self.grid_side))

    def create_final(self):
        # only 4 no need for any smart for
        # added in a r, g, b, y order if that will prove problematic could
use dictionary
        self.finish_lines.append(Final(self.win, self.colors[0],
self.margin + self.grid_side * 7,
                                       self.side - self.margin -
self.grid_side * 2, [0, -self.grid_side],
                                       self.grid_side))

        self.finish_lines.append(Final(self.win, self.colors[1],
self.margin + self.grid_side,
                                       self.margin + self.grid_side * 7,
[self.grid_side, 0], self.grid_side))

        self.finish_lines.append(Final(self.win, self.colors[2],
self.margin + self.grid_side * 7,
                                       self.margin + self.grid_side, [0,
self.grid_side], self.grid_side))

        self.finish_lines.append(Final(self.win, self.colors[3], self.side
- self.margin - self.grid_side * 2,
                                       self.margin + self.grid_side * 7, [-
self.grid_side, 0], self.grid_side))

    def draw(self):
        self.path.draw()
        for final in self.finish_lines:
            final.draw()
        for start in self.starts:
            start.draw()


if __name__ == "__main__":
    WIN_SIDE = 610
```

```
    win = pygame.display.set_mode((WIN_SIDE, WIN_SIDE))
    MARGIN = 5
    color_numbers = {
        "r": 0,
        "g": 1,
        "b": 2,
        "y": 3
    }

    board = Board(win, WIN_SIDE, MARGIN)
    board.draw()
    time.sleep(10)
```

configV1.txt

```
[NEAT]
fitness_criterion     = max
fitness_threshold = 100
no_fitness_termination = True
pop_size              = 64
reset_on_extinction   = False

[DefaultGenome]
# node activation options
activation_default      = identity
activation_mutate_rate  = 0.1
activation_options      = clamped cube exp identity log relu sigmoid
softplus tanh

# node aggregation options
aggregation_default     = sum
aggregation_mutate_rate = 0.0
aggregation_options     = sum

# node bias options
bias_init_mean          = 0.0
bias_init_stdev         = 1.0
bias_max_value          = 100.0
bias_min_value          = -100.0
bias_mutate_power       = 0.5
bias_mutate_rate        = 0.7
bias_replace_rate       = 0.1

# genome compatibility options
compatibility_disjoint_coefficient = 1.0
compatibility_weight_coefficient   = 0.5

# connection add/remove rates
conn_add_prob           = 0.5
conn_delete_prob        = 0.5

# connection enable options
enabled_default         = True
enabled_mutate_rate     = 0.01

feed_forward            = True
initial_connection      = full_direct
```

```
# node add/remove rates
node_add_prob           = 0.2
node_delete_prob        = 0.2

# network parameters
num_inputs              = 17
num_hidden              = 0
num_outputs             = 1

# node response options
response_init_mean      = 1.0
response_init_stdev     = 0.0
response_max_value      = 100.0
response_min_value      = -100.0
response_mutate_power   = 0.0
response_mutate_rate    = 0.0
response_replace_rate   = 0.0

# connection weight options
weight_init_mean        = 0.0
weight_init_stdev       = 1.0
weight_max_value        = 200
weight_min_value        = -200
weight_mutate_power     = 0.5
weight_mutate_rate      = 0.8
weight_replace_rate     = 0.1

[DefaultSpeciesSet]
compatibility_threshold = 3.0

[DefaultStagnation]
species_fitness_func = max
max_stagnation = 500
species_elitism = 2

[DefaultReproduction]
survival_threshold = 0.2
elitism = 0
```

main.py

```python
import time
from ludo.indicator import Indicator
import ludo.dice as dice
from botV1.boardV1 import Board
from botV1.agentV1 import Player
from ludo.pawn import Pawn
import numpy as np
import neat
import os
import random


def main(genomes, config):

    colors = [(255, 0, 0), (0, 255, 0), (0, 0, 255), (255, 255, 0)]
    teams = [0, 1, 2, 3]
```

```
    win_side = 610
    win = 0
    margin = 5
    random.seed()
    nets = []
    ge = []
    for id, g in genomes:
        # setting up genomes, connecting them and appending to the genome
list named ge

        net = neat.nn.FeedForwardNetwork.create(g, config)
        nets.append(net)
        g.fitness = 0
        ge.append(g)

    while not len(ge) < 4:
        advancing_ge = []
        advancing_nets = []

        for j in range(int(len(ge)/4)):
            x = 4 * j
            # one round of a tournament,
            # not checking if ge devisible by 4 so population should be a
power of 4
            # points for following who wins and goes on
            points = [0, 0, 0, 0]
            for game in range(4):
                # making one set of genomes play eachother multiple times
so as to decrease luck factor
                # creating board
                board = Board(win, win_side, margin)
                # creating players
                players = [Player(colors[i], teams[i], board.starts[i],
board.finish_lines[i]) for i in range(4)]
                i = 0
                end = False
                while not end:

                    playing = players[i]
                    strikes = 0
                    again = True
                    # this is used to make multiple moves of one player
possible
                    while again:

                        again = False
                        moves = dice.throw()
                        # playing player moving
                        # activating net of x + indx of player from ge
                        # activating by position of every pawn
                        candidates = []
                        chosen = False
                        for pawn in playing.pawns:
                            pawn.movable(moves)
                            if pawn.possible or (pawn.position == -1 and
moves == 6):
                                candidates.append(pawn)

                        if len(candidates) > 0:
                            if len(candidates) == 1:
                                chosen = candidates[0]
```

```
                                chosen.move(moves)

                            elif len(candidates) > 1:
                                states = []
                                # this is a loop which creates a set of
dummy states
                                for candidate in candidates:
                                    # cooping strikes
                                    c_strikes = strikes
                                    # creating and moving a copy of a pawn
not to disrupt the original game
                                    chosen = Pawn(candidate.color,
candidate.team, candidate.index)
                                    chosen.position =
int(candidate.position)
                                    chosen.possible = True
                                    chosen.move(moves)
                                    # checking for conflicts
                                    conflict, potential_casualties =
board.path.find_conflictsV1(chosen)
                                    # creating state
                                    state = []
                                    for k in range(16):
                                        pawn = players[(i + k // 4) %
4].pawns[k % 4]
                                        # if conflict and in potential
casualties pawn should be teleported
                                        # to the beggining
                                        # but since we do not actually move
pawns just inserting -1 into state
                                        if conflict and pawn in
potential_casualties:
                                            state.append(-1)
                                        elif pawn.finishing == 1:
                                            state.append(pawn.position +
52)
                                        # this pawn is the one moved in
this scenario therefore value of the copy is
                                        # different from this of the actual
pawn
                                        elif pawn in playing.pawns and
pawn.index == chosen.index:
                                            state.append(chosen.position)
                                        else:
                                            state.append(pawn.position)
                                    # changing strikes if need be, but only
a copy
                                    if chosen and (chosen.finished or
conflict):
                                        c_strikes = 0

                                    state = tuple(state + [c_strikes])
                                    states.append(state)

                                # getting output from net for every state
                                outputs = [nets[x + i].activate(state) for
state in states]
                                index = np.argmax(outputs)

                                # choosing and moving the choice, this
already has impact on the game
```

```
                                chosen = candidates[index]
                                chosen.move(moves)

                            # actual conflict resolution
                            if chosen.finished or
board.path.find_conflicts(chosen):
                                again = True
                                strikes = 0

                            # moves pawns back into starting positions if
they were taken out and updates finish lines
                            for player in players:
                                player.update()

                            # after moving and conflicts updating path
since final and starting where updated before
                            on_board = []
                            for player in players:
                                for pawn in player.pawns:
                                    if pawn.position != -1 and not
pawn.finishing and not pawn.finished:
                                        on_board.append(pawn)

                            board.path.update(on_board)

                            # checking if again previously handled but
player.move but in this version
                            # it just does not make sense
                            if moves == 6:
                                strikes += 1
                                again = True

                        status = [pawn.finished for pawn in playing.pawns]
                        if all(status):
                            end = True
                            again = False
                            points[i] += 1

                    i += 1
                    i = i % 4

            winner = x + np.argmax(points)
            advancing_ge.append(ge[winner])
            advancing_nets.append(nets[winner])

        for g in advancing_ge:
            g.fitness += 20
        ge = advancing_ge
        nets = advancing_nets


def run(config_path):
    # those match the topic in configuration file, those names down there,
    config = neat.Config(neat.DefaultGenome, neat.DefaultReproduction,
                         neat.DefaultSpeciesSet, neat.DefaultStagnation,
config_path)

    p = neat.Population(config)

    # showing stats instead of black running screan
    p.add_reporter(neat.Checkpointer(30))
```

```
    p.add_reporter(neat.StdOutReporter(True))
    stats = neat.StatisticsReporter()
    p.add_reporter(stats)

    winner = p.run(main, 1)
    print('\nBest genome:\n{!s}'.format(winner))
    return winner, p.config


if __name__ == "__main__":
    local_dir = os.path.dirname(__file__)
    config_path = os.path.join(local_dir, 'configV1.txt')
    run(config_path)
```

main_restore.py

```
import time
from ludo.indicator import Indicator
import ludo.dice as dice
from botV1.boardV1 import Board
from agentV1 import Player
from ludo.pawn import Pawn
import numpy as np
import neat
import os


def main(genomes, config):

    colors = [(255, 0, 0), (0, 255, 0), (0, 0, 255), (255, 255, 0)]
    teams = [0, 1, 2, 3]
    win_side = 610
    win = 0
    margin = 5

    nets = []
    ge = []
    for id, g in genomes:
        # setting up genomes, connecting them and appending to the genome
list named ge

        net = neat.nn.FeedForwardNetwork.create(g, config)
        nets.append(net)
        g.fitness = 0
        ge.append(g)

    while not len(ge) < 4:
        advancing_ge = []
        advancing_nets = []

        for j in range(int(len(ge)/4)):
            x = 4 * j
            # one round of a tournament,
            # not checking if ge devisible by 4 so population should be a
power of 4
            # points for following who wins and goes on
            points = [0, 0, 0, 0]
            for game in range(4):
```

```python
                # making one set of genomes play eachother multiple times
so as to decrease luck factor
                # creating board
                board = Board(win, win_side, margin)
                # creating players
                players = [Player(colors[i], teams[i], board.starts[i],
board.finish_lines[i]) for i in range(4)]
                i = 0
                end = False
                while not end:

                    playing = players[i]
                    strikes = 0
                    again = True
                    # this is used to make multiple moves of one player
possible
                    while again:

                        again = False
                        moves = dice.throw()
                        # playing player moving
                        # activating net of x + indx of player from ge
                        # activating by position of every pawn
                        candidates = []
                        chosen = False
                        for pawn in playing.pawns:
                            pawn.movable(moves)
                            if pawn.possible or (pawn.position == -1 and
moves == 6):
                                candidates.append(pawn)

                        if len(candidates) > 0:
                            if len(candidates) == 1:
                                chosen = candidates[0]
                                chosen.move(moves)

                            elif len(candidates) > 1:
                                states = []
                                # this is a loop which creates a set of
dummy states
                                for candidate in candidates:
                                    # cooping strikes
                                    c_strikes = strikes
                                    # creating and moving a copy of a pawn
not to disrupt the original game
                                    chosen = Pawn(candidate.color,
candidate.team, candidate.index)
                                    chosen.position =
int(candidate.position)
                                    chosen.possible = True
                                    chosen.move(moves)
                                    # checking for conflicts
                                    conflict, potential_casualties =
board.path.find_conflictsV1(chosen)
                                    # creating state
                                    state = []
                                    for k in range(16):
                                        pawn = players[(i + k // 4) %
4].pawns[k % 4]
                                        # if conflict and in potential
casualties pawn should be teleported
```

```
                                        # to the beggining
                                        # but since we do not actually move
pawns just inserting -1 into state
                                        if conflict and pawn in
potential_casualties:
                                            state.append(-1)
                                        elif pawn.finishing == 1:
                                            state.append(pawn.position +
52)
                                        # this pawn is the one moved in
this scenario therefore value of the copy is
                                        # different from this of the actual
pawn
                                        elif pawn in playing.pawns and
pawn.index == chosen.index:
                                            state.append(chosen.position)
                                        else:
                                            state.append(pawn.position)
                                    # changing strikes if need be, but only
a copy
                                    if chosen and (chosen.finished or
conflict):
                                        c_strikes = 0

                                    state = tuple(state + [c_strikes])
                                    states.append(state)

                                # getting output from net for every state
                                outputs = [nets[x + i].activate(state) for
state in states]
                                index = np.argmax(outputs)

                                # choosing and moving the choice, this
already has impact on the game
                                chosen = candidates[index]
                                chosen.move(moves)

                            # actual conflict resolution
                            if chosen.finished or
board.path.find_conflicts(chosen):
                                again = True
                                strikes = 0

                            # moves pawns back into starting positions if
they were taken out and updates finish lines
                            for player in players:
                                player.update()

                            # after moving and conflicts updating path
since final and starting where updated before
                            on_board = []
                            for player in players:
                                for pawn in player.pawns:
                                    if pawn.position != -1 and not
pawn.finishing and not pawn.finished:
                                        on_board.append(pawn)

                            board.path.update(on_board)

                            # checking if again previously handled but
player.move but in this version
```

```
                            # it just does not make sense
                            if moves == 6:
                                strikes += 1
                                again = True

                            # checking if someone won and ending the game
                            status = [pawn.finished for pawn in
playing.pawns]
                            if all(status):
                                end = True
                                again = False
                                points[i] += 1

                        status = [pawn.finished for pawn in playing.pawns]
                        if all(status):
                            end = True
                            again = False
                            points[i] += 1

                    i += 1
                    i = i % 4

            winner = x + np.argmax(points)
            advancing_ge.append(ge[winner])
            advancing_nets.append(nets[winner])

        for g in advancing_ge:
            g.fitness += 20
        ge = advancing_ge
        nets = advancing_nets


def run(config_path):
    # those match the topic in configuration file, those names down there,
    config = neat.Config(neat.DefaultGenome, neat.DefaultReproduction,
                         neat.DefaultSpeciesSet, neat.DefaultStagnation,
config_path)

    p = neat.Checkpointer.restore_checkpoint('neat-checkpoint-9981')

    # showing stats instead of black running screan
    p.add_reporter(neat.Checkpointer(100, 3600))
    p.add_reporter(neat.StdOutReporter(True))
    stats = neat.StatisticsReporter()
    p.add_reporter(stats)

    winner = p.run(main, 100000)
    print('\nBest genome:\n{!s}'.format(winner))


if __name__ == "__main__":
    local_dir = os.path.dirname(__file__)
    config_path = os.path.join(local_dir, 'configV1.txt')
    run(config_path)
```

main_visual.py

```python
import time
from ludo.indicator import Indicator
import ludo.dice as dice
from botV1.boardV1 import Board
from agentV1 import Player
from ludo.pawn import Pawn
import numpy as np
import neat
import os
import pygame

clock = pygame.time.Clock()
pygame.init()
pygame.font.init()
frame_rate = 30


def main(genomes, config):

    colors = [(255, 0, 0), (0, 255, 0), (0, 0, 255), (255, 255, 0)]
    teams = [0, 1, 2, 3]
    win_side = 610
    win = pygame.display.set_mode((win_side, win_side))
    margin = 5

    nets = []
    ge = []
    for id, g in genomes:
        # setting up genomes, connecting them and appending to the genome
list named ge

        net = neat.nn.FeedForwardNetwork.create(g, config)
        nets.append(net)
        g.fitness = 0
        ge.append(g)

    while not len(ge) < 4:
        advancing_ge = []
        advancing_nets = []

        for j in range(int(len(ge)/4)):
            x = 0 + 4 * j
            # one round of a tournament,
            # not checking if ge devisible by 4 so population should be a
power of 4
            # points for following who wins and goes on
            points = [0, 0, 0, 0]
            for game in range(4):
                # making one set of genomes play eachother multiple times
so as to decrease luck factor
                # creating board
                board = Board(win, win_side, margin)
                # creating players
                players = [Player(colors[i], teams[i], board.starts[i],
board.finish_lines[i]) for i in range(4)]
                i = 0
                end = False
                while not end:

                    playing = players[i]
                    strikes = 0
```

```
                    again = True
                    # this is used to make multiple moves of one player
possible
                    while again:
                        # pygame stuff
                        for event in pygame.event.get():
                            if event.type == pygame.QUIT:
                                pygame.quit()
                        board.draw()
                        pygame.display.update()
                        clock.tick(frame_rate)

                        # adding ability to pouse the game under p
                        pressed = pygame.key.get_pressed()
                        if pressed[pygame.K_p]:
                            time.sleep(30)

                        again = False
                        moves = dice.throw()
                        print(i, strikes, moves)
                        # playing player moving
                        # activating net of x + indx of player from ge
                        # activating by position of every pawn
                        candidates = []
                        chosen = False
                        for pawn in playing.pawns:
                            pawn.movable(moves)
                            if pawn.possible or (pawn.position == -1 and
moves == 6):
                                candidates.append(pawn)

                        if len(candidates) > 0:
                            if len(candidates) == 1:
                                chosen = candidates[0]
                                chosen.move(moves)

                            elif len(candidates) > 1:
                                states = []
                                # this is a loop which creates a set of
dummy states
                                for candidate in candidates:
                                    # cooping strikes
                                    c_strikes = strikes
                                    # creating and moving a copy of a pawn
not to disrupt the original game
                                    chosen = Pawn(candidate.color,
candidate.team, candidate.index)
                                    chosen.position =
int(candidate.position)
                                    chosen.possible = True
                                    chosen.move(moves)
                                    # checking for conflicts
                                    conflict, potential_casualties =
board.path.find_conflictsV1(chosen)
                                    # creating state
                                    state = []
                                    for k in range(16):
                                        pawn = players[(i + k // 4) %
4].pawns[k % 4]
                                        # if conflict and in potential
casualties pawn should be teleported
```

```
                                        # to the beggining
                                        # but since we do not actually move
pawns just inserting -1 into state
                                        if conflict and pawn in
potential_casualties:
                                            state.append(-1)
                                        elif pawn.finishing == 1:
                                            state.append(pawn.position +
52)
                                        # this pawn is the one moved in
this scenario therefore value of the copy is
                                        # different from this of the actual
pawn
                                        elif pawn in playing.pawns and
pawn.index == chosen.index:
                                            state.append(chosen.position)
                                        else:
                                            state.append(pawn.position)
                                    print(f"state {state}")
                                    # changing strikes if need be, but only
a copy
                                    if chosen and (chosen.finished or
conflict):
                                        c_strikes = 0

                                    state = tuple(state + [c_strikes])
                                    states.append(state)

                                # getting output from net for every state
                                outputs = [nets[x + i].activate(state) for
state in states]
                                index = np.argmax(outputs)
                                print(f"outputs:
{outputs}\nstates:{states}")

                                # choosing and moving the choice, this
already has impact on the game
                                chosen = candidates[index]
                                chosen.move(moves)

                            # actual conflict resolution
                            if chosen.finished or
board.path.find_conflicts(chosen):
                                again = True
                                strikes = 0

                            # moves pawns back into starting positions if
they were taken out and updates finish lines
                            for player in players:
                                player.update()

                            # after moving and conflicts updating path
since final and starting where updated before
                            on_board = []
                            for player in players:
                                for pawn in player.pawns:
                                    if pawn.position != -1 and not
pawn.finishing and not pawn.finished:
                                        on_board.append(pawn)

                            board.path.update(on_board)
```

```
                            # checking if again previously handled but
player.move but in this version
                            # it just does not make sense
                            if moves == 6:
                                strikes += 1
                                again = True

                            # checking if someone won and ending the game
                            status = [pawn.finished for pawn in
playing.pawns]
                            if all(status):
                                end = True
                                again = False
                                points[i] += 1

                        status = [pawn.finished for pawn in playing.pawns]
                        if all(status):
                            end = True
                            again = False
                            points[i] += 1

                        # printing the board
                        board.draw()
                        pygame.display.update()
                    i += 1
                    i = i % 4

            winner = x + np.argmax(points)
            advancing_ge.append(ge[winner])
            advancing_nets.append(nets[winner])

        for g in advancing_ge:
            g.fitness += 20
        ge = advancing_ge
        nets = advancing_nets


def run(config_path):
    # those match the topic in configuration file, those names down there,
    config = neat.Config(neat.DefaultGenome, neat.DefaultReproduction,
                         neat.DefaultSpeciesSet, neat.DefaultStagnation,
config_path)

    p = neat.Population(config)

    # showing stats instead of black running screan

    p.add_reporter(neat.StdOutReporter(True))
    stats = neat.StatisticsReporter()
    p.add_reporter(stats)

    winner = p.run(main)
    print('\nBest genome:\n{!s}'.format(winner))


if __name__ == "__main__":
    local_dir = os.path.dirname(__file__)
    config_path = os.path.join(local_dir, 'configV1.txt')
    run(config_path)
```

match directory

choosing_genomes.py

```
import time
from ludo.indicator import Indicator
import ludo.dice as dice
from botV1.boardV1 import Board
from botV1.agentV1 import Player
from ludo.pawn import Pawn
import numpy as np
import neat
import botV1.main as v1
import os


def main(genomes, config):

    colors = [(255, 0, 0), (0, 255, 0), (0, 0, 255), (255, 255, 0)]
    teams = [0, 1, 2, 3]
    win_side = 610
    win = 0
    margin = 5

    nets = []
    ge = []
    for id, g in genomes:
        # setting up genomes, connecting them and appending to the genome
list named ge

        net = neat.nn.FeedForwardNetwork.create(g, config)
        nets.append(net)
        g.fitness = 0
        ge.append(g)

    while not len(ge) < 4:
        advancing_ge = []
        advancing_nets = []

        for j in range(int(len(ge)/4)):
            x = 4 * j
            # one round of a tournament,
            # not checking if ge devisible by 4 so population should be a
power of 4
            # points for following who wins and goes on
            points = [0, 0, 0, 0]
            for game in range(4):
                # making one set of genomes play eachother multiple times
so as to decrease luck factor
                # creating board
                board = Board(win, win_side, margin)
                # creating players
                players = [Player(colors[i], teams[i], board.starts[i],
board.finish_lines[i]) for i in range(4)]
                i = 0
                end = False
                while not end:

                    playing = players[i]
```

```
                    strikes = 0
                    again = True
                    # this is used to make multiple moves of one player
possible
                    while again:

                        again = False
                        moves = dice.throw()
                        # playing player moving
                        # activating net of x + indx of player from ge
                        # activating by position of every pawn
                        candidates = []
                        chosen = False
                        for pawn in playing.pawns:
                            pawn.movable(moves)
                            if pawn.possible or (pawn.position == -1 and
moves == 6):
                                candidates.append(pawn)

                        if len(candidates) > 0:
                            if len(candidates) == 1:
                                chosen = candidates[0]
                                chosen.move(moves)

                            elif len(candidates) > 1:
                                states = []
                                # this is a loop which creates a set of
dummy states
                                for candidate in candidates:
                                    # cooping strikes
                                    c_strikes = strikes
                                    # creating and moving a copy of a pawn
not to disrupt the original game
                                    chosen = Pawn(candidate.color,
candidate.team, candidate.index)
                                    chosen.position =
int(candidate.position)
                                    chosen.possible = True
                                    chosen.move(moves)
                                    # checking for conflicts
                                    conflict, potential_casualties =
board.path.find_conflictsV1(chosen)
                                    # creating state
                                    state = []
                                    for k in range(16):
                                        pawn = players[(i + k // 4) %
4].pawns[k % 4]
                                        # if conflict and in potential
casualties pawn should be teleported
                                        # to the beggining
                                        # but since we do not actually move
pawns just inserting -1 into state
                                        if conflict and pawn in
potential_casualties:
                                            state.append(-1)
                                        elif pawn.finishing == 1:
                                            state.append(pawn.position +
52)
                                        # this pawn is the one moved in
this scenario therefore value of the copy is
                                        # different from this of the actual
```

```
pawn
                                        elif pawn in playing.pawns and
pawn.index == chosen.index:
                                            state.append(chosen.position)
                                        else:
                                            state.append(pawn.position)
                                    # changing strikes if need be, but only
a copy
                                    if chosen and (chosen.finished or
conflict):
                                        c_strikes = 0

                                    state = tuple(state + [c_strikes])
                                    states.append(state)

                                # getting output from net for every state
                                outputs = [nets[x + i].activate(state) for
state in states]
                                index = np.argmax(outputs)

                                # choosing and moving the choice, this
already has impact on the game
                                chosen = candidates[index]
                                chosen.move(moves)

                            # actual conflict resolution
                            if chosen.finished or
board.path.find_conflicts(chosen):
                                again = True
                                strikes = 0

                            # moves pawns back into starting positions if
they were taken out and updates finish lines
                            for player in players:
                                player.update()

                            # after moving and conflicts updating path
since final and starting where updated before
                            on_board = []
                            for player in players:
                                for pawn in player.pawns:
                                    if pawn.position != -1 and not
pawn.finishing and not pawn.finished:
                                        on_board.append(pawn)

                            board.path.update(on_board)

                            # checking if again previously handled but
player.move but in this version
                            # it just does not make sense
                            if moves == 6:
                                strikes += 1
                                again = True

                            # checking if someone won and ending the game
                            status = [pawn.finished for pawn in
playing.pawns]
                            if all(status):
                                end = True
                                again = False
                                points[i] += 1
```

```
                        status = [pawn.finished for pawn in playing.pawns]
                        if all(status):
                            end = True
                            again = False
                            points[i] += 1

                    i += 1
                    i = i % 4

            winner = x + np.argmax(points)
            advancing_ge.append(ge[winner])
            advancing_nets.append(nets[winner])

        for g in advancing_ge:
            g.fitness += 20
        ge = advancing_ge
        nets = advancing_nets


def run():
    p = neat.Checkpointer.restore_checkpoint('neat-checkpoint-47')

    # showing stats instead of black running screan
    p.add_reporter(neat.StdOutReporter(True))
    stats = neat.StatisticsReporter()
    p.add_reporter(stats)
    # the first, better contender
    config1 = p.config
    winner1 = p.run(main, 1)
    print('\nBest genome in 1:\n{!s}'.format(winner1))
    '''
    p = neat.Checkpointer.restore_checkpoint('neat-checkpoint-47')

    # showing stats instead of black running screan
    p.add_reporter(neat.StdOutReporter(True))
    stats = neat.StatisticsReporter()
    p.add_reporter(stats)
    # second better contender
    config2 = p.config
    '''
    local_dir = os.path.dirname(__file__)
    config_path = os.path.join(local_dir, 'config.txt')
    winner2, config2 = v1.run(config_path)
    print('\nBest genome in 1:\n{!s}'.format(winner2))

    return winner1, config1, winner2, config2


if __name__ == "__main__":
    pass
```

compare_generations.py

```
import os
import matplotlib.pyplot as plt
import neat
```

```python
import main
import choosing_genomes
import botV1.main as v1


def compare_generations(directory):
    # getting files with checkpoints
    files = []
    for filename in os.scandir(directory):
        file = filename
        name = filename.path[9:]
        if name[:4] == "neat":
            generation = name[16:]
            files.append([filename, int(generation)])

    # sorting by generation
    files = sorted(files, key=lambda f: f[1])

    # getting win rates
    win_rates = []
    # creating constant random opponent
    local_dir = os.path.dirname(__file__)
    config_path = os.path.join(local_dir, 'config.txt')
    winner2, config2 = v1.run(config_path)
    for f in files:
        path = f[0]
        p = neat.Checkpointer.restore_checkpoint(path)
        config1 = p.config
        winner1 = p.run(choosing_genomes.main, 1)
        win_rate = main.match(winner1, config1, winner2, config2, 1000)
        win_rates.append(win_rate)
        print(f"win rate: {win_rate}, generation: {f[1]}")

    generations = [f[1] for f in files]
    sub_par = []
    for x, r in enumerate(win_rates):
        if r > 35:
            sub_par.append(generations[x])
    print(sub_par)
    plt.plot(generations, win_rates, 'ro')
    plt.axis([0, 8881, 0, 100])
    plt.show()


if __name__ == "__main__":
    directory = os.path.join("..", "botv1")
    compare_generations(directory)
```

config.txt

```
[NEAT]
fitness_criterion     = max
fitness_threshold = 100
no_fitness_termination = True
pop_size              = 2
reset_on_extinction   = False
```

```
[DefaultGenome]
# node activation options
activation_default      = identity
activation_mutate_rate  = 0.1
activation_options      = clamped cube exp identity log relu sigmoid
softplus tanh

# node aggregation options
aggregation_default     = sum
aggregation_mutate_rate = 0.0
aggregation_options     = sum

# node bias options
bias_init_mean          = 0.0
bias_init_stdev         = 1.0
bias_max_value          = 100.0
bias_min_value          = -100.0
bias_mutate_power       = 0.5
bias_mutate_rate        = 0.7
bias_replace_rate       = 0.1

# genome compatibility options
compatibility_disjoint_coefficient = 1.0
compatibility_weight_coefficient   = 0.5

# connection add/remove rates
conn_add_prob           = 0.5
conn_delete_prob        = 0.5

# connection enable options
enabled_default         = True
enabled_mutate_rate     = 0.01

feed_forward            = True
initial_connection      = full_direct

# node add/remove rates
node_add_prob           = 0.2
node_delete_prob        = 0.2

# network parameters
num_inputs              = 17
num_hidden              = 0
num_outputs             = 1

# node response options
response_init_mean      = 1.0
response_init_stdev     = 0.0
response_max_value      = 100.0
response_min_value      = -100.0
response_mutate_power   = 0.0
response_mutate_rate    = 0.0
response_replace_rate   = 0.0

# connection weight options
weight_init_mean        = 0.0
weight_init_stdev       = 1.0
weight_max_value        = 200
weight_min_value        = -200
weight_mutate_power     = 0.5
weight_mutate_rate      = 0.8
```

```
weight_replace_rate      = 0.1

[DefaultSpeciesSet]
compatibility_threshold = 3.0

[DefaultStagnation]
species_fitness_func = max
max_stagnation = 20
species_elitism = 2

[DefaultReproduction]
survival_threshold = 0.2
elitism = 0
```

main.py

```
import neat.nn
from match.choosing_genomes import run
import time
from ludo.indicator import Indicator
import ludo.dice as dice
from botV1.boardV1 import Board
from botV1.agentV1 import Player
from ludo.pawn import Pawn
import numpy as np
import neat
import os


def main(nets, matches):

    colors = [(255, 0, 0), (0, 255, 0), (0, 0, 255), (255, 255, 0)]
    teams = [0, 1, 2, 3]
    win_side = 610
    win = 0
    margin = 5

    points = [0, 0, 0, 0]
    for m in range(matches):
        # one round of a tournament,
        # points for who wins
        # making one set of nets play eachother multiple times so as to
decrease luck factor
        # creating board
        board = Board(win, win_side, margin)
        # creating players
        players = [Player(colors[i], teams[i], board.starts[i],
board.finish_lines[i]) for i in range(4)]
        i = 0
        end = False
        while not end:

            playing = players[i]
            strikes = 0
            again = True
            # this is used to make multiple moves of one player possible
            while again:
```

```
                again = False
                moves = dice.throw()
                # playing player moving
                # activating net of x + indx of player from ge
                # activating by position of every pawn
                candidates = []
                chosen = False
                for pawn in playing.pawns:
                    pawn.movable(moves)
                    if pawn.possible or (pawn.position == -1 and moves ==
6):
                        candidates.append(pawn)

                if len(candidates) > 0:
                    if len(candidates) == 1:
                        chosen = candidates[0]
                        chosen.move(moves)

                    elif len(candidates) > 1:
                        states = []
                        # this is a loop which creates a set of dummy
states
                        for candidate in candidates:
                            # cooping strikes
                            c_strikes = strikes
                            # creating and moving a copy of a pawn not to
disrupt the original game
                            chosen = Pawn(candidate.color, candidate.team,
candidate.index)
                            chosen.position = int(candidate.position)
                            chosen.possible = True
                            chosen.move(moves)
                            # checking for conflicts
                            conflict, potential_casualties =
board.path.find_conflictsV1(chosen)
                            # creating state
                            state = []
                            for k in range(16):
                                pawn = players[(i + k // 4) % 4].pawns[k %
4]
                                # if conflict and in potential casualties
pawn should be teleported
                                # to the beggining
                                # but since we do not actually move pawns
just inserting -1 into state
                                if conflict and pawn in
potential_casualties:
                                    state.append(-1)
                                elif pawn.finishing == 1:
                                    state.append(pawn.position + 52)
                                # this pawn is the one moved in this
scenario therefore value of the copy is
                                # different from this of the actual pawn
                                elif pawn in playing.pawns and pawn.index
== chosen.index:
                                    state.append(chosen.position)
                                else:
                                    state.append(pawn.position)

                            # changing strikes if need be, but only a copy
                            if chosen and (chosen.finished or conflict):
```

```
                                c_strikes = 0

                            state = tuple(state + [c_strikes])
                            states.append(state)

                        # getting output from net for every state
                        outputs = [nets[i].activate(state) for state in
states]
                        index = np.argmax(outputs)

                        # choosing and moving the choice, this already has
impact on the game
                        chosen = candidates[index]
                        chosen.move(moves)

                    # actual conflict resolution
                    if chosen.finished or
board.path.find_conflicts(chosen):
                        again = True
                        strikes = 0

                    # moves pawns back into starting positions if they were
taken out and updates finish lines
                    for player in players:
                        player.update()

                    # after moving and conflicts updating path since final
and starting where updated before
                    on_board = []
                    for player in players:
                        for pawn in player.pawns:
                            if pawn.position != -1 and not pawn.finishing
and not pawn.finished:
                                on_board.append(pawn)

                    board.path.update(on_board)

                    # checking if again previously handled but player.move
but in this version
                    # it just does not make sense
                    if moves == 6:
                        strikes += 1
                        again = True

                status = [pawn.finished for pawn in playing.pawns]
                if all(status):
                    end = True
                    again = False
                    points[i] += 1

            i += 1
            i = i % 4

    winner_index = np.argmax(points)
    print(f"winner has index: {winner_index}\npoints are: {points}\n"
          f"winning percatage of the winner is:
{((points[winner_index])/matches) * 100}\n"
          f"out of {matches}")
    return ((points[winner_index])/matches) * 100
```

```python
def match(g1, config1, g2, config2, matches):

    # creating players and their nets
    nets = []

    # creating the rest
    for _ in range(3):
        nets.append(neat.nn.FeedForwardNetwork.create(g2, config2))

    # creating net of the better player
    nets.append(neat.nn.FeedForwardNetwork.create(g1, config1))

    win_rate = main(nets, matches)

    return win_rate


if __name__ == "__main__":
    g1, config1, g2, config2 = run()
    match(g1, config1, g2, config2, 5000)
```

main_visual.py

```python
import neat.nn
from match.choosing_genomes import run
import time
from ludo.indicator import Indicator
import ludo.dice as dice
from botV1.boardV1 import Board
from botV1.agentV1 import Player
from ludo.pawn import Pawn
import numpy as np
import neat
import os
import pygame

clock = pygame.time.Clock()
pygame.init()
pygame.font.init()
frame_rate = 120


def main(nets, matches):

    colors = [(255, 0, 0), (0, 255, 0), (0, 0, 255), (255, 255, 0)]
    teams = [0, 1, 2, 3]
    win_side = 610
    win = pygame.display.set_mode((win_side, win_side))
    margin = 5

    points = [0, 0, 0, 0]
    for m in range(matches):
        # one round of a tournament,
        # points for who wins
        # making one set of nets play eachother multiple times so as to
decrease luck factor
        # creating board
        board = Board(win, win_side, margin)
```

```
        # creating players
        players = [Player(colors[i], teams[i], board.starts[i],
board.finish_lines[i]) for i in range(4)]
        i = 0
        end = False
        while not end:

            playing = players[i]
            strikes = 0
            again = True
            # this is used to make multiple moves of one player possible
            while again:
                # pygame stuff
                for event in pygame.event.get():
                    if event.type == pygame.QUIT:
                        pygame.quit()
                board.draw()
                pygame.display.update()
                clock.tick(frame_rate)

                # adding ability to pouse the game under p
                pressed = pygame.key.get_pressed()
                if pressed[pygame.K_p]:
                    time.sleep(30)

                again = False
                moves = dice.throw()
                # playing player moving
                # activating net of x + indx of player from ge
                # activating by position of every pawn
                candidates = []
                chosen = False
                for pawn in playing.pawns:
                    pawn.movable(moves)
                    if pawn.possible or (pawn.position == -1 and moves ==
6):
                        candidates.append(pawn)

                if len(candidates) > 0:
                    if len(candidates) == 1:
                        chosen = candidates[0]
                        chosen.move(moves)

                    elif len(candidates) > 1:
                        states = []
                        # this is a loop which creates a set of dummy
states
                        for candidate in candidates:
                            # cooping strikes
                            c_strikes = strikes
                            # creating and moving a copy of a pawn not to
disrupt the original game
                            chosen = Pawn(candidate.color, candidate.team,
candidate.index)
                            chosen.position = int(candidate.position)
                            chosen.possible = True
                            chosen.move(moves)
                            # checking for conflicts
                            conflict, potential_casualties =
board.path.find_conflictsV1(chosen)
                            # creating state
```

```
                            state = []
                            for k in range(16):
                                pawn = players[(i + k // 4) % 4].pawns[k %
4]
                                # if conflict and in potential casualties
pawn should be teleported
                                # to the beggining
                                # but since we do not actually move pawns
just inserting -1 into state
                                if conflict and pawn in
potential_casualties:
                                    state.append(-1)
                                elif pawn.finishing == 1:
                                    state.append(pawn.position + 52)
                                # this pawn is the one moved in this
scenario therefore value of the copy is
                                # different from this of the actual pawn
                                elif pawn in playing.pawns and pawn.index
== chosen.index:
                                    state.append(chosen.position)
                                else:
                                    state.append(pawn.position)
                            # changing strikes if need be, but only a copy
                            if chosen and (chosen.finished or conflict):
                                c_strikes = 0

                            state = tuple(state + [c_strikes])
                            states.append(state)

                        # getting output from net for every state
                        outputs = [nets[i].activate(state) for state in
states]
                        index = np.argmax(outputs)

                        # choosing and moving the choice, this already has
impact on the game
                        chosen = candidates[index]
                        chosen.move(moves)

                    # actual conflict resolution
                    if chosen.finished or
board.path.find_conflicts(chosen):
                        again = True
                        strikes = 0

                    # moves pawns back into starting positions if they were
taken out and updates finish lines
                    for player in players:
                        player.update()

                    # after moving and conflicts updating path since final
and starting where updated before
                    on_board = []
                    for player in players:
                        for pawn in player.pawns:
                            if pawn.position != -1 and not pawn.finishing
and not pawn.finished:
                                on_board.append(pawn)

                    board.path.update(on_board)
```

```
                    # checking if again previously handled but player.move
but in this version
                    # it just does not make sense
                    if moves == 6:
                        strikes += 1
                        again = True

                status = [pawn.finished for pawn in playing.pawns]
                if all(status):
                    end = True
                    again = False
                    points[i] += 1

            # printing the board
            board.draw()
            pygame.display.update()
            i += 1
            i = i % 4

    winner_index = np.argmax(points)
    print(f"winner has index: {winner_index}\npoints are: {points}\n"
          f"winning percatage of the winner is:
{((points[winner_index])/matches) * 100}\n"
          f"out of {matches}")


def match(g1, config1, g2, config2, matches):

    # creating players and their nets
    nets = []
    # creating net of the better player
    nets.append(neat.nn.FeedForwardNetwork.create(g1, config1))

    # creating the rest
    for _ in range(3):
        nets.append(neat.nn.FeedForwardNetwork.create(g2, config2))

    main(nets, matches)


if __name__ == "__main__":
    g1, config1, g2, config2 = run()
    match(g1, config1, g2, config2, 100)
```

saved_rolls.py

```
import random

import neat.nn
from match.choosing_genomes import run
import time
from ludo.indicator import Indicator
import ludo.dice as dice
from botV1.boardV1 import Board
from botV1.agentV1 import Player
from ludo.pawn import Pawn
import numpy as np
import neat
```

```
from random import randint
import os


def main(nets, matches):

    colors = [(255, 0, 0), (0, 255, 0), (0, 0, 255), (255, 255, 0)]
    teams = [0, 1, 2, 3]
    win_side = 610
    win = 0
    margin = 5

    points = [0, 0, 0, 0]
    for m in range(matches):
        # one round of a tournament,
        # points for who wins
        # making one set of nets play eachother multiple times so as to
decrease luck factor
        # creating list of rolls
        # creating random number which starts pseudo random sequence
        random.seed()
        rolls = [randint(1, 6) for _ in range(1000)]
        for shift in range(4):
            # shifting nets
            nets_shifted = nets[shift:] + nets[:shift]
            # creating board
            board = Board(win, win_side, margin)
            # creating players
            players = [Player(colors[i], teams[i], board.starts[i],
board.finish_lines[i]) for i in range(4)]
            i = 0
            roll_counter = 0
            end = False
            while not end:

                playing = players[i]
                strikes = 0
                again = True
                # this is used to make multiple moves of one player
possible
                while again:

                    again = False
                    moves = rolls[roll_counter]
                    # playing player moving
                    # activating net of x + indx of player from ge
                    # activating by position of every pawn
                    candidates = []
                    chosen = False
                    for pawn in playing.pawns:
                        pawn.movable(moves)
                        if pawn.possible or (pawn.position == -1 and moves
== 6):
                            candidates.append(pawn)

                    if len(candidates) > 0:
                        if len(candidates) == 1:
                            chosen = candidates[0]
                            chosen.move(moves)

                        elif len(candidates) > 1:
```

```
                            states = []
                            # this is a loop which creates a set of dummy
states
                            for candidate in candidates:
                                # cooping strikes
                                c_strikes = strikes
                                # creating and moving a copy of a pawn not
to disrupt the original game
                                chosen = Pawn(candidate.color,
candidate.team, candidate.index)
                                chosen.position = int(candidate.position)
                                chosen.possible = True
                                chosen.move(moves)
                                # checking for conflicts
                                conflict, potential_casualties =
board.path.find_conflictsV1(chosen)
                                # creating state
                                state = []
                                for k in range(16):
                                    pawn = players[(i + k // 4) %
4].pawns[k % 4]
                                    # if conflict and in potential
casualties pawn should be teleported
                                    # to the beggining
                                    # but since we do not actually move
pawns just inserting -1 into state
                                    if conflict and pawn in
potential_casualties:
                                        state.append(-1)
                                    elif pawn.finishing == 1:
                                        state.append(pawn.position + 52)
                                    # this pawn is the one moved in this
scenario therefore value of the copy is
                                    # different from this of the actual
pawn
                                    elif pawn in playing.pawns and
pawn.index == chosen.index:
                                        state.append(chosen.position)
                                    else:
                                        state.append(pawn.position)

                                # changing strikes if need be, but only a
copy
                                if chosen and (chosen.finished or
conflict):
                                    c_strikes = 0

                                state = tuple(state + [c_strikes])
                                states.append(state)

                            # getting output from net for every state
                            outputs = [nets_shifted[i].activate(state) for
state in states]

                            index = np.argmax(outputs)

                            # choosing and moving the choice, this already
has impact on the game
                            chosen = candidates[index]
                            chosen.move(moves)

                        # actual conflict resolution
```

```
                        if chosen.finished or
board.path.find_conflicts(chosen):
                            again = True
                            strikes = 0

                        # moves pawns back into starting positions if they
were taken out and updates finish lines
                        for player in players:
                            player.update()

                        # after moving and conflicts updating path since
final and starting where updated before
                        on_board = []
                        for player in players:
                            for pawn in player.pawns:
                                if pawn.position != -1 and not
pawn.finishing and not pawn.finished:
                                    on_board.append(pawn)

                        board.path.update(on_board)

                        # checking if again previously handled but
player.move but in this version
                        # it just does not make sense
                        if moves == 6:
                            strikes += 1
                            again = True

                    status = [pawn.finished for pawn in playing.pawns]
                    if all(status):
                        end = True
                        again = False
                        points[(i + shift) % 4] += 1
                    roll_counter += 1

                i += 1
                i = i % 4
    winner_index = np.argmax(points)
    print(f"winner has index: {winner_index}\npoints are: {points}\n"
          f"winning percatage of the winner is: {((points[winner_index]) /
(matches * 4)) * 100}\n"
          f"out of {matches}")
    return ((points[winner_index])/matches) * 100


def match(g1, config1, g2, config2, matches):

    # creating players and their nets
    nets = []

    # creating the rest
    for _ in range(3):
        nets.append(neat.nn.FeedForwardNetwork.create(g2, config2))

    # creating net of the better player
    nets.append(neat.nn.FeedForwardNetwork.create(g1, config1))

    win_rate = main(nets, matches)

    return win_rate
```

```python
if __name__ == "__main__":
    g1, config1, g2, config2 = run()
    match(g1, config1, g2, config2, 1250)
```

Generation checkpoints

NEAT-python documentation: https://neat-python.readthedocs.io/en/latest/

Original paper on the NEAT algorithm: http://nn.cs.utexas.edu/downloads/papers/stanley.cec02.pdf