An Exercise in Emulating Human Chess Intelligence

Thomas Newbold

February 2023

Abstract

This project is independent research related to my ECM2423 Artificial Intelligence and Applications module, intended to supplement my own understanding of machine learning. A link to the GitHub repositry for this project: `https://github.com/tom-newbold/tensorflow-chess`.

1 Rationale for this project

As many have in recent months, I have developed an obsession for chess. Typical chess bots are a classic software-developer portfolio project, which have been mathematically optimised and implemented so efficiently by others to the point of redundancy, so was looking for a way to put my own spin on the norm.

Upon discovering I could download my an archive of my games, I saw a perfect opportunity to delve into the world of machine learning (a field I have been avoiding for some time owing to its complexity) - not to create the perfect bot, but a notably human one.

Machine learning has its foundations in pattern recognition - a key skill in "chess intelligence" - which most other AI models struggle to replicate. I will attempt to extract useful "state-decision" data from my own chess game archive, and to develop a machine learning model using this to emulate my ability.

2 Data Acquisition

2.1 Coupled Stockfish Engine

The initial step in the data acquisition process was to develop a class which could: store board states, check move validity, and evaluate positions. For this, I used the python-chess module (`https://python-chess.readthedocs.io/en/latest/`), which provides in-built functionality for reading PGN data, and relevant move validity functions. I accompanied this with an Stockfish instance (using the Stockfish module: `https://pypi.org/project/stockfish/`) which allows for position evaluation from its internal board data structure.

While this second part is not essential for the main loss function I intend to use, it will be required for a certain variation I want to test, so implementing it at this point seems most convenient.

2.2 Extracting Data Points

Using this class, I can take a game PGN (**GAME_PATH** in the code below), run through it and save each state on the player's turn. Figure 1 shows an example data point.

```
import coupled_stockfish_engine as cse
sf = cse.CoupledStockfish(GAME_PATH, 'The111thTom')

states, b = sf.run()
p = sf.player
out_json = {'data_points':[]}
for s in states:
  _state = {
    'player': p,
    'fen': s['fen'],
    'following_move': s['move'],
    'resulting_eval': s['eval']
  }
  out_json['data_points'].append(_state)

import json
with open('bin\\out.json', 'w') as file:
  json.dump(out_json, file)
```


```
"data_points": [
    {
        "player": 0,
        "fen": "rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1",
        "following_move": "c4",
        "resulting_eval": {
            "type": "cp",
            "value": -16
        }
    },
```


Figure 1: Data Point JSON format

2.3 API calls

Online chess services such as chess.com and lichess.org offer API which allow queries to fetch game archives for any player. I was able to iterate over PGNs obtained as such, and run them through the aforementioned class to extract state data.

From my own chess.com archive of more than 250 games I was able to extract about 7000 data-points.

2.4 Move trees

I also wanted a data set which grouped together all moves I have made from any board state I have encountered. This will be useful in checking the model's outputs from a given position.

3 TensorFlow

My idea for the design of this model is one which would take in a board state, and return a move. For this to work, I needed to find a way of representing these numerically.

3.1 Inputs

Initially, I hoped to use SAN strings to represent the game state, but struggled to formulate a way to standardise strings of different lengths for use as inputs. Instead, I have opted to use FEN strings, which I will convert to an 8 by 8 state tensor containing a variation on the classic piece value system to numerically represent pieces (using positive and negative values to distinguish between white and black pieces). *The code below shows a function to do this:*

```
def fen_to_tensor(fen_string):
  map = { 'p':1, 'n':2, 'b':3, 'r':4, 'q':5, 'k':6 }
  array = np.zeros((8,8))
  rows = fen_string.split(' ')[0].split('/')
  for y in range(8):
    for x in range(8):
      piece = rows[y][x]
      try:
        val = int(piece)
        if val!=0:
          rows[y] = rows[y][:x] + val*'0' + rows[y][x+1:]
        for i in range(val):
          array[x+i, y] = 0
        x += val
      except ValueError:
        val = map[piece.lower()]
        array[x, y] = val if piece.isupper() else -val
  output = tf.convert_to_tensor(array.transpose())
  return output
```

3.2 Input Weighting

For the main version of this model, I will not be classifying the data by outcome (win, loss, stalemate, etc.) as I simply want it to play as I play. I will, however, classify older games to be less important - it will be interesting to see if this can achieve a similar learning curve to classic human decision making. The

```
[[-4.  0. -3. -5.  0.  0. -2. -4.]
 [ 0.  0. -1. -6.  0.  3. -1. -1.]
 [-1.  0.  0.  0.  0. -1.  0.  0.]
 [-2. -1.  0.  0. -1.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.  1.  0.  0.]
 [ 1.  5.  1.  0.  1.  0.  0.  1.]
 [ 0.  0.  0.  0.  0.  0.  1.  0.]
 [ 4.  0.  3.  0.  6.  0.  2.  4.]]
```

Figure 2: Example Output Tensor

first rudimentary iteration of the model will use a learning rate which is some function of this property. A later iteration might use a weighting based on the favourability of the outcome, which should allow it to "learn from my mistakes" in a sense.

Note that duplicate cases should not be ignored, as this will strengthen/reinforce moves that I play more commonly (namely openings). I cannot (linearly)[1] combine these cases, as this does not uniquely determine the individual moves, and will lead to ambiguous outputs.

3.3 Outputs

To represent a move as an output, I will use an 8 by 8 by 2 tensor, containing values between 1 and 0. One 8 by 8 will pertain to the source square, and the other the destination, allowing me to interpret in LAN. *The code below shows a pair of functions to convert between LAN notation and a movement tensor:*

```
def lan_to_tensor(lan_string):
  a = np.zeros((2, 8, 8))
  s = [lan_string[:2], lan_string[2:]]
  for i in range(2):
    a[i, 8-int(s[i][1]), 'abcdefgh'.index(s[i][0])] = 1
  return tf.convert_to_tensor(a)

def tensor_to_lan(tensor):
  out = ''
  t = tensor.numpy()
  for d in range(2):
    for y in range(8):
      for x in range(8):
        if t[d][y][x] == 1:
          out += 'abcdefgh'[x] + str(8-y)
  return out
```

[1] Some combination scaled by prime factors might work, but I am eager to avoid this, as it would likely significantly skew the training.

3.4 Loss Function

This model will use the composition of loss functions, to encapsulate the different aspects I want to optimise.

3.4.1 Matrix Sum

In order for the output to approximate a move with confidence, my first loss function will be a variation on the classic "Mean Squared Error" technique, with a similar metric.

In my case, I will be using `tf.math.reduce_sum()` to extract a scalar loss value from the element-wise difference between target and output tensor. The error will be calculated from the product of the sum of the differences, and the difference of the sums (squared), as follows:

$$E(T, O) = E_{\Sigma\Delta}(T, O)E_{\Delta\Sigma}(T, O) \tag{1}$$

where

$$E_{\Sigma\Delta}(T, O) := \sum_{k=0}^{1} \sum_{i,j=0}^{7} (T_{ijk} - O_{ijk})^2 \tag{2}$$

and

$$E_{\Delta\Sigma}(T, O) := (\sum_{k=0}^{1} \sum_{i,j=0}^{7} T_{ijk})^2 - (\sum_{k=0}^{1} \sum_{i,j=0}^{7} O_{ijk})^2 \tag{3}$$

An ideal move tensor will only contain two 1s, with the rest being zero. If this tensor represents the correct move, $E_{\Sigma\Delta}$ will equal 0. Otherwise, this can take a value up to 4. The maximum possible value (for tensors containing values between 0 and 1) of $E_{\Sigma\Delta}$ is 128. $E_{\Delta\Sigma}$ will be 0 for any ideal move tensor, and can range anywhere between $\pm 128^2$ for possible output tensors.

3.4.2 Piece-wise

To more strongly match the outputs to the move cases I have collected, I will introduce the following function:

$$P_\omega(move) = \begin{cases} \omega & \text{if move in set} \\ 0 & \text{if move is valid} \\ P_{max} & \text{if move is invalid} \end{cases}$$

setting ω to a value between 0 and 1. Larger values will be more lenient in allowing for valid moves, whereas smaller values will favour moves in the data-set. P_{max} should be set to a significantly large value in order to discourage invalid moves.

For this, I will interpret the output as the closest ideal tensor, and use the tensor to LAN conversions from 3.3 to determine the move, which I will then match against my set of moves from 2.4.

3.4.3 Combination

The piece-wise function will drive the outputs towards valid and preferable moves, while the $E_{\Sigma\Delta}$ and $E_{\Delta\Sigma}$ functions provide granularity to the resulting loss value. The product of these functions should a comprehensive loss function, sufficient for my purposes.

$$L_\omega(T, O) = E(T, O)P_\omega(\text{tensor\_to\_lan}(O_{ideal})) \tag{4}$$

3.4.4 Using Position Evaluation

Another variation on the loss function I will consider is one which introduces the positional evaluation of the desired move. Taking the difference between the evaluation of the resulting board state and desired evaluation would produce a value which could be used influence the loss.

A Chess Notation

A.1 Forsyth–Edwards Notation

Shortened to FEN. Represents the state of the board by the location of each piece.

`https://en.wikipedia.org/wiki/Forsyth%E2%80%93Edwards_Notation`

A.2 Short Algebraic Notation

Shortened to SAN. Indicates a move by destination square, with prefixes in the case of ambiguity (if multiple pieces can reach the destination square).

A.3 Long Algebraic Notation

Shortened to LAN. Indicates a move using both source and destination square. Unambiguous.

A.4 Portable Game Notation

Shortened to PGN. A text-based file format allowing storage of meta-data relating to a game, such as the result, and including a SAN string of (pairs of) moves.

`https://en.wikipedia.org/wiki/Algebraic_notation_(chess)`