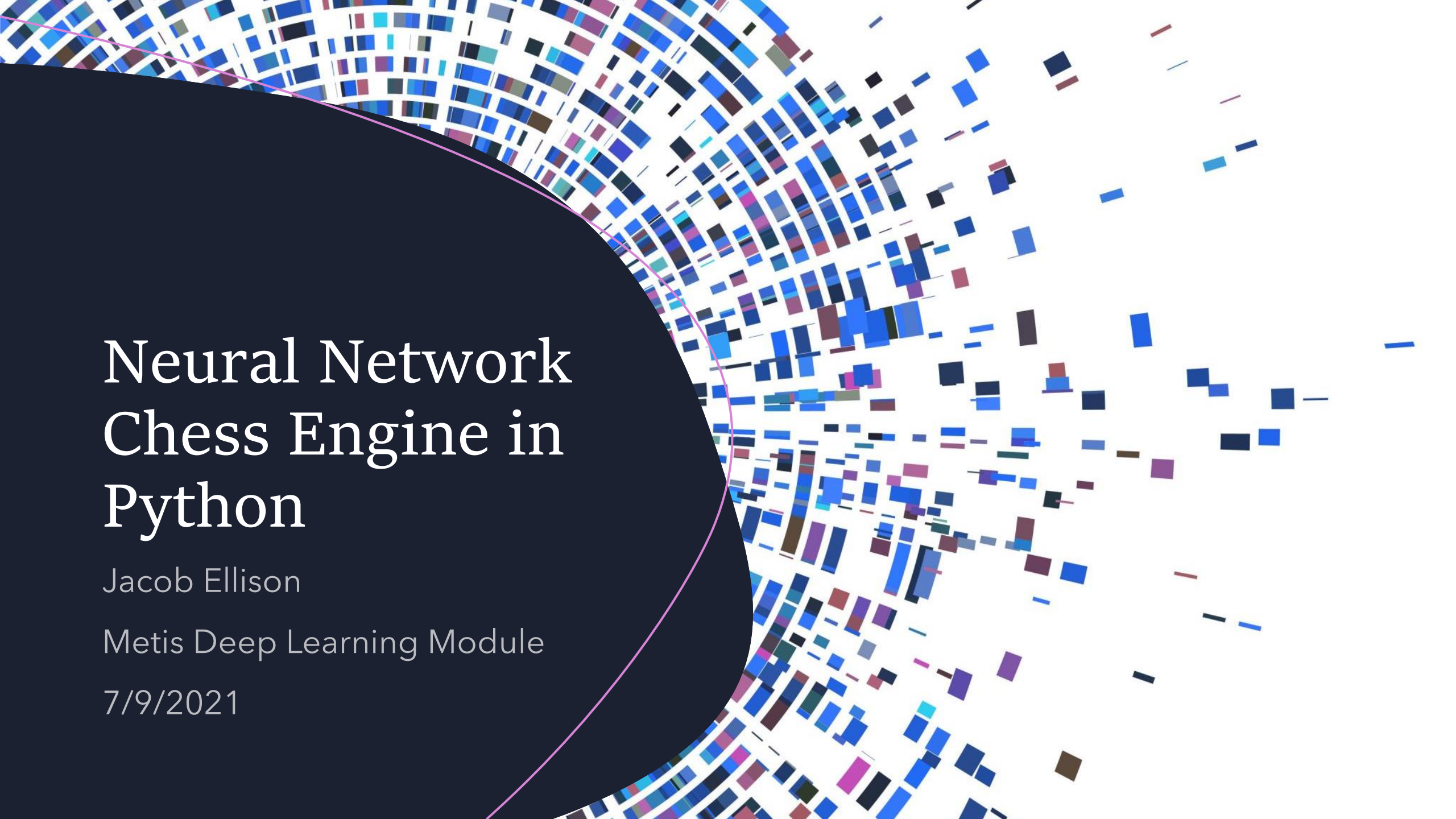


Neural Network Chess Engine in Python

Jacob Ellison

Metis Deep Learning Module

7/9/2021

Goal and Design

- Create a chess engine that can play valid moves to explore and understand machine learning models and strategies.
- I began this project knowing that I would not create a good chess engine, since reinforcement learning is necessary to show the model the reward and loss associated with a move.
- Instead, I wanted to create something where the process would be instructive, and the final model would be functional.

Data and Tools

- I used chess games played in 2020 from the FICS database.
- This dataset contains over 44k games, representing over 3M board states (training inputs).
- I also used the Python Chess library to parse board states and interpret legal moves.

https://www.ficsgames.org/download.html
https://python-chess.readthedocs.io

Data Processing

- Chess moves are normally shown in PGN notation.
- For each move, I converted the PGN move to a snapshot of the board using FEN notation.
- I then adapted the FEN notation into a numeric format and concatenated additional metadata about the board like castling rights and whether the king is in check.

ASCII BOARD

```
r n b q k b n r
p p p p p p p p
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
P P P P P P P P
R N B Q K B N R
```

PGN Notation

1. e4 e5 2. Nf3 Nc6 3. Bb5 a6 4. Ba4 Nf6 5. O-O Be7 6. Re1 b5 7. Bb3 d6 8. c3 O-O 9. h3 Nb8 10. d4 Nbd7

First 10 moves contain 20 different board states. Convert each board state to ...

FEN Notation

rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1

Adapt the FEN notation into numeric inputs (len 64)

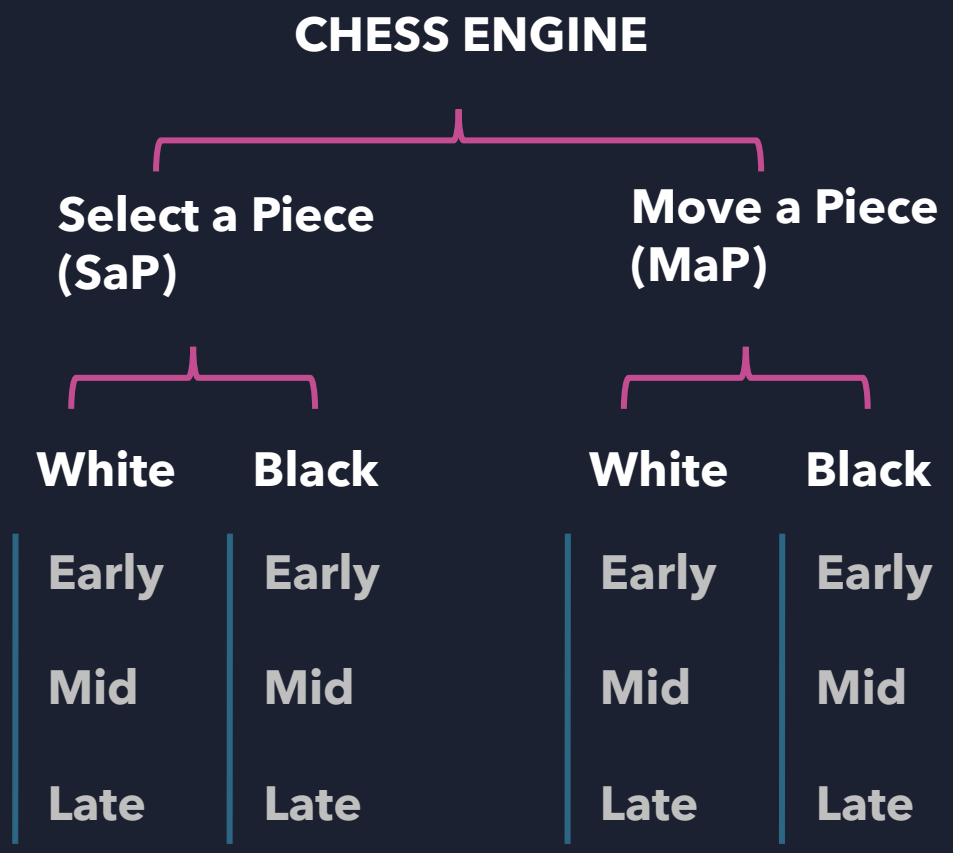
[-5, -3, -4, -9, -10, -4, -3, -5, -1, -1, -1, -1, -1, -1, -1, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 5, 3, 4, 9, 10, 4, 3, 5]

General Strategy

- Inputs to the neural network are the processed numeric board states.
- Target to the network is the square to move a piece to.
- Implementation challenge:
  - The neural network has no idea about which moves are legal. For the eight starting pieces on the board, the model might try to move each to one of the 64 squares on the 8x8 chess board = 8 * 64, or 512 possibilities.
  - However, the model has no notion of a piece, so it might try to move one of the opponent's pieces, or even an empty square. 64 possible pieces moving to 64 possible squares = 64 * 64, or 4,096 possibilities.
- I wanted to simplify the output decision space.

Specific Implementation

- Split up the chess engine into neural networks trained to perform specific tasks. In doing so, improve the accuracy for these tasks and simply the decision space.
- I broke up the decision to select a piece and move a piece so that each decision could be represented with 64 outputs rather than their product which required 4,096 outputs.
- I also split out models on piece color and game stage. In total, this meant creating 12 different models



Validation Performance

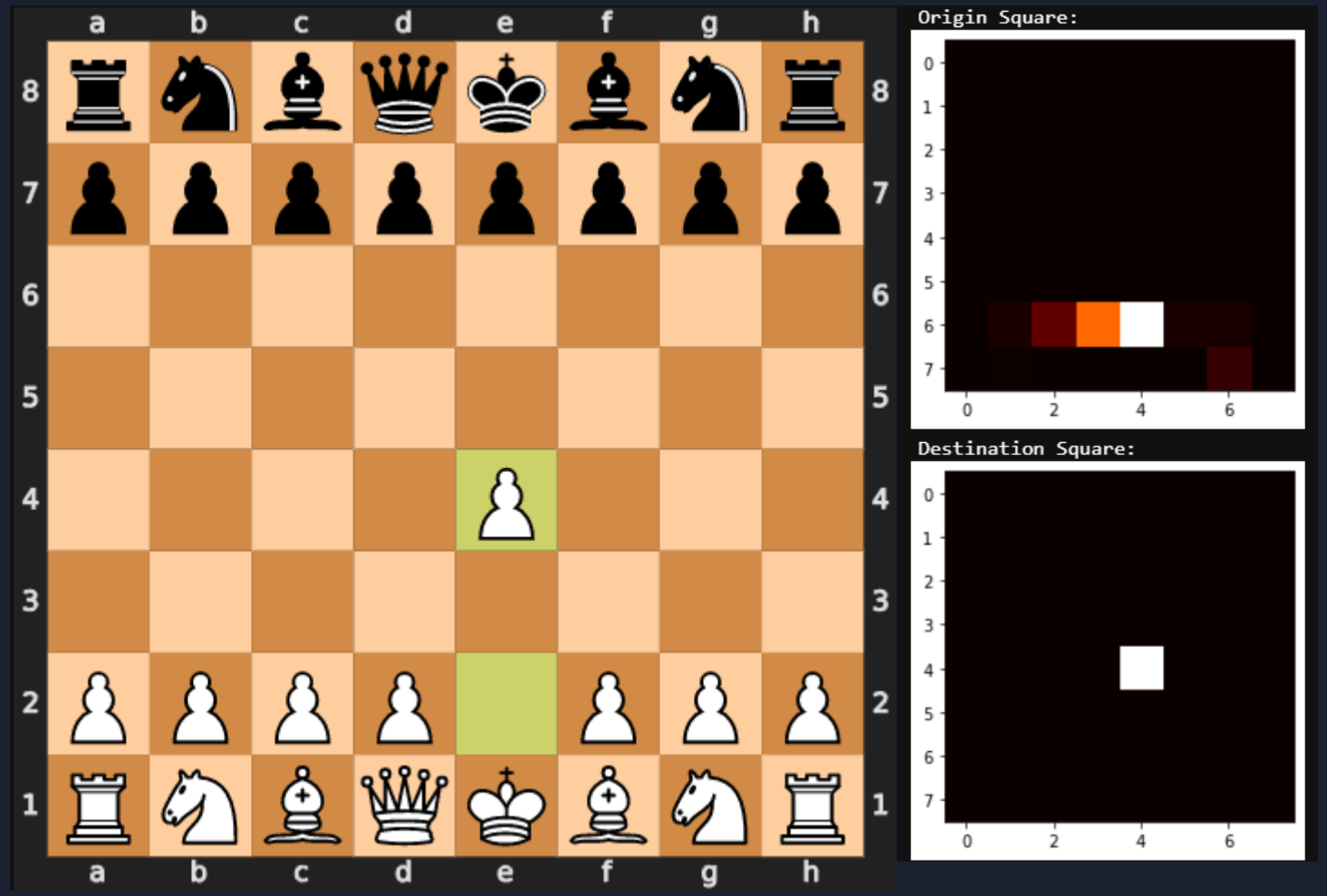
Model	Stage	Loss	Acc
SaP	Early	1.15	0.61
SaP	Mid	1.98	0.34
SaP	Late	1.61	0.40
MaP	Early	0.32	0.89
MaP	Mid	1.74	0.47
MaP	Late	2.90	0.19

- Compared against one another, the SaP models are more accurate in the late game, and the MaP models are more accurate early.
- Intuitively, this makes sense: in the early game, there are many pieces that could be moved given a certain board state, but once a piece is chosen, the number of moves is limited (the knight has only 2 available moves on its starting square).
- In the late game, each piece will usually have more moves available (the knight in the center of the board has 8 possible moves), but there are fewer pieces to move because many will have been captured.

So How Does It Play??

- In short, poorly. However, I knew this coming into the project. These models are optimized by looking at a board and trying to predict a move played by a high-rated player. With no sense of piece evaluation or ability to look ahead, this model can be beaten easily.
- In the following slides, I'll show the initial moves from a game I played against the engine. The two heatmaps show the output from the SaP and MaP models. As a recap, the SaP predicts the origin square (where to move from), and the MaP model predicts the destination square (where to move to).
- My chess engine will be playing with the white pieces.

Move 1w

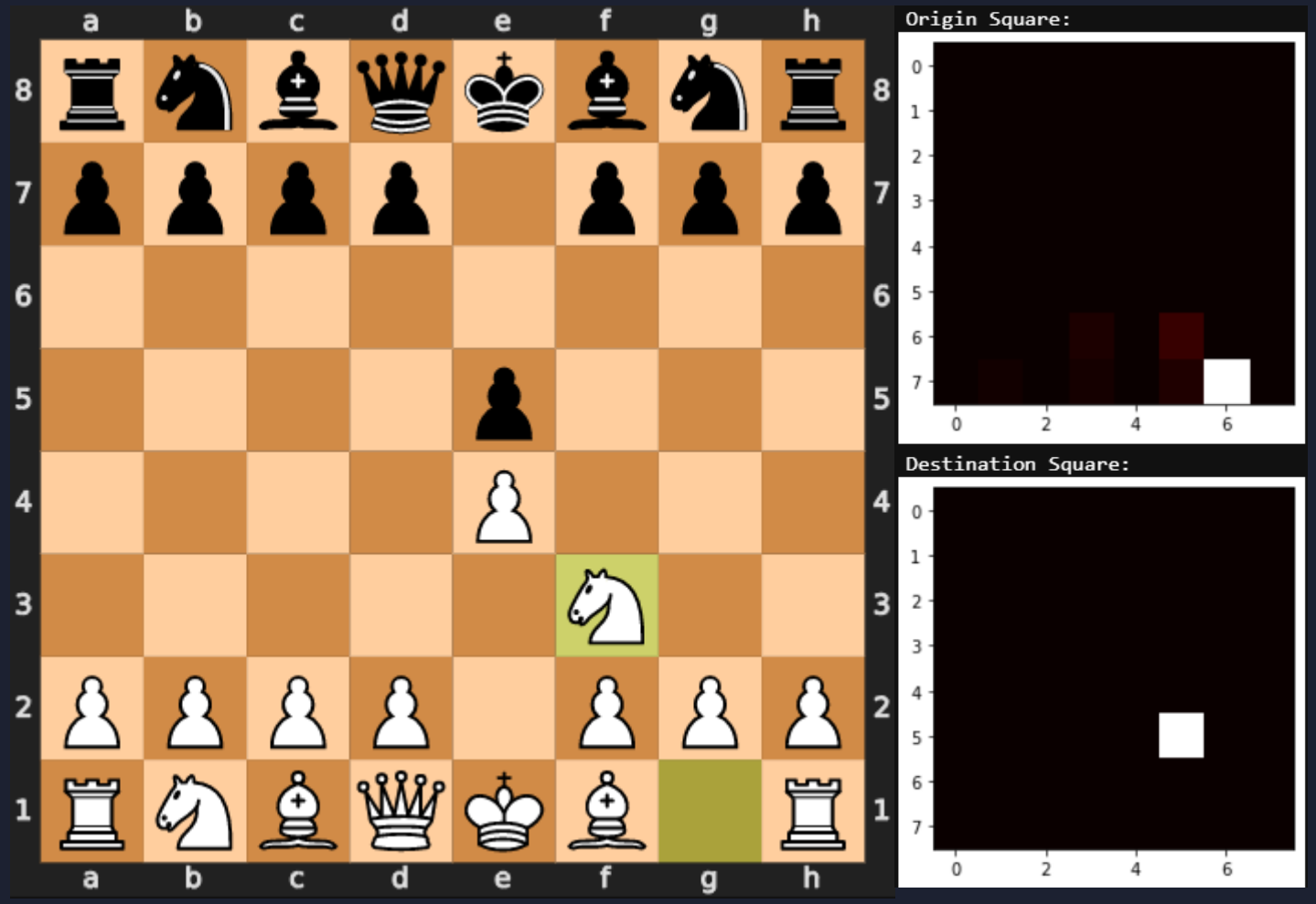

a b c d e f g h
8 7 6 5 4 3 2 1

Origin Square:
0 1 2 3 4 5 6 7
0 2 4 6

Destination Square:
0 1 2 3 4 5 6 7
0 2 4 6

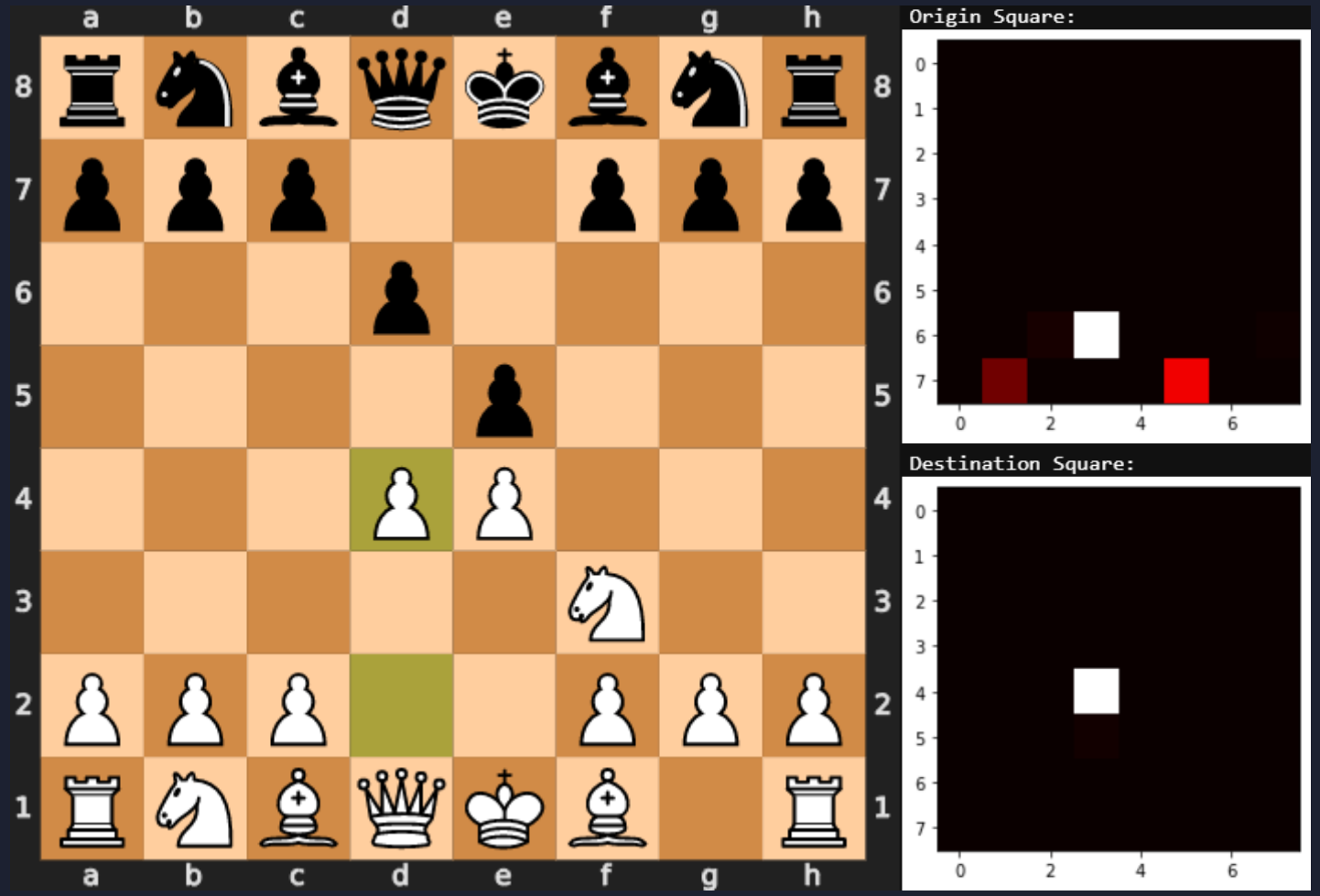
Move 1b

Move 2w


a b c d e f g h
8 7 6 5 4 3 2 1
Origin Square:
Destination Square:
0 1 2 3 4 5 6 7
0 2 4 6

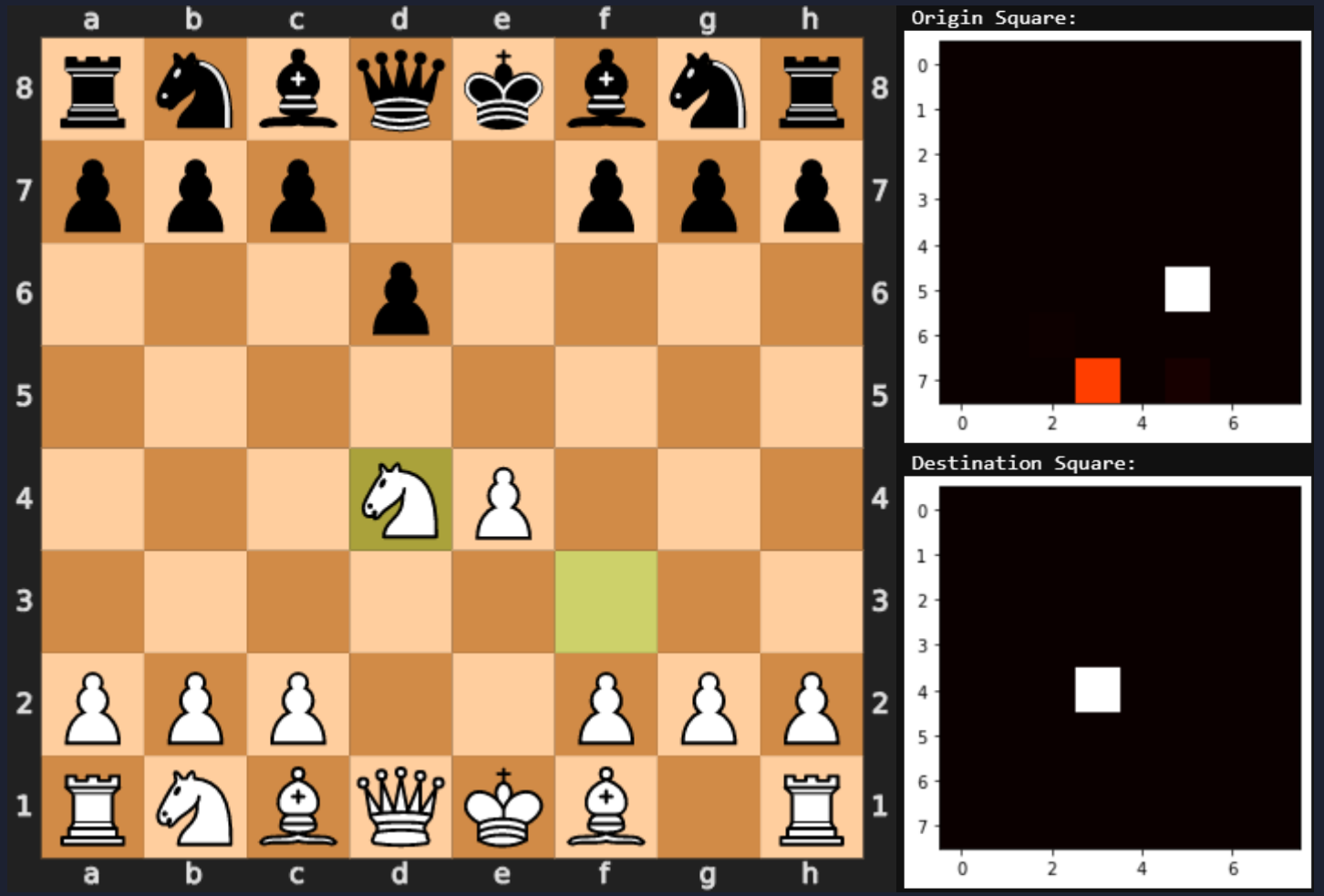
Move 2b

Move 3w


a b c d e f g h
8 7 6 5 4 3 2 1
Origin Square:
Destination Square:
0 1 2 3 4 5 6 7
0 2 4 6

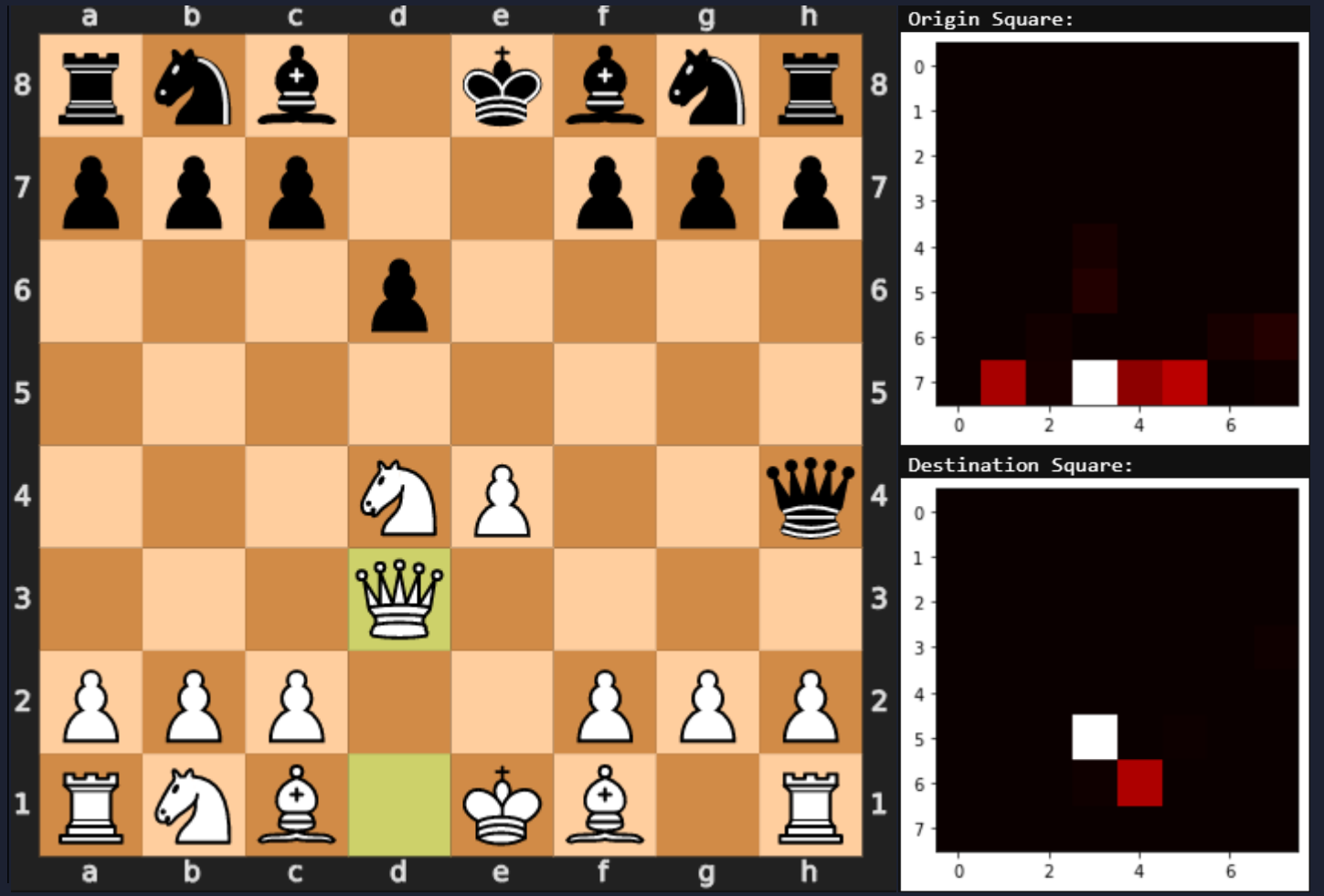
Move 3b

Move 4w

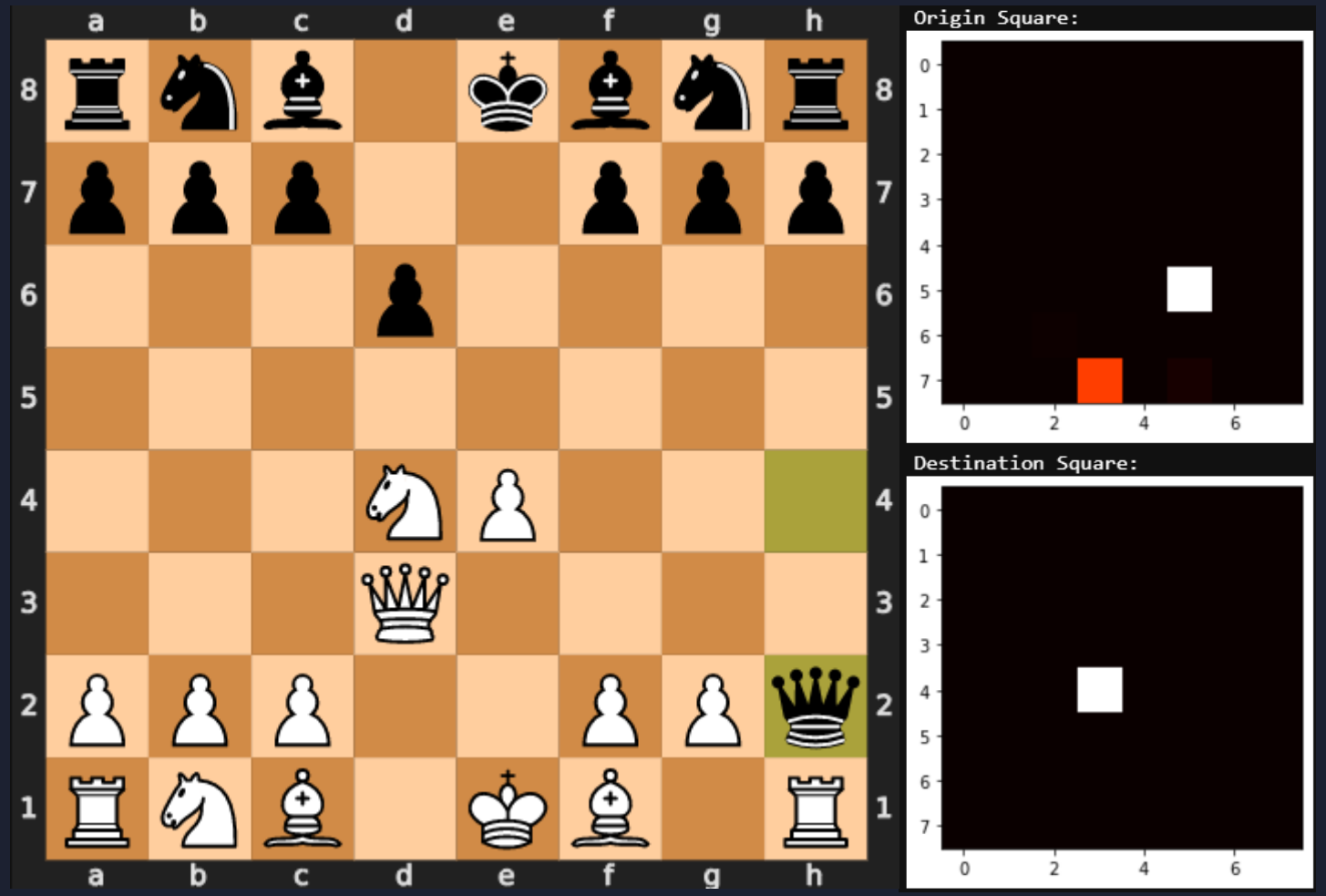

a b c d e f g h
8 7 6 5 4 3 2 1
Origin Square:
Destination Square:
0 1 2 3 4 5 6 7
0 2 4 6

Move 4b

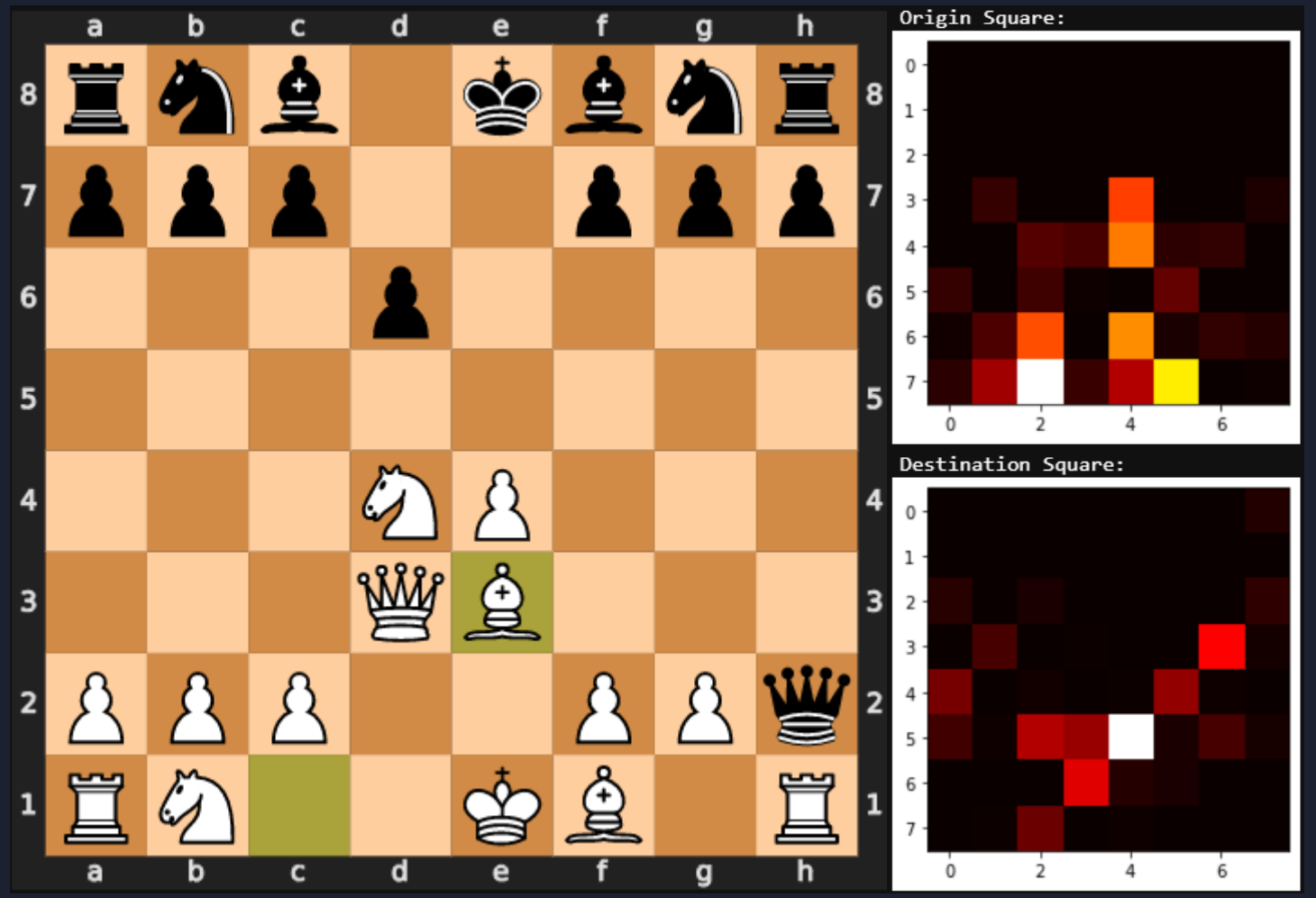
Move 5w


a b c d e f g h
8 7 6 5 4 3 2 1
Origin Square:
Destination Square:
0 1 2 3 4 5 6 7
0 2 4 6

Move 5b

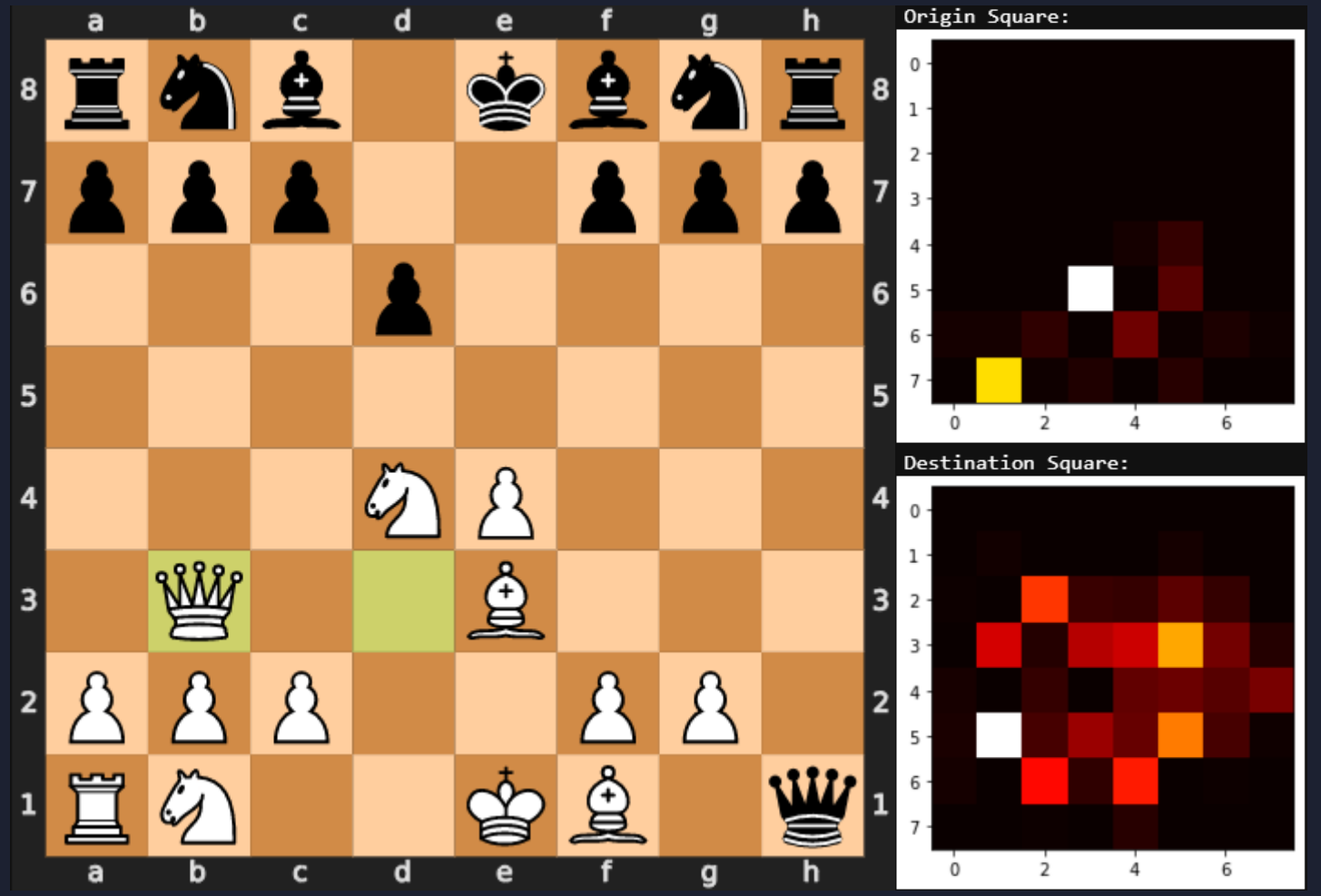

a b c d e f g h
8 7 6 5 4 3 2 1
Origin Square:
Destination Square:
0 1 2 3 4 5 6 7
0 2 4 6

Move 6w


a b c d e f g h
8 7 6 5 4 3 2 1
Origin Square:
Destination Square:
0 1 2 3 4 5 6 7
0 2 4 6

Move 6b

Move 7w


a b c d e f g h
8 7 6 5 4 3 2 1
Origin Square:
Destination Square:
0 1 2 3 4 5 6 7
0 2 4 6

Evaluation

- The engine's strengths are in the early game when many of the moves are formulaic and there are direct, known responses to many lines of play.
- It becomes very weak when it is pushed outside of it's training data. For instance, black's queen sacrifice on move 5 is a bad move. However, that move is not something that was played in the training data, so the model doesn't have a good response.

Potential Improvements & Next Steps

- I built this model using only dense layers, however I'd like to try adding convolutional layers to see if they will help the engine see useful move patterns anywhere on the board (like finding an easy queen capture on a nonstandard square).
- Currently I move from early -> mid -> late game by tracking turn count. Instead, I'd like to try advancing the game stage based on pieces remaining on the board.
- I would like to try splitting out the MaP neural networks into separate models for each piece. The SaP model might choose a pawn move, for instance, and I'll then pass the MaP decision to a NN built with only pawn-move training data. I think that this change might reduce the amount of piece-shuffling in the late game.