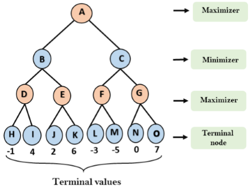
Creating a Chess AI Using CNNs and Stockfish

An Applied Machine Learning Presentation by:

Anirudh Bhatnagar, Asger Gårdsvoll, Atlas Varsted, August Birk, Cooper Nicolaysen, and Petroula Karacosta (All Contributed Equally)

Introduction

- Create an AI that uses board evaluations to play the best move in a given position
- Train a Convolutional Neural Network using board evaluations from the established chess engine Stockfish
- Fit 150000 boards as a 14x8x8 object to a score
- Create a Min-Max algorithm to maximize position for the AI
- Use it to play against Stockfish (0-100 Stockfish wins), against itself (watching children play chess), and a human player

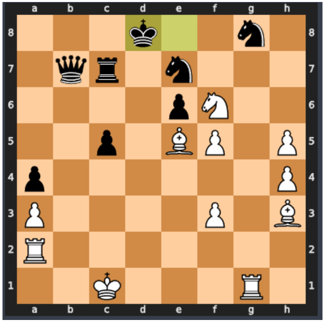


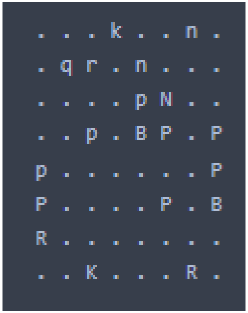
Stockfish

- Stockfish is an open-source chess engine released in 2008
- Uses raw material (piece) advantages in early, mid, and lategame to evaluate position
- Optimal piece placement for knights, bishops, and kings, pawn formation matters.
- All weighted differently over years of fine-tuning
- Finds best move through 30+ deep tree and applies evaluations for each board state.
- Ranked consistently 1st/2nd for best chess engine since 2013, only recently losing to Alphazero by the company DeepMind which uses self-play to train neural networks
- We use this engine to evaluate self created random boards and train the CNN

Setting the Board

We can use a python chess API to create a chess board on a given turn and display as an image and represent in a matrix,



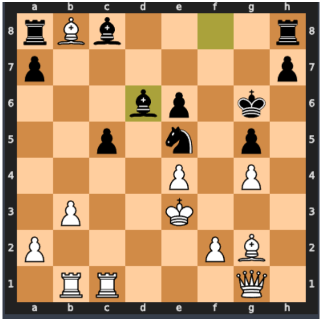


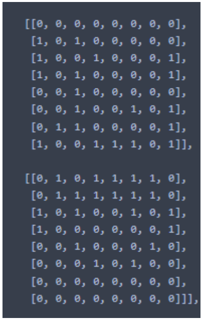
```
. . . k . . n .
. q r . n . . .
. . . . p N . .
. . p . B P . P
p . . . . . . P
P . . . . P . B
R . . . . . . .
. . K . . . R .
```

Figure: Random board on move 80 with a stockfish evaluation of 1684 (White favored), alongside is the game represented as an 8x8 matrix.

Splitting the Board

For each board we split it into 14 dimensions; 1 for each color's pieces (pawns, bishops, knights, rooks, the queen, and the king), and 1 for each sides attacking squares.





```
[[0, 0, 0, 0, 0, 0, 0, 0],
 [1, 0, 1, 0, 0, 0, 0, 0],
 [1, 0, 0, 1, 0, 0, 0, 1],
 [1, 0, 1, 0, 0, 0, 0, 1],
 [0, 0, 1, 0, 0, 0, 0, 0],
 [0, 0, 1, 0, 0, 1, 0, 1],
 [0, 1, 1, 0, 0, 0, 0, 1],
 [1, 0, 0, 1, 1, 1, 0, 1]],

[[0, 1, 0, 1, 1, 1, 1, 0],
 [0, 1, 1, 1, 1, 1, 1, 0],
 [1, 0, 1, 0, 0, 1, 0, 1],
 [1, 0, 0, 0, 0, 0, 0, 1],
 [0, 0, 1, 0, 0, 0, 1, 0],
 [0, 0, 0, 1, 0, 1, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 0]]],
```

Figure: Random board on move 80 with a stockfish evaluation of 1261 (White favored), alongside is where each side has attack pressure.

Data gathering

For the data we generated random chess boards at a random number of turns.

- For the first model the data-set for training was created with 150000 random boards with various depth between turn 1 and turn 200. (random number of boards for each turn)
- For the second model the data-set for training was created with 100000 random boards evenly spaced in depth between turn 1 and turn 20. (5000 boards for each turn)
- For each board we have 1 8x8 matrix which gets split into 1 14x8x8 matrix
- Stockfish evaluations for both as target values

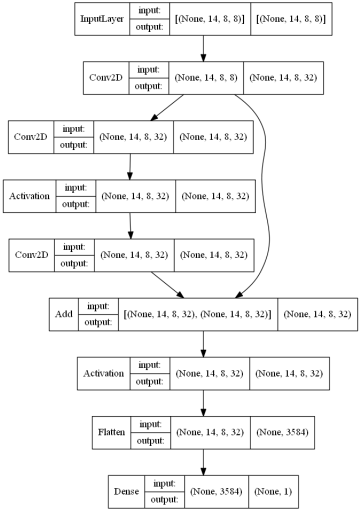
A alternative method is to use data from chess playing websites.

Convolutional Neural Network

We wish to train a CNN that has the ability to read a 3-dimensional board and return an integer value from $-\infty$ to ∞. The larger the number the larger the positional advantages for a given side.

- Keras CNN
- Input layer of 14,8,8 with a convolutional size of 32 and a convolutional depth of 4
- 2D Convolutional layer -> Activation -> 2D Convutional -> Relu Activation -> Flatten -> 1 Dense layer
- Evaluation of boardstate

Convolutional Model



Compiling the models

We create a model for each of the datasets, which will be compared later.

- Batch size $= 2048$
- Epochs $= 1000$
- Loss Equation: Mean Squared Error
- Two Callbacks to avoid overfitting: ReduceLROnPlateau and EarlyStopping both with a patience of 10 epochs and minimum delta of 0.
- Adam optimizer: Momentum and and Root Mean Square Propagation

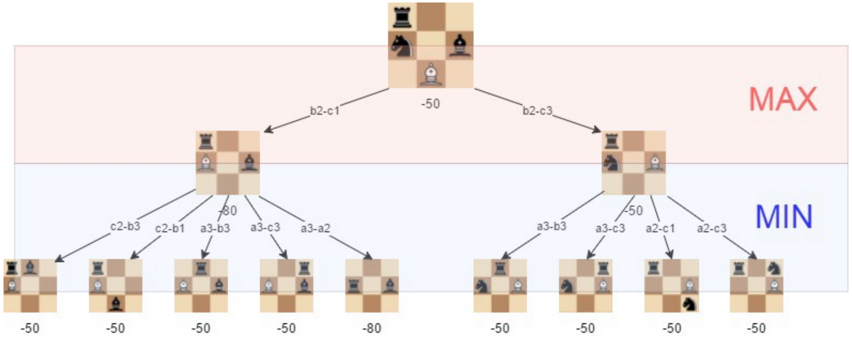
What can we actually do with a model?

Model Uses

Original idea was to see how model could perform at identifying a chess board's position favor for white or black, but we can actually use it to play live chess games.

With a Mini-Max Algorithm we can use the model to evaluate the best moves for a given board

- Use chess API to load stockfish engine to play as the opponent
- Use model to play against itself
- Use human to play against the model

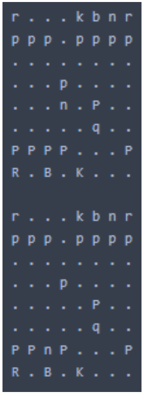


If white is the engine, then we want to find the maximized evaluation for their possible moves but we know the opponent will always minimize white's advantage by playing the best move.

- Layer 0: Current board state to maximize (White's turn)
- Layer 1: All current possible moves for layer 0 to minimize (Black's turn)
- Layer 2: All current possible moves for layer 1 (White's turn again)
- Layer n: All current possible moves for layer n-1 (Black's turn)

Engine vs Stockfish - Model 1

Last 2 moves



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

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

Final position

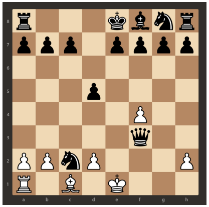
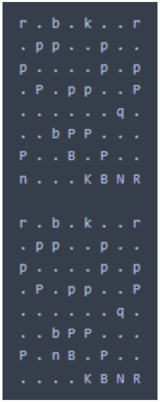


Figure: Checkmated by knight and queen

Example game 1

Last 2 moves



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

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

Final position

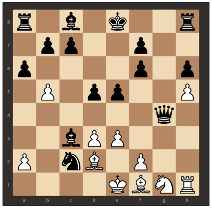
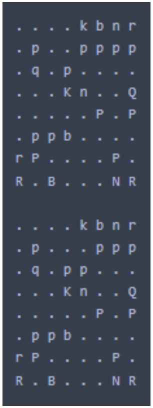


Figure: Checkmated by knight and queen

Example game 2

Engine vs Stockfish - Model 2

Last 2 moves



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

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

Final position

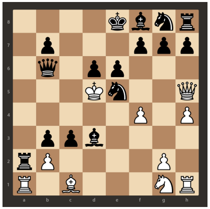
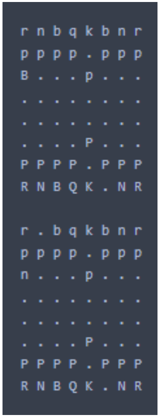


Figure: Checkmated by a pawn in the center of the board

Example game 1

Last 2 moves



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

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

Final position

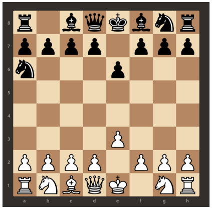
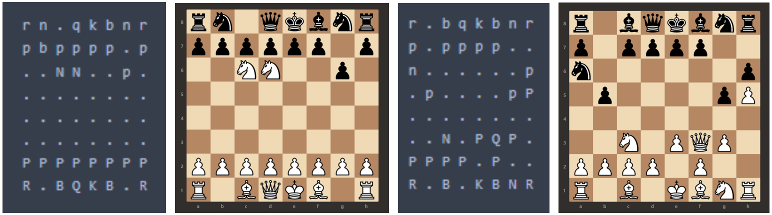


Figure: Sacrificing bishop for no exchange

Example game 2

Engine vs Engine

There was no way our engine could beat Stockfish, so we paired it against itself.



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

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

Figure: Turn 6: Model 1 (left), Model 2 (right)

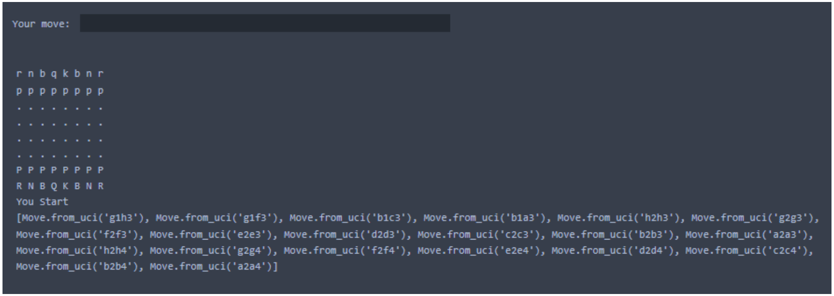
Some of the engine's (many) mistakes:

- No orderly piece development - bad opening principles
- King exposure - no concept of King's safety
- ''Hanging'' pieces

Early game model seems dominant but the structure is still poor.

Engine vs Human

Currently it's possible to play vs the AI, but we have no GUI experience. Input of move notations (no drag and drop) and lack of board coordinates makes it very hard to play a full game.



```
Your move:

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
You Start
[Move.from_uci('g1h3'), Move.from_uci('g1f3'), Move.from_uci('b1c3'), Move.from_uci('b1a3'), Move.from_uci('h2h3'), Move.from_uci('g2g3'),
Move.from_uci('f2f3'), Move.from_uci('e2e3'), Move.from_uci('d2d3'), Move.from_uci('c2c3'), Move.from_uci('b2b3'), Move.from_uci('a2a3'),
Move.from_uci('h2h4'), Move.from_uci('g2g4'), Move.from_uci('f2f4'), Move.from_uci('e2e4'), Move.from_uci('d2d4'), Move.from_uci('c2c4'),
Move.from_uci('b2b4'), Move.from_uci('a2a4')]
```

More time would have been useful to explore this further.

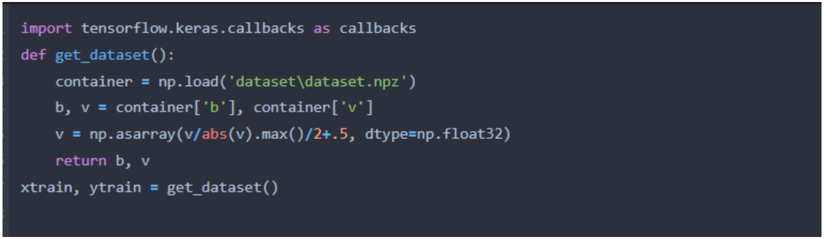
Limitations and Ways to Improve

- Lack of data; Feed full chess games instead of boards
- Never castling/King safety; Indicate between early game and late game where king positioning matters
- Moving too many pieces at in a row; Better move evaluation and deciding what piece is actually in danger
- Too many moves to evaluate; Get better resources and computer
- Not enough time to test more than 2 models due to computing time; a team size of google might help

Conclusion

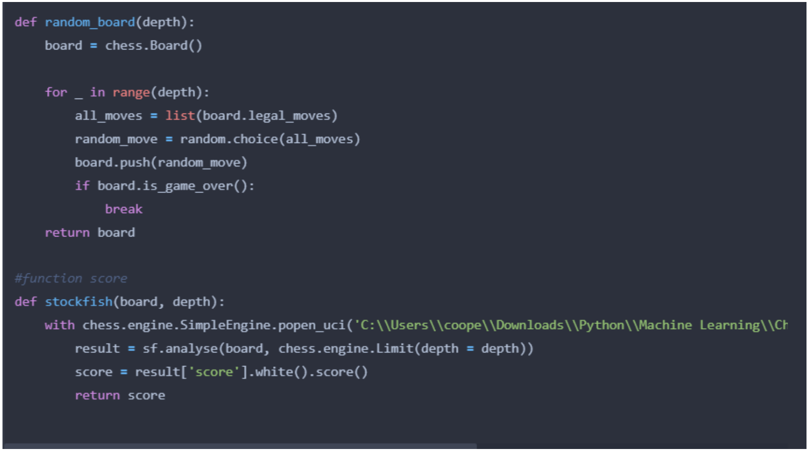
- After attempting two models, we see that the CNN is a very poor choice in evaluating a chess board and picking a move
- Due to a lack of data and computation power, our AI has no ways to evaluate a position other than random boards likely never played before
- Without a way to give incentives to priority move capture (queen $>$ pawn), castling (not seen once), and awareness of attacked pieces, then the model will always perform poor.
- In the future, need more full length games with constant board evaluation. Need to learn from played games

APPENDIX



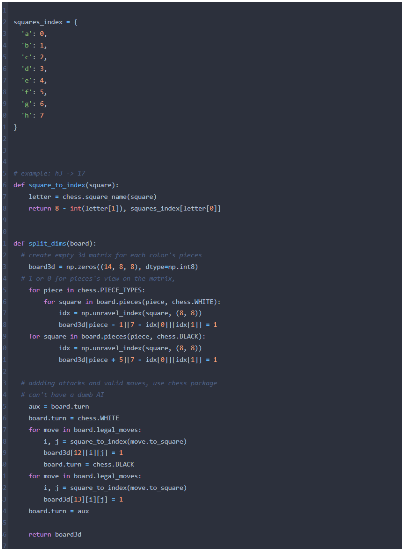
```python
import tensorflow.keras.callbacks as callbacks
def get_dataset():
    container = np.load('dataset\dataset.npz')
    b, v = container['b'], container['v']
    v = np.asarray(v/abs(v).max()/2+.5, dtype=np.float32)
    return b, v
xtrain, ytrain = get_dataset()
```

Figure: Retrieving data after it's made



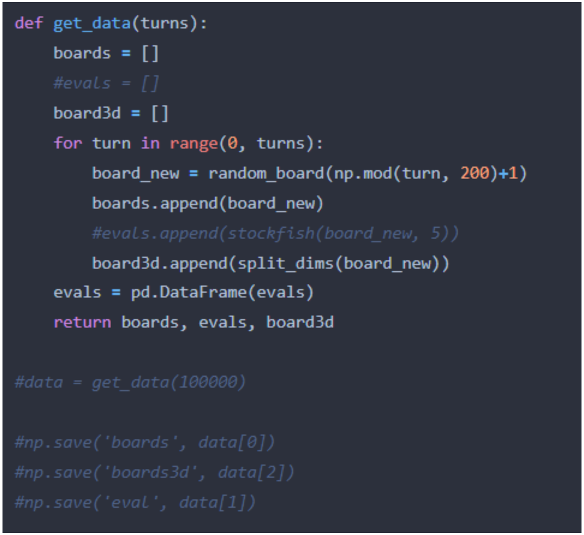
```
def random_board(depth):
    board = chess.Board()

    for _ in range(depth):
        all_moves = list(board.legal_moves)
        random_move = random.choice(all_moves)
        board.push(random_move)
        if board.is_game_over():
            break
    return board

#function score
def stockfish(board, depth):
    with chess.engine.SimpleEngine.popen_uci('C:\\Users\\coope\\Downloads\\Python\\Machine Learning\\Ch
        result = sf.analyse(board, chess.engine.Limit(depth = depth))
        score = result['score'].white().score()
        return score
```

Figure: Random board algorithm and stockfish loader



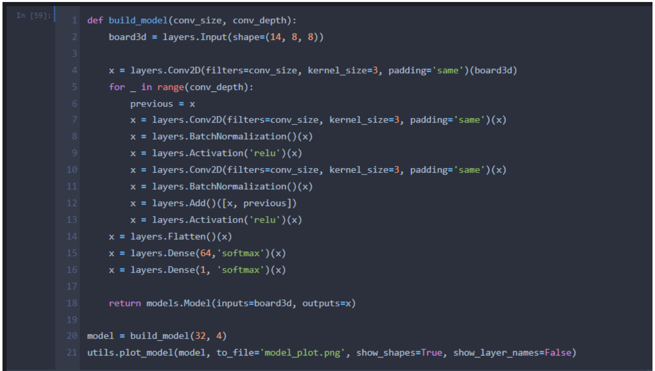
```
squares_index = {
  'a': 0,
  'b': 1,
  'c': 2,
  'd': 3,
  'e': 4,
  'f': 5,
  'g': 6,
  'h': 7
}



# example: h3 -> 17
def square_to_index(square):
    letter = chess.square_name(square)
    return 8 - int(letter[1]), squares_index[letter[0]]


def split_dims(board):
  # create empty 3d matrix for each color's pieces
    board3d = np.zeros((14, 8, 8), dtype=np.int8)
  # 1 or 0 for pieces's view on the matrix,
    for piece in chess.PIECE_TYPES:
        for square in board.pieces(piece, chess.WHITE):
            idx = np.unravel_index(square, (8, 8))
            board3d[piece - 1][7 - idx[0]][idx[1]] = 1
    for square in board.pieces(piece, chess.BLACK):
            idx = np.unravel_index(square, (8, 8))
            board3d[piece + 5][7 - idx[0]][idx[1]] = 1

  # addding attacks and valid moves, use chess package
  # can't have a dumb AI
    aux = board.turn
    board.turn = chess.WHITE
    for move in board.legal_moves:
        i, j = square_to_index(move.to_square)
        board3d[12][i][j] = 1
        board.turn = chess.BLACK
    for move in board.legal_moves:
        i, j = square_to_index(move.to_square)
        board3d[13][i][j] = 1
    board.turn = aux

    return board3d
```

Figure: Caption

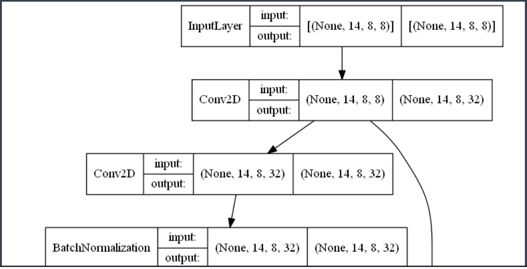


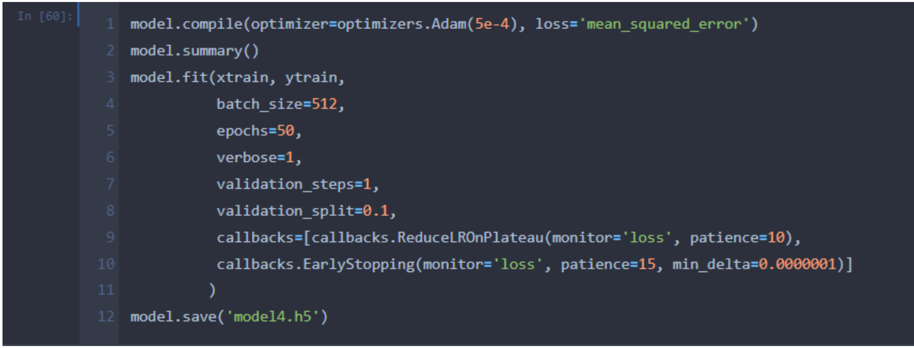
```python
def get_data(turns):
    boards = []
    #evals = []
    board3d = []
    for turn in range(0, turns):
        board_new = random_board(np.mod(turn, 200)+1)
        boards.append(board_new)
        #evals.append(stockfish(board_new, 5))
        board3d.append(split_dims(board_new))
    evals = pd.DataFrame(evals)
    return boards, evals, board3d

#data = get_data(100000)

#np.save('boards', data[0])
#np.save('boards3d', data[2])
#np.save('eval', data[1])
```

Figure: Data maker



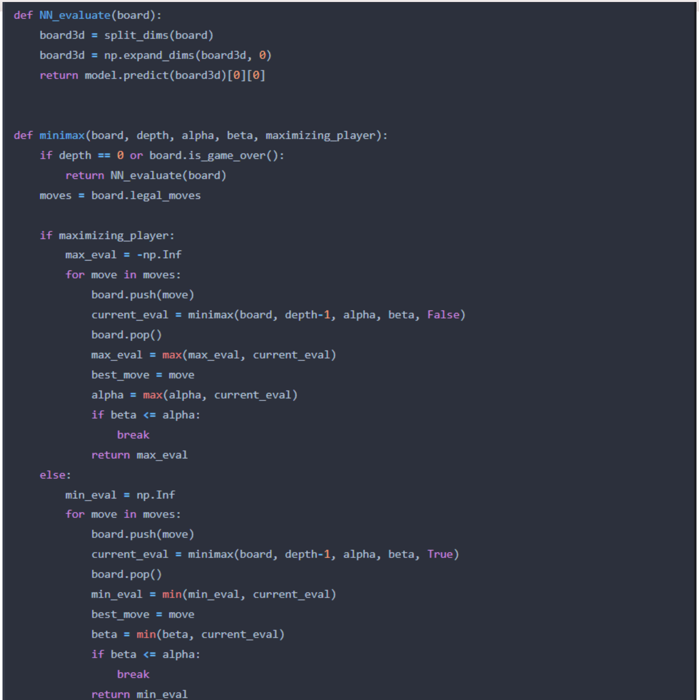
```
In [59]:
def build_model(conv_size, conv_depth):
    board3d = layers.Input(shape=(14, 8, 8))

    x = layers.Conv2D(filters=conv_size, kernel_size=3, padding='same')(board3d)
    for _ in range(conv_depth):
        previous = x
        x = layers.Conv2D(filters=conv_size, kernel_size=3, padding='same')(x)
        x = layers.BatchNormalization()(x)
        x = layers.Activation('relu')(x)
        x = layers.Conv2D(filters=conv_size, kernel_size=3, padding='same')(x)
        x = layers.BatchNormalization()(x)
        x = layers.Add()([x, previous])
        x = layers.Activation('relu')(x)
    x = layers.Flatten()(x)
    x = layers.Dense(64,'softmax')(x)
    x = layers.Dense(1, 'softmax')(x)

    return models.Model(inputs=board3d, outputs=x)

model = build_model(32, 4)
utils.plot_model(model, to_file='model_plot.png', show_shapes=True, show_layer_names=False)
```



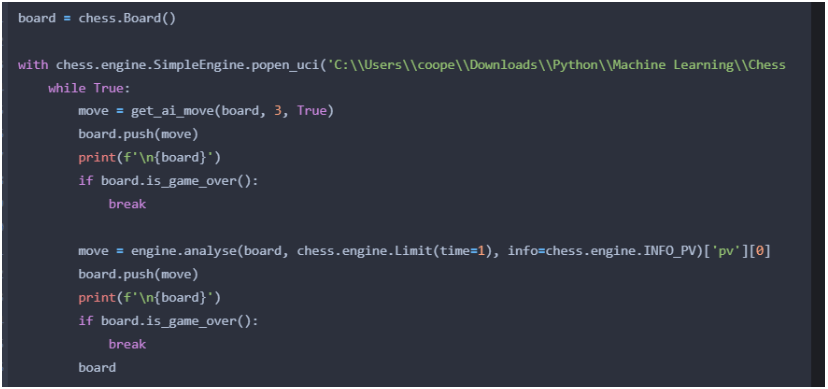


```
In [60]:
model.compile(optimizer=optimizers.Adam(5e-4), loss='mean_squared_error')
model.summary()
model.fit(xtrain, ytrain,
          batch_size=512,
          epochs=50,
          verbose=1,
          validation_steps=1,
          validation_split=0.1,
          callbacks=[callbacks.ReduceLROnPlateau(monitor='loss', patience=10),
          callbacks.EarlyStopping(monitor='loss', patience=15, min_delta=0.0000001)]
         )
model.save('model4.h5')
```

Figure: Model fitting

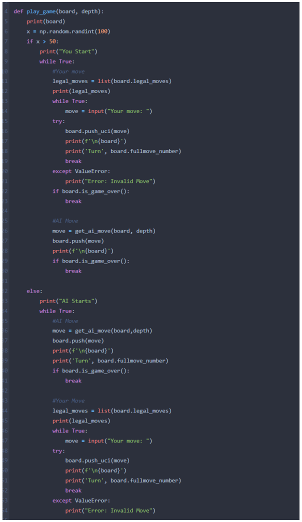


```
def NN_evaluate(board):
    board3d = split_dims(board)
    board3d = np.expand_dims(board3d, 0)
    return model.predict(board3d)[0][0]


def minimax(board, depth, alpha, beta, maximizing_player):
    if depth == 0 or board.is_game_over():
        return NN_evaluate(board)
    moves = board.legal_moves

    if maximizing_player:
        max_eval = -np.Inf
        for move in moves:
            board.push(move)
            current_eval = minimax(board, depth-1, alpha, beta, False)
            board.pop()
            max_eval = max(max_eval, current_eval)
            best_move = move
            alpha = max(alpha, current_eval)
            if beta <= alpha:
                break
            return max_eval
    else:
        min_eval = np.Inf
        for move in moves:
            board.push(move)
            current_eval = minimax(board, depth-1, alpha, beta, True)
            board.pop()
            min_eval = min(min_eval, current_eval)
            best_move = move
            beta = min(beta, current_eval)
            if beta <= alpha:
                break
            return min_eval
```



```
board = chess.Board()

with chess.engine.SimpleEngine.popen_uci('C:\\Users\\coope\\Downloads\\Python\\Machine Learning\\Chess
    while True:
        move = get_ai_move(board, 3, True)
        board.push(move)
        print(f'\n{board}')
        if board.is_game_over():
            break

        move = engine.analyse(board, chess.engine.Limit(time=1), info=chess.engine.INFO_PV)['pv'][0]
        board.push(move)
        print(f'\n{board}')
        if board.is_game_over():
            break
        board
```

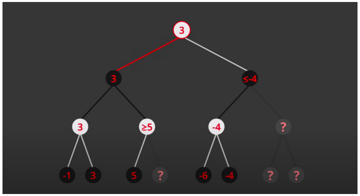
Figure: How to play against stockfish



```python
def play_game(board, depth):
    print(board)
    x = np.random.randint(100)
    if x > 50:
        print("You Start")
        while True:
            #Your move
            legal_moves = list(board.legal_moves)
            print(legal_moves)
            while True:
                move = input("Your move: ")
            try:
                board.push_uci(move)
                print(f'\n{board}')
                print('Turn', board.fullmove_number)
                break
            except ValueError:
                print("Error: Invalid Move")
            if board.is_game_over():
                break

            #AI Move
            move = get_ai_move(board, depth)
            board.push(move)
            print(f'\n{board}')
            if board.is_game_over():
                break

    else:
        print("AI Starts")
        while True:
            #AI Move
            move = get_ai_move(board,depth)
            board.push(move)
            print(f'\n{board}')
            print('Turn', board.fullmove_number)
            if board.is_game_over():
                break

            #Your Move
            legal_moves = list(board.legal_moves)
            print(legal_moves)
            while True:
                move = input("Your move: ")
            try:
                board.push_uci(move)
                print(f'\n{board}')
                print('Turn', board.fullmove_number)
                break
            except ValueError:
                print("Error: Invalid Move")
```

Figure: Really weak chess game code

Extra Slide



- Alpha-beta pruning is a way to limit computation time
- White will never choose a move that gives black a better evaluation than a previously calculated move, so the rest of the right side branches are snipped after the ≤ -4 is calculated. White will always choose left branch.
- On the left side, we know to cut the rest of the ≥ 5 node as black would never choose a move that gives white a larger evaluation than the 3 previously calculated. Black will always choose the left branch.