VILNIAUS UNIVERSITY OF APPLIED SCIENCES

FACULTY OF ELECTRONICS AND INFORMATICS

SOFTWARE ENGINEERING DEPARTEMENT

Artificial Intelligence

Neural Network Chess Engine Project 6531BX028 PI18E

Evaldas Paulauskas

STUDENT

(Signatare)
2021.__.__

Lukaš Jutkevič

D. Savulionis

LECTURER

(Signatare)
2021.__.__

2021

Table of Contents

INTRODUCTION

Neural-network based chess engine that runs as flask web-app in users' browser.

The goal of the project.

The goal of the project was to learn and explore how to create, train, and use convolutional neural networks by using one to create playable chess engine.

Project requirements.

- User should be able to play chess in the browser.
- Chess engine evaluation function should use convolutional neural network for board state evaluation.

Technologies used.

Project was created using Python 3.8 and various libraries were used:

- Python-chess – was used to simplify the creation of the engine.
- Flask – this web framework was used to enable playing chess with the computer in the browser.
- PyTorch – was used to create and train the convolutional neural network used in evaluation function.

ALGORITHMS USED

How does computer play chess? The easiest way to program the chess engine is to create the evaluation function that would rate the current chess board state (in this case board state would be position of all pieces and whose turn is it) and predict which one of the players has favourable position.

Evaluation function.

This evaluation function can be created by training neural network to recognise who is wining and rating players chance to win in current board state.

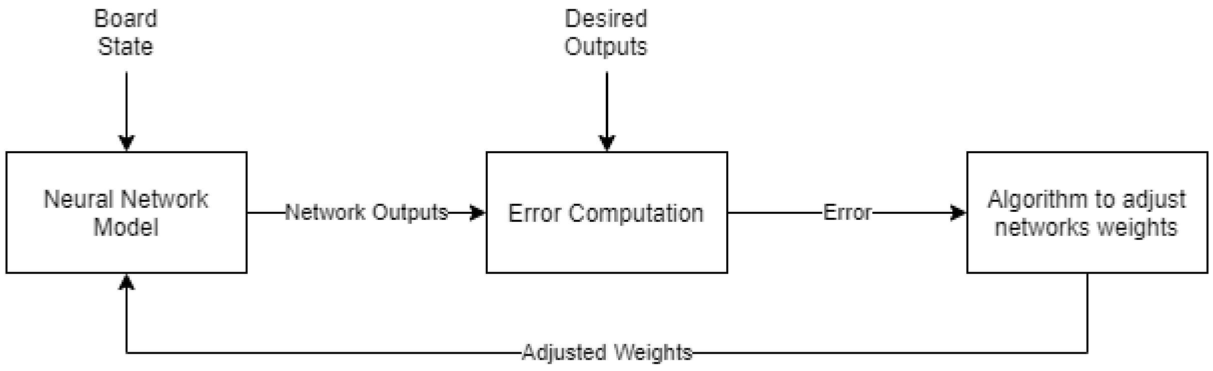


Figure 1 Training Neural Network

Image above shows how to train neural network when board state and desired outputs are given and loop above is repeated multiple times neural networks weights are adjusted to closely match desired outputs. When network output deviation from desired outputs is low enough, we can use this trained neural network in engines evaluation function.

Search Algorithm.

Having created the evaluation function that takes in board state as and input and outputs rating of the state we can create she search tree by making every possible move and rating it using evaluation function and then taking new states of the board after making moves for the opponent and then rating the board state. This search tree can be several levels deep but its mainly limited my computers computational power since number of moves made grows exponentially. For example, there is number of moves from begging of the chess game:

- Depth: 1 (Whites turn) – 20 possible moves.
- Depth: 2 (Black turn) – 400 possible moves.
- Depth 3 (Whites turn) – 8902 possible moves.
- Depth 4 (Blacks turn) – 197281 possible moves.
- Depth 5 (Whites turn) – 4865609 possible moves.

- Depth 6 (Whites turn) – 119060324 possible moves.

So, building search trees with large depths is unpractical since it takes a lot of computational power and time to calculate and evaluate all possible moves.

To make the best move we need to apply the minmax algorithm to the tree. It will calculate which moves are in computers favour.

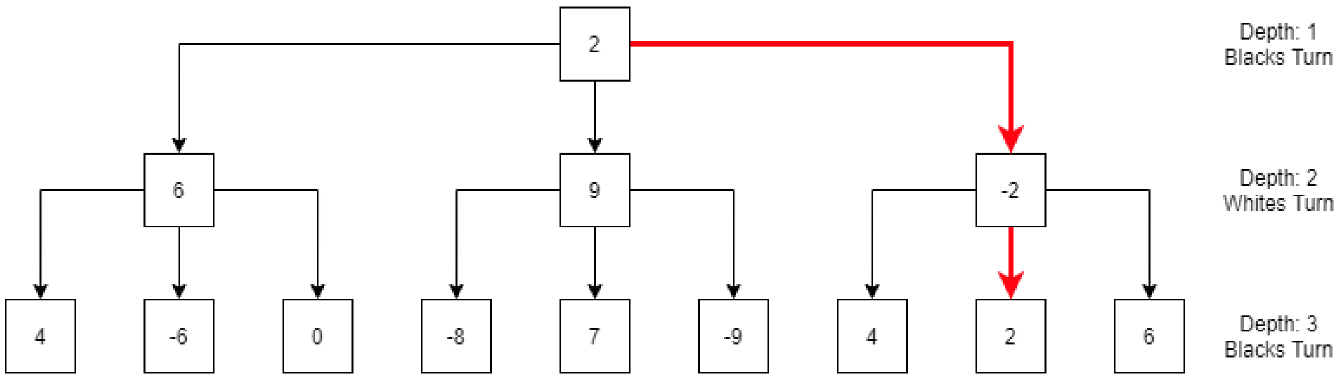


Figure 2 Example of how minmax works.

Image above shows how minmax algorithm works. If each node represents the value of board for black and we assume that white will always make the best moves (values that are negative represent favourable moves for white), black will try to maximise its chance to win by picking moves that in 3 turns will give the highest value.

Optimising the search algorithm

To make minmax algorithm faster we can use alpha beta pruning. It is a search algorithm that seeks to decrease the number of nodes that are evaluated by the minimax algorithm in its search tree.

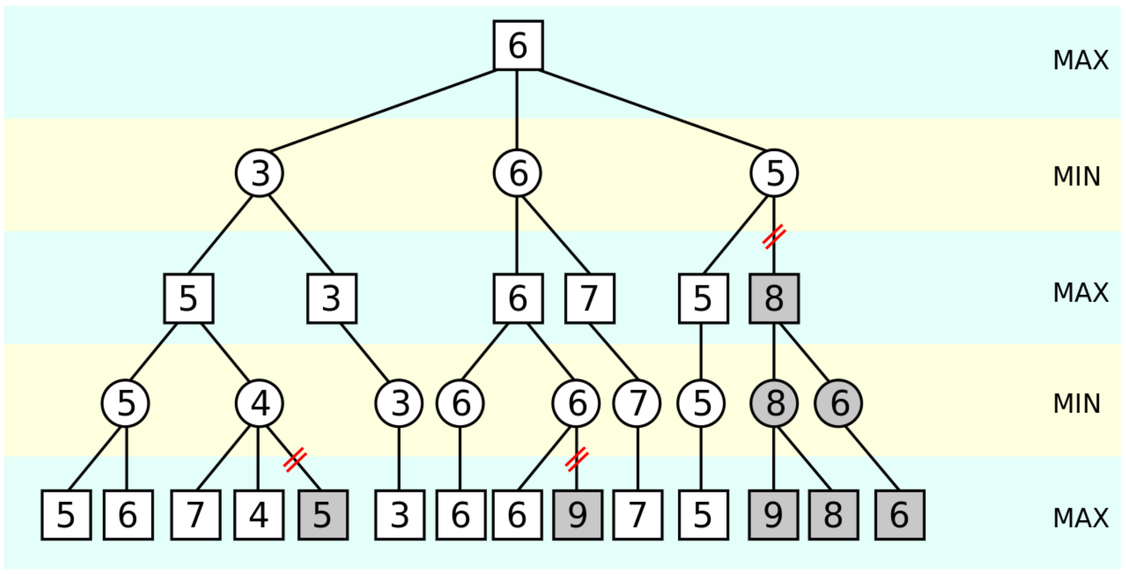


Figure 3 Example of alpha beta pruning.

Image above shows the example of alpha beta pruning working. When moves are evaluated from left to right greyed out subtrees do not need to be searched since its known that group of the sub trees as a whole yields the value of an equivalent subtree or worse, and as such cannot influence the final result. This speeds up the search algorithm considerably.

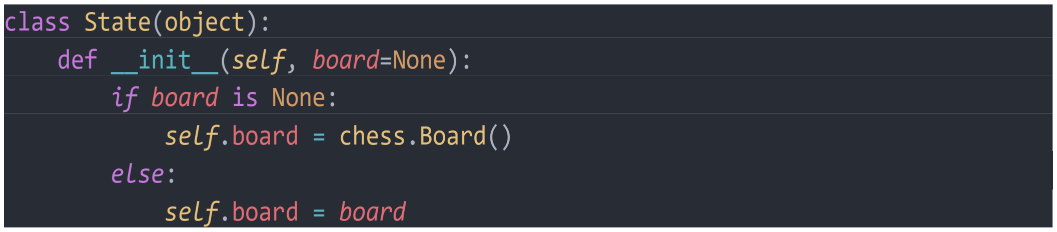
CREATING THE CHESS ENGINE

Project implementation can be split into distinct parts:

1. Creating board state class to serialize any board state and make it parseable by the neural network.
2. Gathering training datasets to train the neural network.
3. Training the neural network.
4. Implementing the evaluator function and search algorithm.
5. Making game playable in browser.

Board state class.

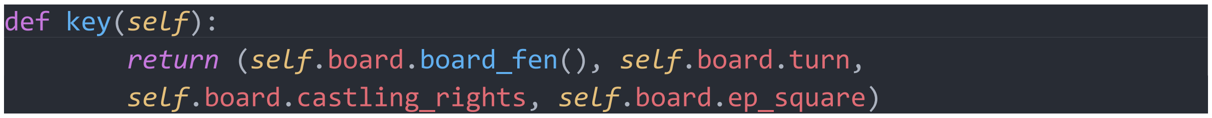
Board state class is implemented in state.py file. Only library used is python-chess. Class has constructor, key function, serialization function, and edges function.



```
class State(object):
    def __init__(self, board=None):
        if board is None:
            self.board = chess.Board()
        else:
            self.board = board
```

Figure 4 State class' constructor.

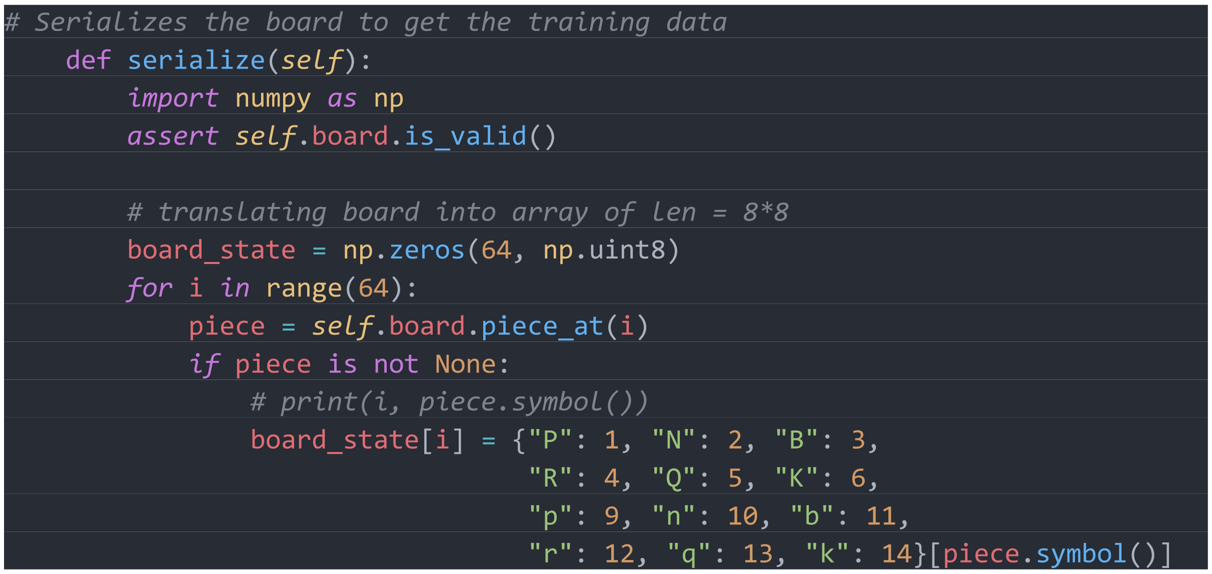
Constructor that takes in board class form python-chess library and created new board if none is passed.



```
def key(self):
        return (self.board.board_fen(), self.board.turn,
        self.board.castling_rights, self.board.ep_square)
```

Figure 5 State class key function.

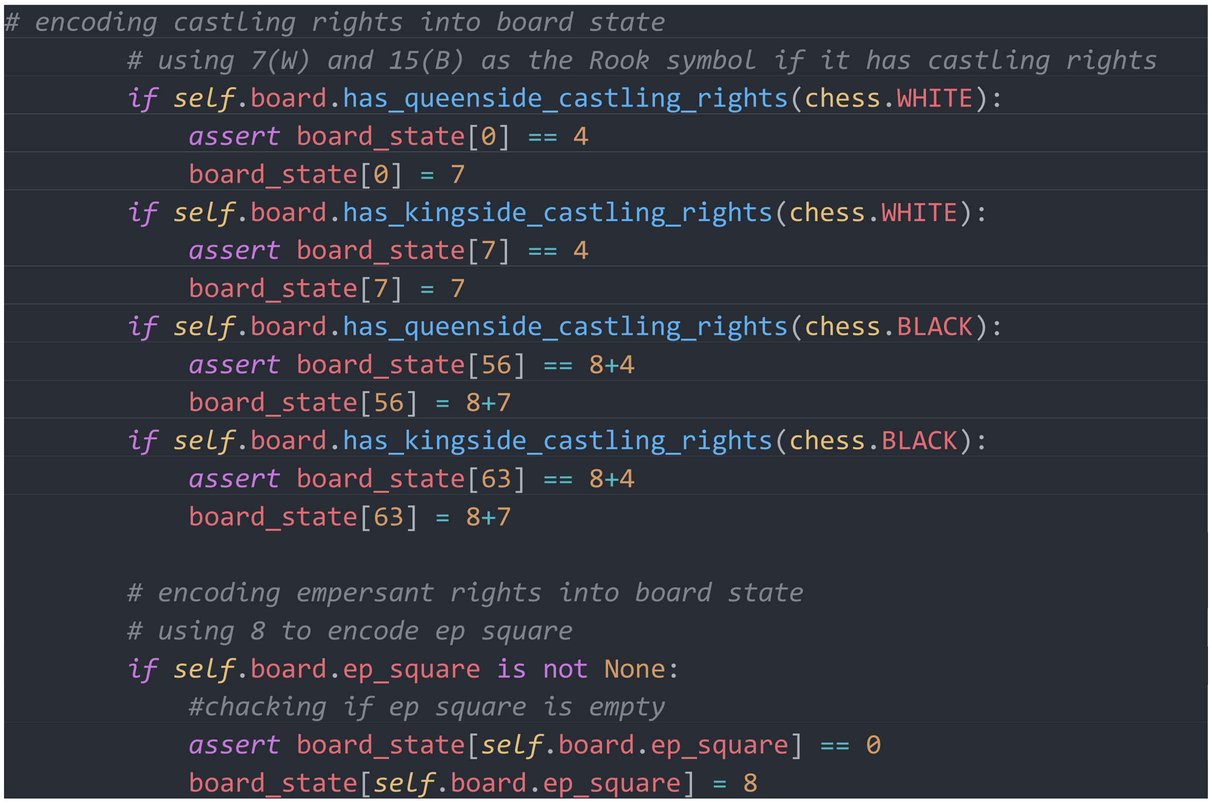
Key function return boards state easily readable by human.



```python
# Serializes the board to get the training data
    def serialize(self):
        import numpy as np
        assert self.board.is_valid()

        # translating board into array of len = 8*8
        board_state = np.zeros(64, np.uint8)
        for i in range(64):
            piece = self.board.piece_at(i)
            if piece is not None:
                # print(i, piece.symbol())
                board_state[i] = {"P": 1, "N": 2, "B": 3,
                                  "R": 4, "Q": 5, "K": 6,
                                  "p": 9, "n": 10, "b": 11,
                                  "r": 12, "q": 13, "k": 14}[piece.symbol()]
```

Figure 6 State class' serialize function 1.

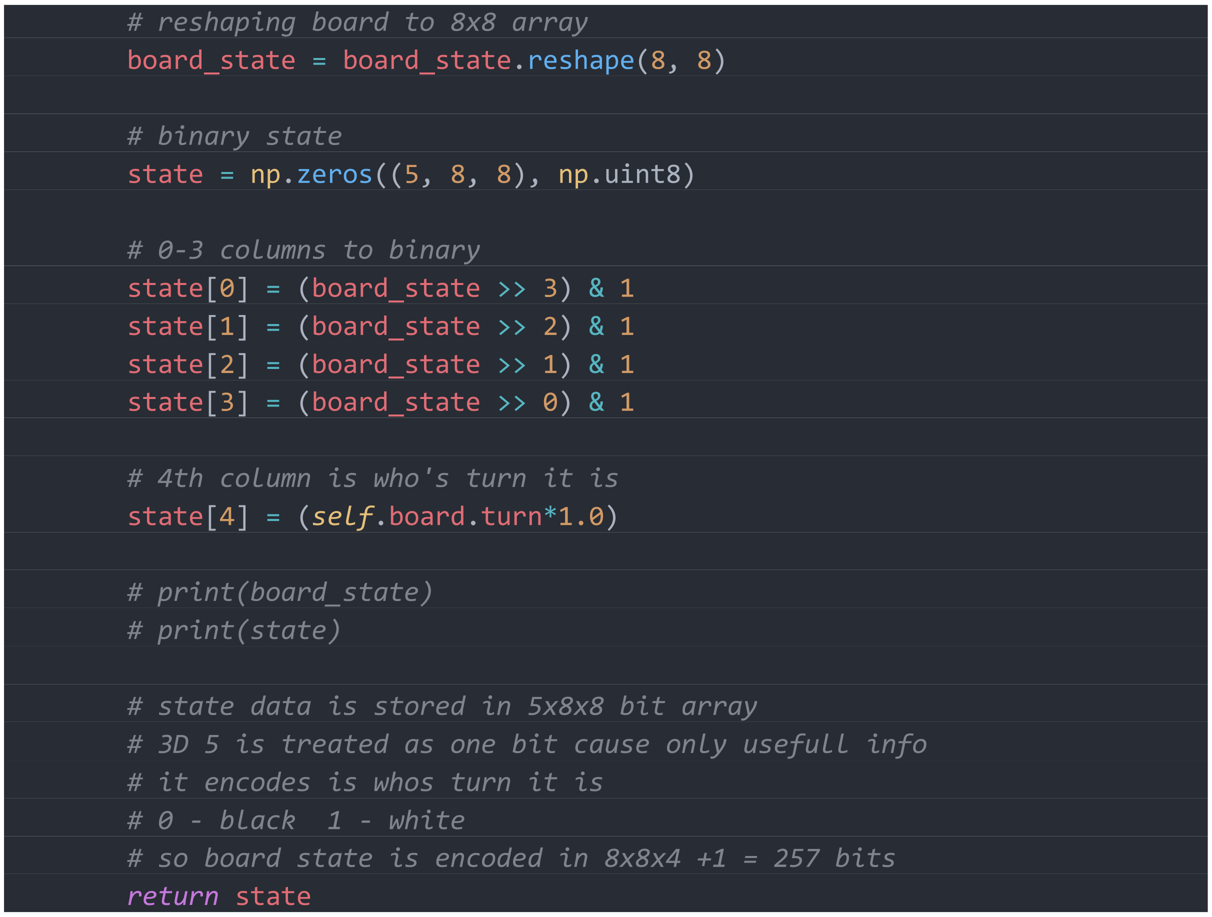
This code translates board into 8x8 array and encodes pieces as numbers in this array.



```python
# encoding castling rights into board state
        # using 7(W) and 15(B) as the Rook symbol if it has castling rights
        if self.board.has_queenside_castling_rights(chess.WHITE):
            assert board_state[0] == 4
            board_state[0] = 7
        if self.board.has_kingside_castling_rights(chess.WHITE):
            assert board_state[7] == 4
            board_state[7] = 7
        if self.board.has_queenside_castling_rights(chess.BLACK):
            assert board_state[56] == 8+4
            board_state[56] = 8+7
        if self.board.has_kingside_castling_rights(chess.BLACK):
            assert board_state[63] == 8+4
            board_state[63] = 8+7

        # encoding empersant rights into board state
        # using 8 to encode ep square
        if self.board.ep_square is not None:
            #chacking if ep square is empty
            assert board_state[self.board.ep_square] == 0
            board_state[self.board.ep_square] = 8
```

Figure 7 State class' serialize function 2.

Encoding castling rights and “En Passant” move into the same 8x8 array.



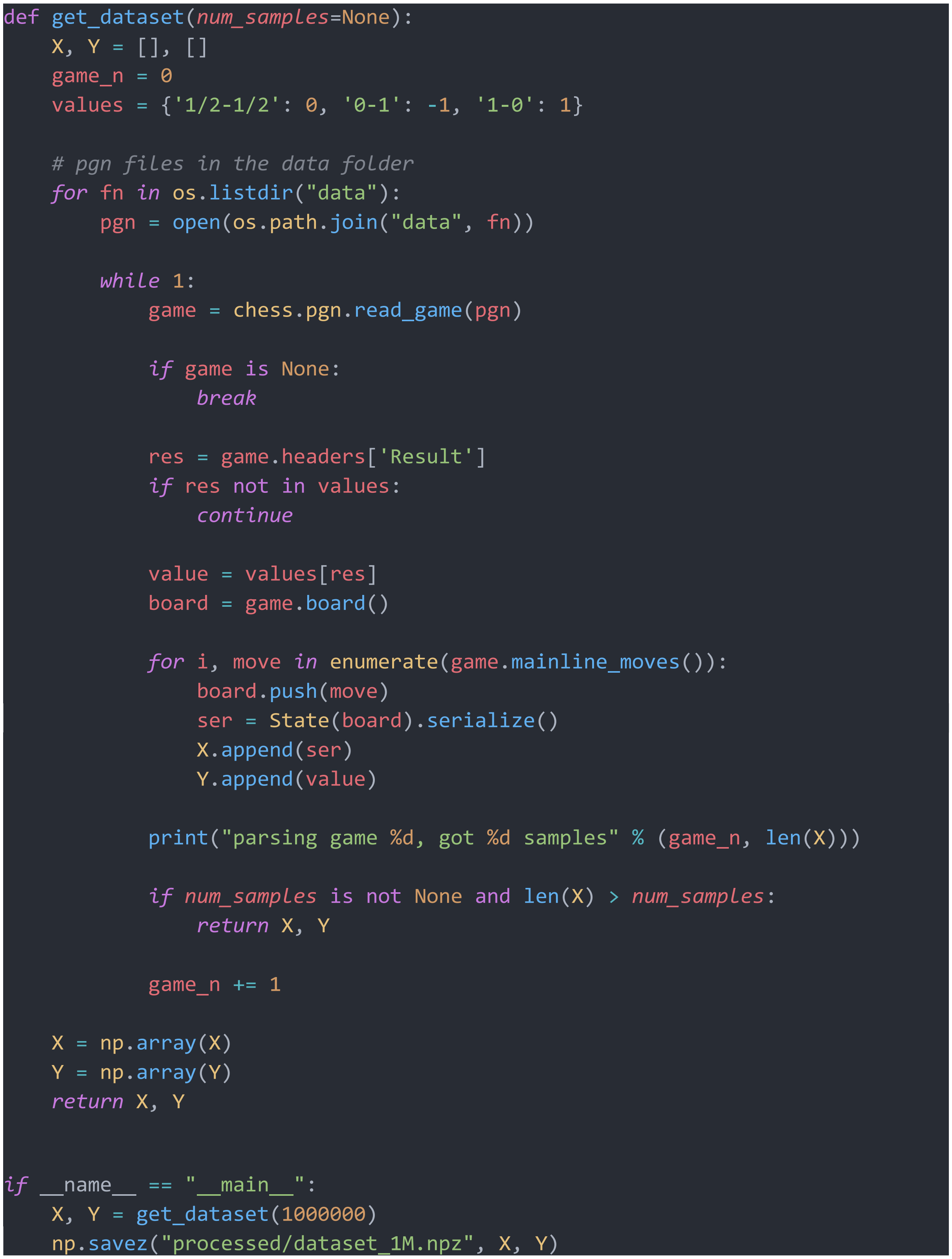
```
        # reshaping board to 8x8 array
        board_state = board_state.reshape(8, 8)

        # binary state
        state = np.zeros((5, 8, 8), np.uint8)

        # 0-3 columns to binary
        state[0] = (board_state >> 3) & 1
        state[1] = (board_state >> 2) & 1
        state[2] = (board_state >> 1) & 1
        state[3] = (board_state >> 0) & 1

        # 4th column is who's turn it is
        state[4] = (self.board.turn*1.0)

        # print(board_state)
        # print(state)

        # state data is stored in 5x8x8 bit array
        # 3D 5 is treated as one bit cause only usefull info
        # it encodes is whos turn it is
        # 0 - black  1 - white
        # so board state is encoded in 8x8x4 +1 = 257 bits
        return state
```

Figure 8 State class' serialize function 3.

Converting 8x8 array to 5x8x8 array to represent decimal piece numbers in binary.

Gathering the training dataset.

To create training dataset for the neural network state class is used. Libraries used: Python-chess and NumPy.

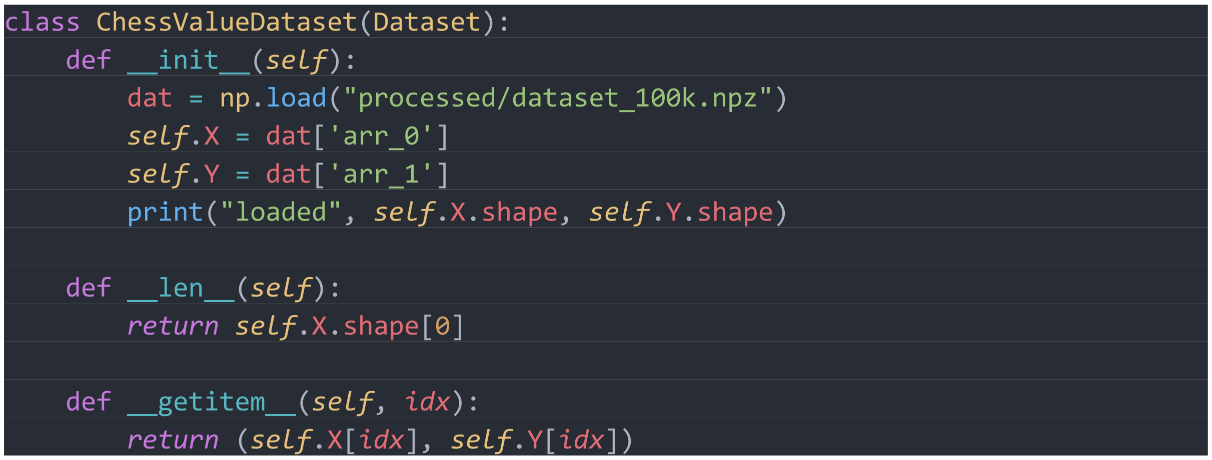


```python
def get_dataset(num_samples=None):
    X, Y = [], []
    game_n = 0
    values = {'1/2-1/2': 0, '0-1': -1, '1-0': 1}

    # pgn files in the data folder
    for fn in os.listdir("data"):
        pgn = open(os.path.join("data", fn))

        while 1:
            game = chess.pgn.read_game(pgn)

            if game is None:
                break

            res = game.headers['Result']
            if res not in values:
                continue

            value = values[res]
            board = game.board()

            for i, move in enumerate(game.mainline_moves()):
                board.push(move)
                ser = State(board).serialize()
                X.append(ser)
                Y.append(value)

            print("parsing game %d, got %d samples" % (game_n, len(X)))

            if num_samples is not None and len(X) > num_samples:
                return X, Y

            game_n += 1

    X = np.array(X)
    Y = np.array(Y)
    return X, Y


if __name__ == "__main__":
    X, Y = get_dataset(1000000)
    np.savez("processed/dataset_1M.npz", X, Y)
```

Code above parses game data from the site Lichess.org iterating each games every move end recording board state in any during move to array X and game result to array Y. 1000000 board states were gathered to train the neural network and saved to “dataset_1M.npz” file.

Training the neural network.

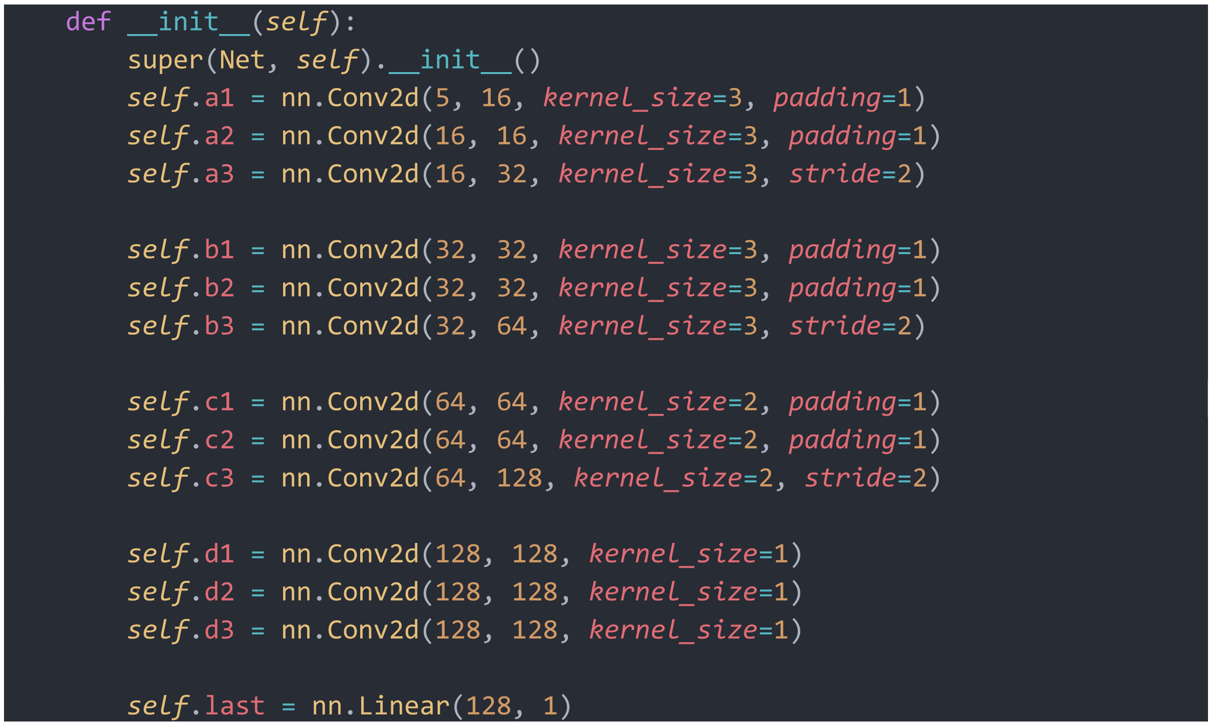
Neural network class is defined in “train.py” file. Libraries used: PyTorch and NumPy.



```
class ChessValueDataset(Dataset):
    def __init__(self):
        dat = np.load("processed/dataset_100k.npz")
        self.X = dat['arr_0']
        self.Y = dat['arr_1']
        print("loaded", self.X.shape, self.Y.shape)

    def __len__(self):
        return self.X.shape[0]

    def __getitem__(self, idx):
        return (self.X[idx], self.Y[idx])
```

Figure 9 Training dataset loading.

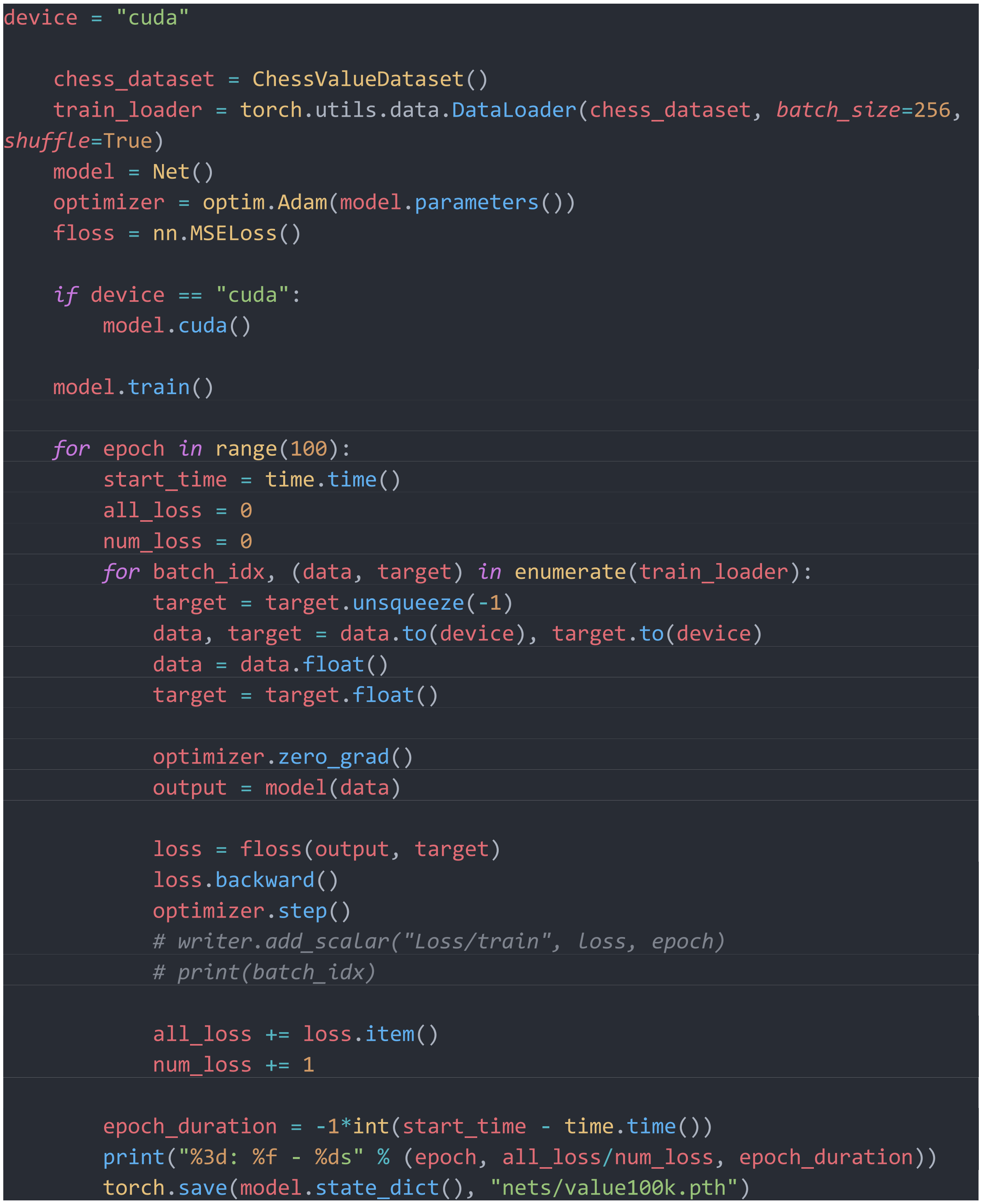
Firstly, we create a class to load training dataset from .npz file into X and Y arrays.



```
    def __init__(self):
        super(Net, self).__init__()
        self.a1 = nn.Conv2d(5, 16, kernel_size=3, padding=1)
        self.a2 = nn.Conv2d(16, 16, kernel_size=3, padding=1)
        self.a3 = nn.Conv2d(16, 32, kernel_size=3, stride=2)

        self.b1 = nn.Conv2d(32, 32, kernel_size=3, padding=1)
        self.b2 = nn.Conv2d(32, 32, kernel_size=3, padding=1)
        self.b3 = nn.Conv2d(32, 64, kernel_size=3, stride=2)

        self.c1 = nn.Conv2d(64, 64, kernel_size=2, padding=1)
        self.c2 = nn.Conv2d(64, 64, kernel_size=2, padding=1)
        self.c3 = nn.Conv2d(64, 128, kernel_size=2, stride=2)

        self.d1 = nn.Conv2d(128, 128, kernel_size=1)
        self.d2 = nn.Conv2d(128, 128, kernel_size=1)
        self.d3 = nn.Conv2d(128, 128, kernel_size=1)

        self.last = nn.Linear(128, 1)
```

Figure 10 Defining the neural network.

Then we create “Net” class create convolutional neural network.



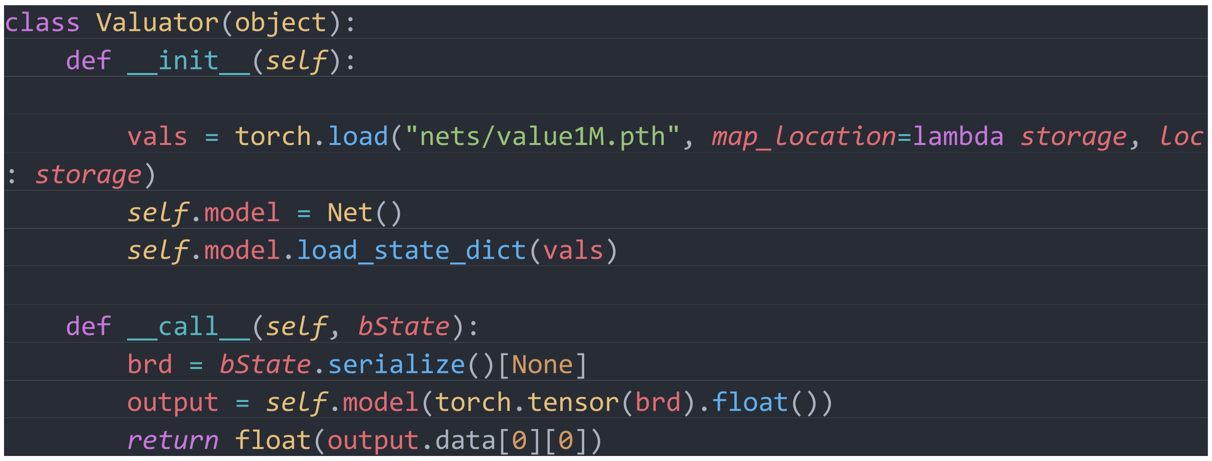
```python
device = "cuda"

    chess_dataset = ChessValueDataset()
    train_loader = torch.utils.data.DataLoader(chess_dataset, batch_size=256,
shuffle=True)
    model = Net()
    optimizer = optim.Adam(model.parameters())
    floss = nn.MSELoss()

    if device == "cuda":
        model.cuda()

    model.train()

    for epoch in range(100):
        start_time = time.time()
        all_loss = 0
        num_loss = 0
        for batch_idx, (data, target) in enumerate(train_loader):
            target = target.unsqueeze(-1)
            data, target = data.to(device), target.to(device)
            data = data.float()
            target = target.float()

            optimizer.zero_grad()
            output = model(data)

            loss = floss(output, target)
            loss.backward()
            optimizer.step()
            # writer.add_scalar("Loss/train", loss, epoch)
            # print(batch_idx)

            all_loss += loss.item()
            num_loss += 1

        epoch_duration = -1*int(start_time - time.time())
        print("%3d: %f - %ds" % (epoch, all_loss/num_loss, epoch_duration))
        torch.save(model.state_dict(), "nets/value100k.pth")
```

Figure 11 Training neural network.

To train PyTorch neural network we need to define a device, in this this case its “cuda” because model was trained on a Nvidia GeForce GTX 1650Ti and took around 16 hours. After 100 epochs neural net model is saved to “nets/value100k.pth” file.

Evaluation function and search algorithm.

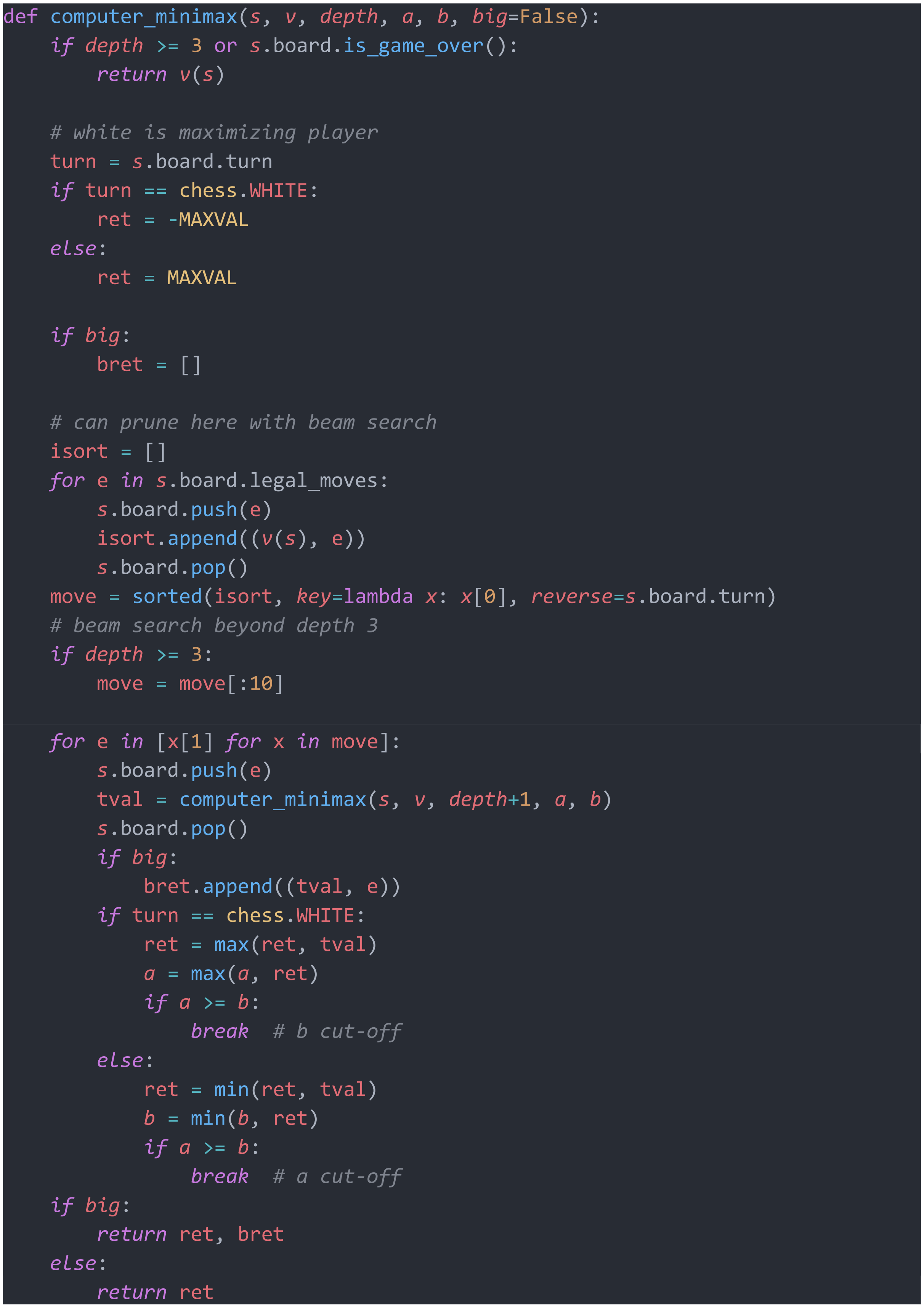
In “play.py” file we created evaluation function and implemented the chess engine as a whole using trained neural network to create evaluation function and defined the flask web application that allowed users to play chess in the browser.



```
class Valuator(object):
    def __init__(self):

        vals = torch.load("nets/value1M.pth", map_location=lambda storage, loc
: storage)
        self.model = Net()
        self.model.load_state_dict(vals)

    def __call__(self, bState):
        brd = bState.serialize()[None]
        output = self.model(torch.tensor(brd).float())
        return float(output.data[0][0])
```

Figure 12 Evaluation function.

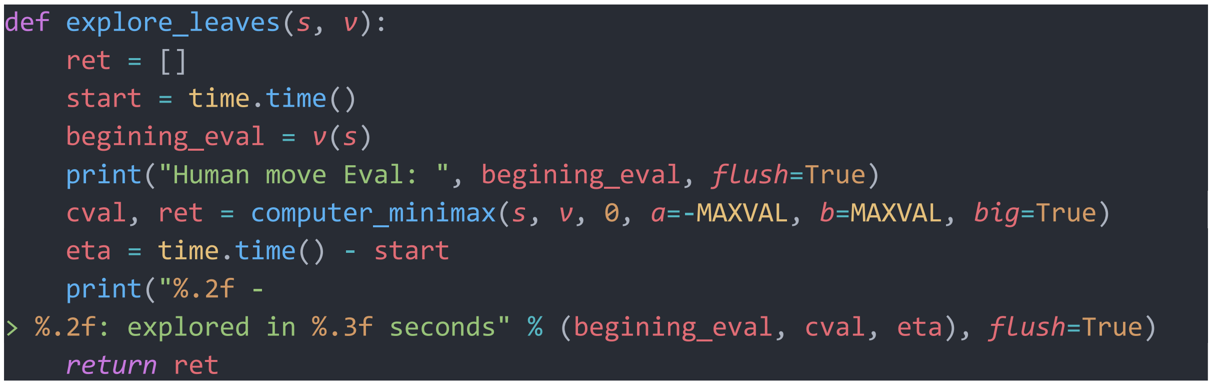
Evaluation function constructor loads trained neural network. When calling evaluation function with serialized board state it returns board evaluation from nn.



```python
def computer_minimax(s, v, depth, a, b, big=False):
    if depth >= 3 or s.board.is_game_over():
        return v(s)

    # white is maximizing player
    turn = s.board.turn
    if turn == chess.WHITE:
        ret = -MAXVAL
    else:
        ret = MAXVAL

    if big:
        bret = []

    # can prune here with beam search
    isort = []
    for e in s.board.legal_moves:
        s.board.push(e)
        isort.append((v(s), e))
        s.board.pop()
    move = sorted(isort, key=lambda x: x[0], reverse=s.board.turn)
    # beam search beyond depth 3
    if depth >= 3:
        move = move[:10]

    for e in [x[1] for x in move]:
        s.board.push(e)
        tval = computer_minimax(s, v, depth+1, a, b)
        s.board.pop()
        if big:
            bret.append((tval, e))
        if turn == chess.WHITE:
            ret = max(ret, tval)
            a = max(a, ret)
            if a >= b:
                break  # b cut-off
        else:
            ret = min(ret, tval)
            b = min(b, ret)
            if a >= b:
                break  # a cut-off
    if big:
        return ret, bret
    else:
        return ret
```

Figure 13 Minmax algorithm.

Then we implement minmax algorithm.



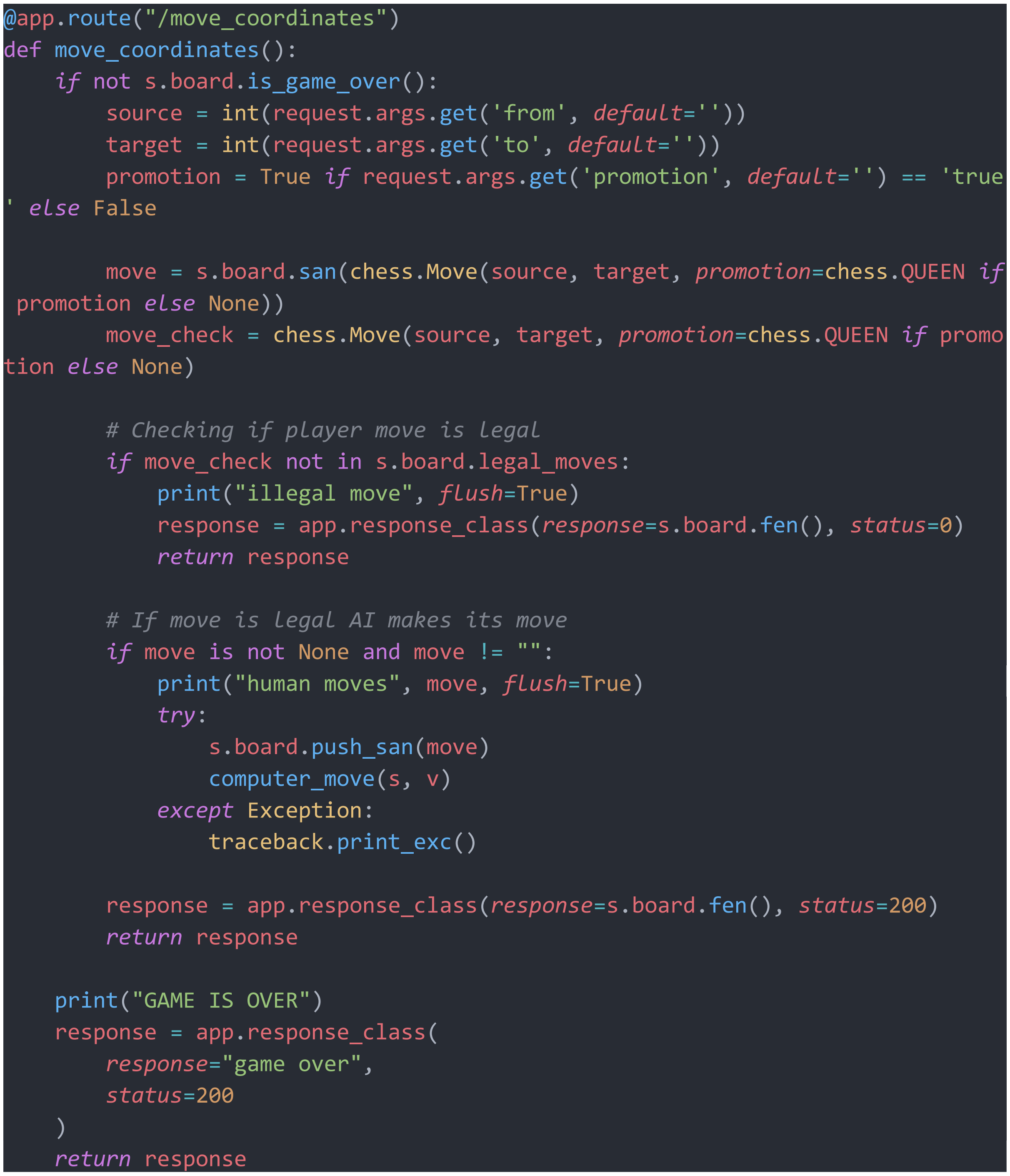
```python
def explore_leaves(s, v):
    ret = []
    start = time.time()
    begining_eval = v(s)
    print("Human move Eval: ", begining_eval, flush=True)
    cval, ret = computer_minimax(s, v, 0, a=-MAXVAL, b=MAXVAL, big=True)
    eta = time.time() - start
    print("%.2f -
> %.2f: explored in %.3f seconds" % (begining_eval, cval, eta), flush=True)
    return ret
```

Figure 14 Search algorithm.

And search algorithm that when called with current board state and valuation function returns best move available.

Creating web application.

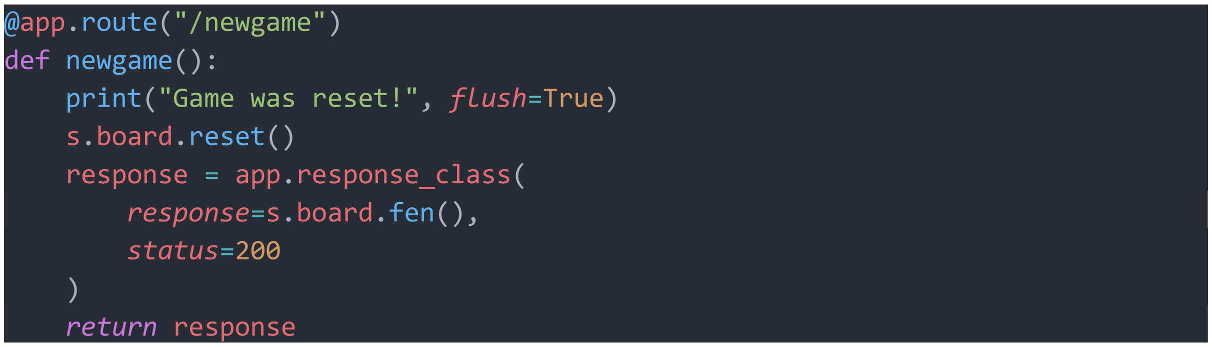
Web application was creates using PyFlask framework. Two API endpoints were made one for resetting the game and second one to send player made moves to the chess engine and return a blacks move. Game board and pieces were implemented in “index.html” file.



```python
@app.route("/move_coordinates")
def move_coordinates():
    if not s.board.is_game_over():
        source = int(request.args.get('from', default=''))
        target = int(request.args.get('to', default=''))
        promotion = True if request.args.get('promotion', default='') == 'true' else False

        move = s.board.san(chess.Move(source, target, promotion=chess.QUEEN if promotion else None))
        move_check = chess.Move(source, target, promotion=chess.QUEEN if promotion else None)

        # Checking if player move is legal
        if move_check not in s.board.legal_moves:
            print("illegal move", flush=True)
            response = app.response_class(response=s.board.fen(), status=0)
            return response

        # If move is legal AI makes its move
        if move is not None and move != "":
            print("human moves", move, flush=True)
            try:
                s.board.push_san(move)
                computer_move(s, v)
            except Exception:
                traceback.print_exc()

        response = app.response_class(response=s.board.fen(), status=200)
        return response

    print("GAME IS OVER")
    response = app.response_class(
        response="game over",
        status=200
    )
    return response
```

Figure 15 Engine access endpoint.

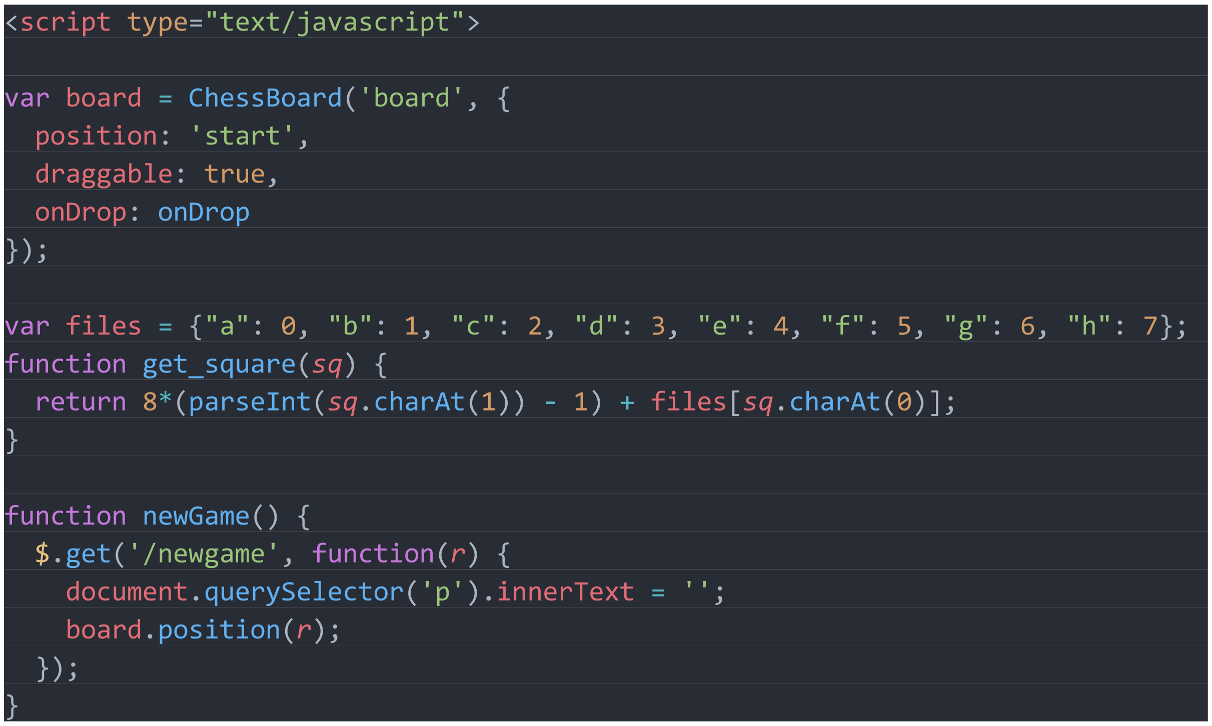
This endpoint upon receiving a played made move checks if game hasn’t ended, promotions and move legality.



```python
@app.route("/newgame")
def newgame():
    print("Game was reset!", flush=True)
    s.board.reset()
    response = app.response_class(
        response=s.board.fen(),
        status=200
    )
    return response
```

Figure 16 New game endpoint.

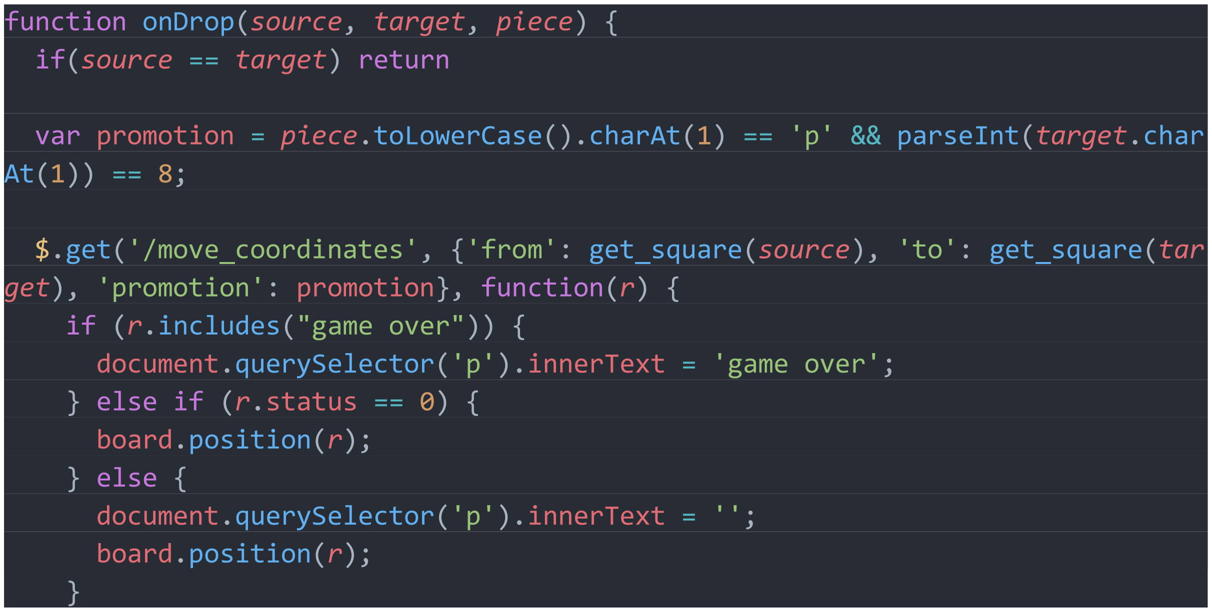
This API endpoint resets the board state.



```javascript
<script type="text/javascript">

var board = ChessBoard('board', {
  position: 'start',
  draggable: true,
  onDrop: onDrop
});

var files = {"a": 0, "b": 1, "c": 2, "d": 3, "e": 4, "f": 5, "g": 6, "h": 7};
function get_square(sq) {
  return 8*(parseInt(sq.charAt(1)) - 1) + files[sq.charAt(0)];
}

function newGame() {
  $.get('/newgame', function(r) {
    document.querySelector('p').innerText = '';
    board.position(r);
  });
}
```

Figure 17 JS to create board in browser.

This JavaScript code defines the chess board and new game logic.

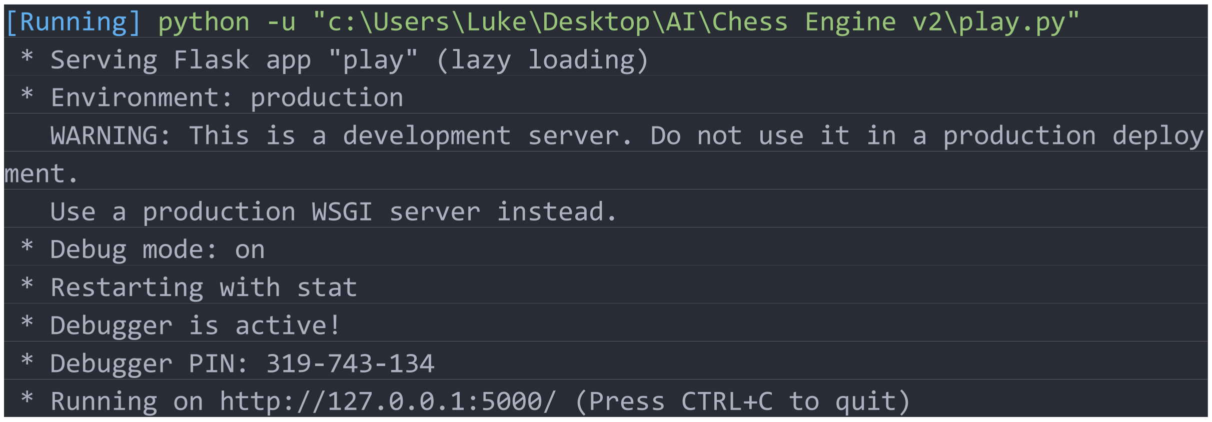


```
function onDrop(source, target, piece) {
  if(source == target) return

  var promotion = piece.toLowerCase().charAt(1) == 'p' && parseInt(target.char
At(1)) == 8;

  $.get('/move_coordinates', {'from': get_square(source), 'to': get_square(tar
get), 'promotion': promotion}, function(r) {
    if (r.includes("game over")) {
      document.querySelector('p').innerText = 'game over';
    } else if (r.status == 0) {
      board.position(r);
    } else {
      document.querySelector('p').innerText = '';
      board.position(r);
    }
```

This JS function is responsible for on drop logic and sends moves to the API endpoint.

HOW TO USE CHESS ENGINE

To start playing chess user needs to run “play.py” file.



```
[Running] python -u "c:\Users\Luke\Desktop\AI\Chess Engine v2\play.py"
 * Serving Flask app "play" (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deploy
ment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 319-743-134
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Figure 18 Output of play.py

By running play.py user launches a web application running on local host. Game can be accessed by going to 127.0.0.1:5000 in the browser.

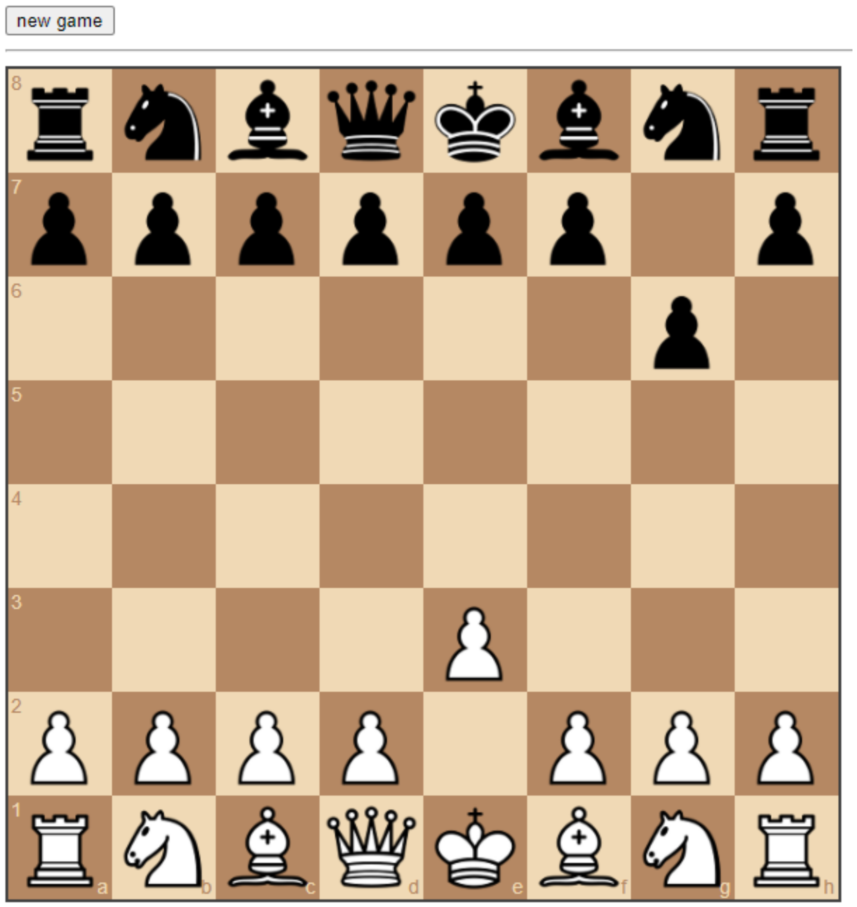


Figure 19 Game in browser tab.