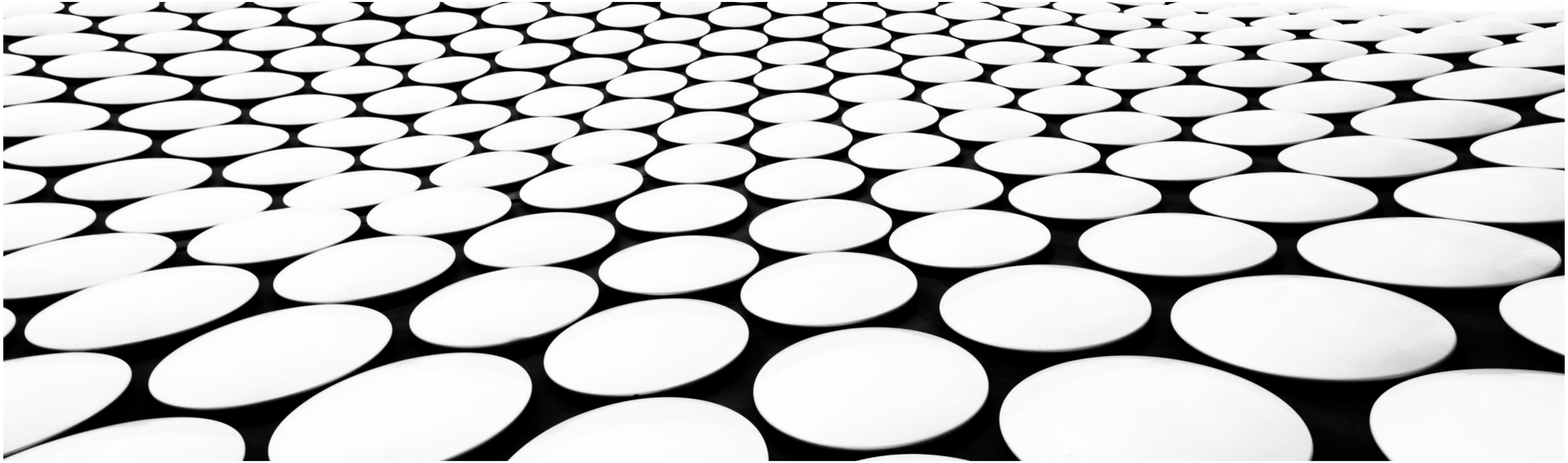
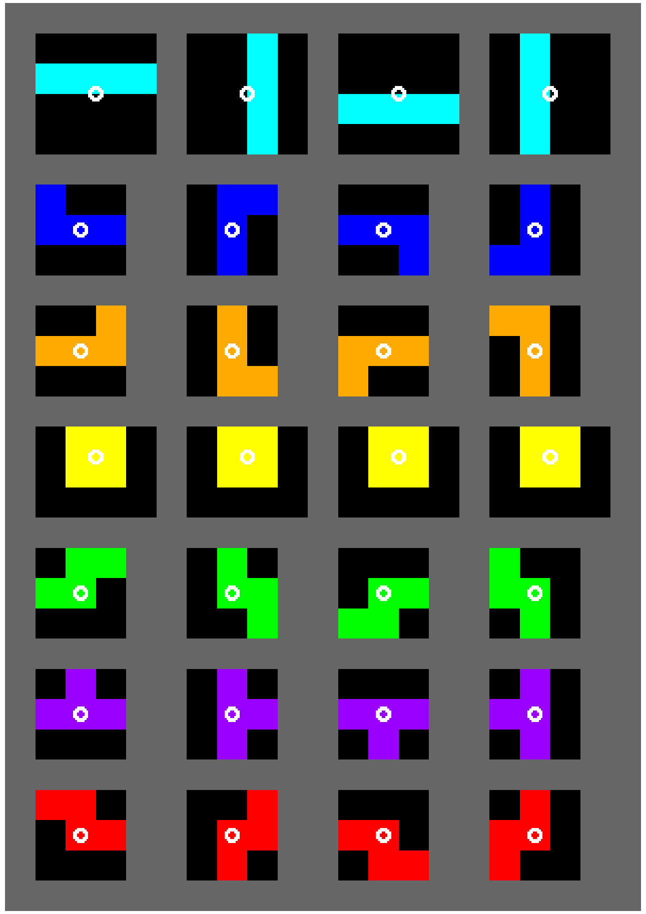
TETRIS GAME PLAYING AI USING STATIC EVALUATION FUNCTION

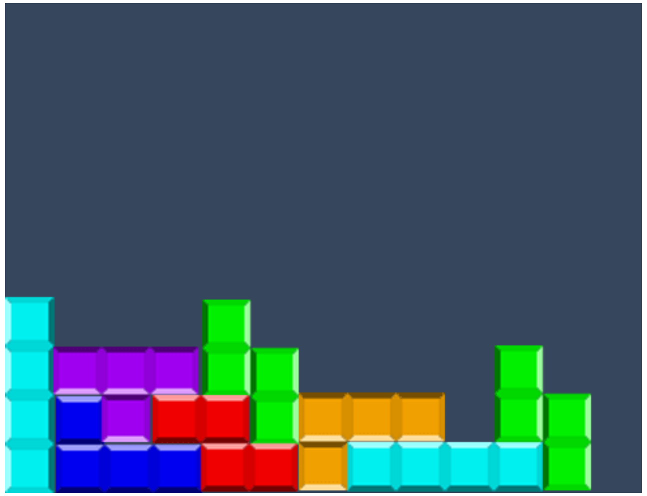
GABE BECKER



THE GAME

- Tetris is a video game where a 4-block piece, called a tetromino, falls at a constant rate from the top of a 10x 20 grid until it touches the bottom of the grid or another block
- When a row is full it disappears and all rows above it move down
- There are 7 different types of tetrominos, and it is random which one will appear
- You can do 3 things with a tetromino:
  - Move it left
  - Move it right
  - Rotate the piece





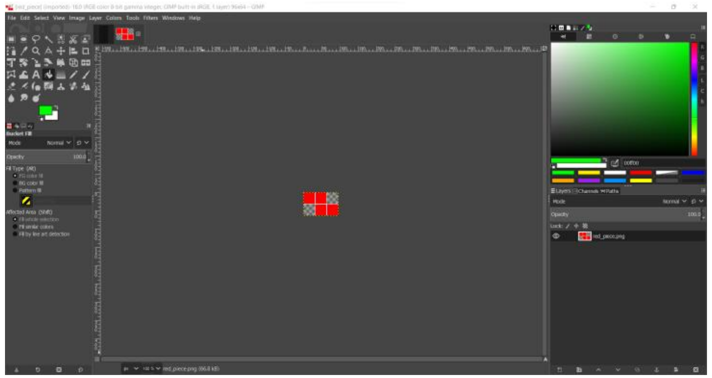
A QUICK ASIDE..

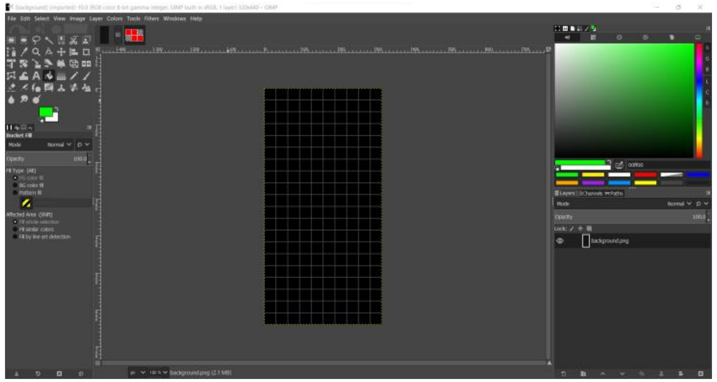
- Real Tetris uses a scoring system based on points
  - For example: clearing 3 lines with 1 move is worth more points than clearing 3 lines over 3 moves
- This Tetris AI does not focus on points
  - Too complicated to implement
- Instead, I focused on clearing as many lines as possible and surviving for as long as possible
  - Strategy: If the AI can survive forever, it will score more points

MAKING THE GAME

- All texture assets made in Gimp
  - 10x20 grid
    - Each square is 32x32 pixels
    - 320x640 pixels
- Tetris theme A – GBA
- Clear line sound effect – NES
- Game made in Python 3 with pygame

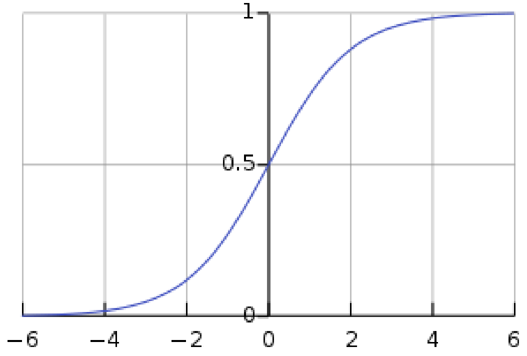
```python
# sets up pygame
pygame.init()
pygame.font.init()
pygame.mixer.init()
BlockletterBig = pygame.font.SysFont('Blockletter', 48)
BlockletterSmall = pygame.font.SysFont('Blockletter', 32)
screen = pygame.display.set_mode((320, 640))
pygame.display.set_caption("Tetris AI")
pygame.display.set_icon(pygame.image.load("./assets/blue_piece.png"))
game_background = pygame.image.load("./assets/background.png")
title_screen1 = pygame.image.load("./assets/titlescreen1.png")
title_screen2 = pygame.image.load("./assets/titlescreen2.png")
pygame.mixer.music.load("./assets/Tetris_theme.mp3")
pygame.mixer.music.set_volume(0.7)
pygame.mixer.music.play(loops=-1)
```





MAKING THE EVALUATION FUNCTION

USING THE SIGMOID FUNCTION



$$S(x) = \frac{1}{1 + e^{-x}}$$

By modifying this function and using information from the board, we can create an evaluation function

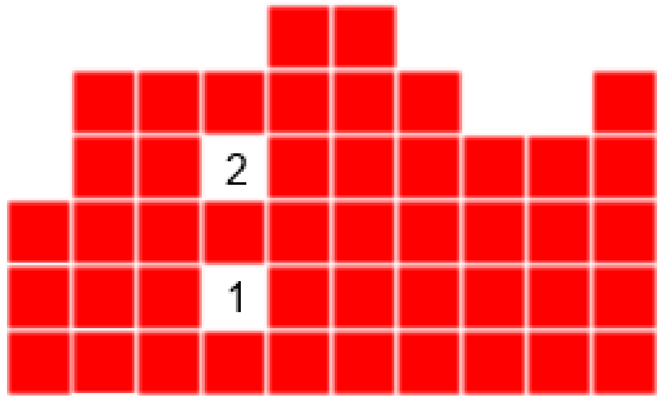
THE FUNCTION

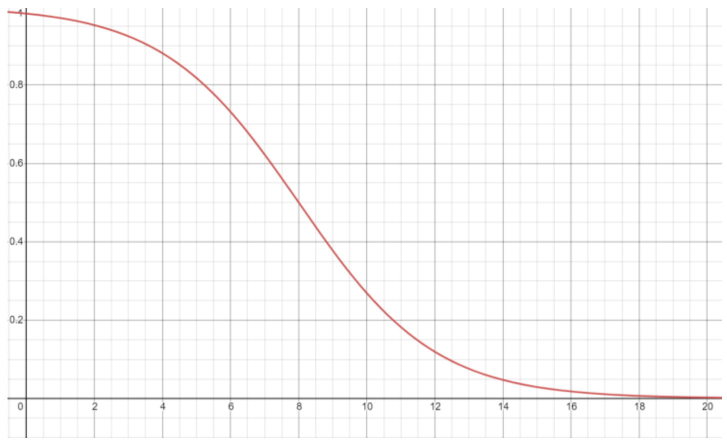
$$E(\text{board}) = w_1 A + w_2 B + w_3 C + w_4 D$$

- Where:
  - A, B, C, D are modified Sigmoid functions that represent an attribute of the board, each of which are evaluated between 0 and 1
    - A – Holes
    - B – Aggregate Height
    - C – Bumpiness
    - D – Max Height
  - w_1, w_2, w_3, w_4 are “weights” where $0 < w_i < 1$ and $\sum w_i = 1$

- This makes it so E is between 0 and 1, where 0 is the worst position and 1 is the best position

```
return (AI.eval_holes(holes) * 0.2
        + AI.eval_aggregate_height(aggregate_height) * 0.35
        + AI.eval_bumpiness(bumpiness) * 0.1
        + AI.eval_max_height(max_height) * 0.35)
```



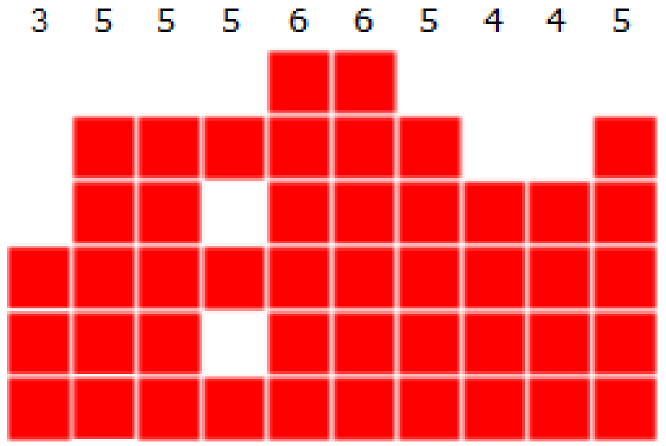


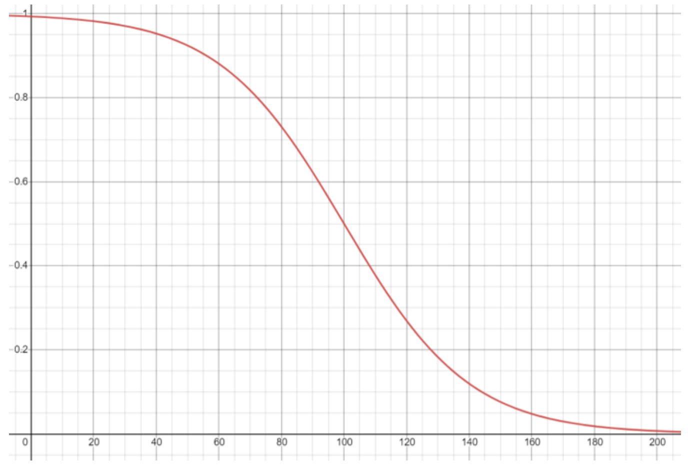

```
@staticmethod
def eval_holes(holes):

    if holes == 0:
        return 1

    return 1 / (1 + e ** ((0.5 * holes) - 4))
```

HOLES
(X = NUMBER OF HOLES)

$$A = S(-0.5x + 4) = \frac{1}{1 + e^{0.5x-4}}$$

- 0 holes - 1
- 8 holes – 0.5
- 20 holes – basically 0
- $w_1 = 0.2$



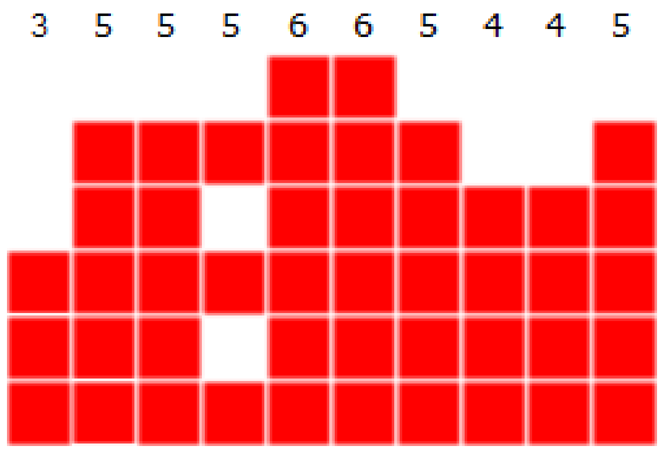


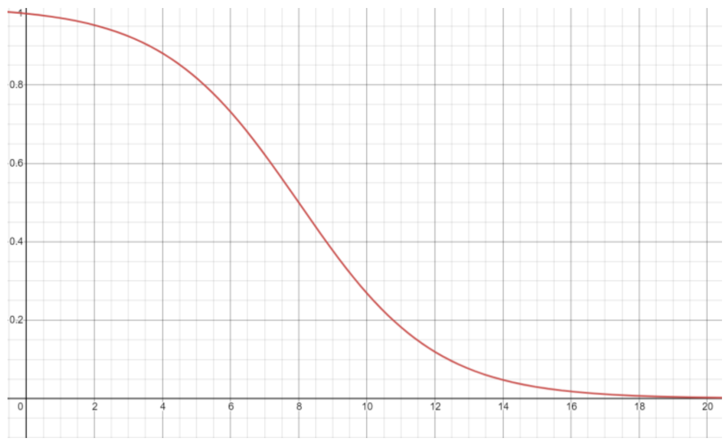

```python
@staticmethod
def eval_aggregate_height(aggregate_height):

    if aggregate_height == 0:
        return 1

    return 1 / (1 + e ** ((0.05 * aggregate_height) - 5))
```

AGGREGATE HEIGHT (X = SUM OF ALL COLUMN HEIGHTS)

$$B = S(-0.05x + 5) = \frac{1}{1 + e^{0.05x-5}}$$

- 0 aggregate height - 1
- 100 aggregate height – 0.5
- 200 aggregate height – basically 0
- $w_2 = 0.35$



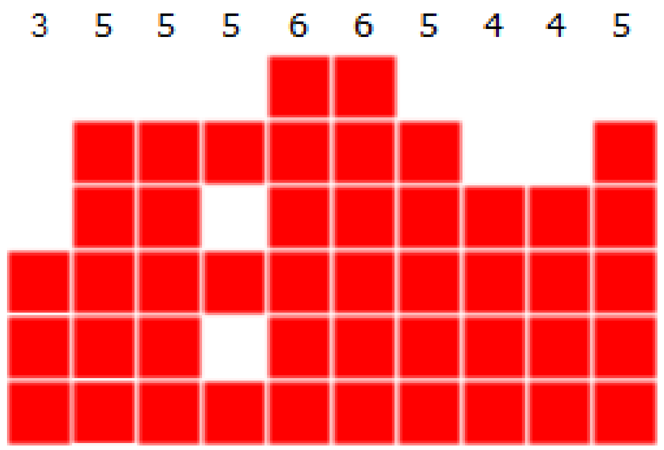


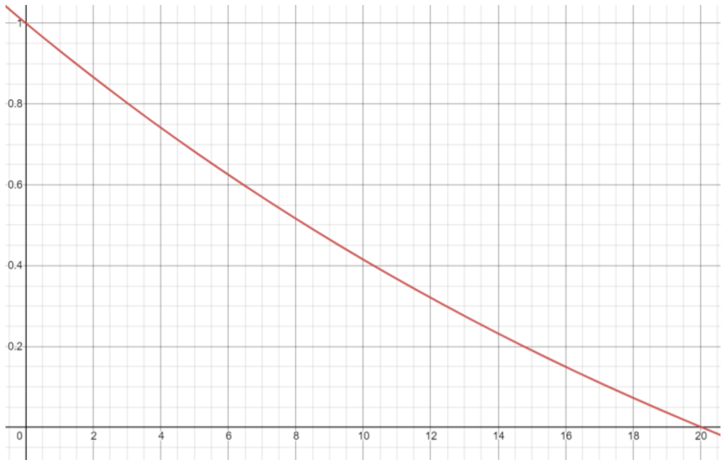

```python
@staticmethod
def eval_bumpiness(bumpiness):

    if bumpiness == 0:
        return 1

    return 1 / (1 + e ** ((0.5 * bumpiness) - 4))
```

BUMPINESS (X = SUM OF ABSOLUTE DIFFERENCES BETWEEN ADJACENT COLUMN HEIGHTS)

$$C = S(-0.5x + 4) = \frac{1}{1 + e^{0.5x-4}}$$

- 0 bumpiness - 1
- 8 bumpiness – 0.5
- 20 bumpiness – basically 0
- $w_3 = 0.1$






```python
@staticmethod
def eval_max_height(height):

    if height <= 1:
        return 1
    if height >= 20:
        return 0

    return 2 / (e ** (height * 0.0346)) - 1
```

MAX HEIGHT (X = HIGHEST COLUMN HEIGHT)

$$D = \frac{2}{e^{0.0346x}} - 1$$

- 0 max height - 1
- 8 max height – ~0.5
- 20 max height – 0
- $w_4 = 0.35$

HOW DO WE TO USE THIS?

- First, we loop through possible rotations and placements of the tetromino, and simulate “dropping” the piece
- Next, we calculate the evaluation of the board given the tetromino placement
- We then choose the placement of the tetromino that gives the best evaluation score
- Finally, we convert the best move into actions (rotation x time, move left, move right) and execute the actions

```
actions = best_move.convert_to_actions(falling_piece)
```

```
@staticmethod
def best_move(board: Board, tetromino: Tetromino):

    highest_score = 0
    best_move = Move(None, None, None)

    newTetromino = Tetromino(tetromino.color)
    for r in range(4 if tetromino.color in ['B', 'O', 'P'] else (1 if tet
        newTetromino.rotation = r
        skeleton = newTetromino.skeleton()
        for x in range(10):
            if x + len(skeleton[0]) - 1 < 10:
                newTetromino.x_pos = x

                newBoard, newTetromino = board.drop(newTetromino)
                newBoard.remove(newBoard.full_lines())

                score = AI.evaluate(newBoard)

                if highest_score < score:
                    highest_score = score
                    best_move = Move(r, x, newTetromino.y_pos)

    return best_move
```

MULTITHREADING

- To prevent blocking while deciding the best move, multithreading is used
- The main thread continues to drop the piece while a child thread calculates the move
- In retrospect, probably not needed (evaluation function is lightning fast)
  - Approximately 0.0003 seconds per evaluation
  - Really hard to measure because it’s so fast

```
falling_piece = Tetromino(colors[random.randrange(7)])
ai_thread = AiThread(target=ai.AI.best_move, args=(board, falling_piece,))  # generates a new move
calculated = False
ai_thread.start()
```

SO, DOES IT WORK???

IT'S PRETTY GOOD!

- Clears ~23 lines per game
  - Better approximation: 20-30 lines per game
  - Highest amount achieved: 90

- Making a good evaluation with only "pencil and paper" mathematics is *very* difficult
  - Using machine learning to change the values of weights and functions is definitely the way to make a proper Tetris AI

- This doesn't mean the project was a failure – the AI can make decisions usually clear some lines
  - I'd describe it as a slightly less-than-average Tetris player and better than a first-time human Tetris player
  - It makes decisions way faster than the average human, meaning it can play much faster
  - I also learned a lot about creating and using a static evaluation function, which is a success in its own way