Chess Tactics Trainer

Saigautam Bonam
saigautambonam@gmail.com

TJHSST Computer Systems Lab

April 2021

Abstract

In professional chess competition, two very important skills are move calculation and pattern recognition. Move calculation allows a player to compute a variation without physical moves, and pattern recognition refers to the ability to spot specific ideas in a chess position. One of the most efficient ways to practice these skills is **Chess Tactical Training**. As most current tactical training resources have specific drawbacks, I aim to develop a personalized tactical training web application that would automatically generate tactics, classify them according to tactical motif, automatically generate a difficulty rating for each tactic, and present tactics to the user using a personalized algorithm. The following paper documents my research process.

1 Introduction

A **chess tactic** is a puzzle in which the player must find the best continuation of moves in a given board position, to a point where they have made a significant gain in the position. In the example FIGURE 1, Black has retreated their queen to the the square `c6`. This is a huge mistake, as now White can play the very strong move `Bb5`, winning Black's queen.

Another important term to introduce is the concept of a **tactical motif**. A tactical motif refers to an underlying theme involved in a tactical puzzle. A *pin* would be a tactical motif in FIGURE 1. In many tactical puzzles, there are several motifs, which would indicate that their solutions incorporate several themes.

Chess tactical puzzles improve move calculation as they require the player to find a single, optimal continuation of moves out of several combinations of moves to solve the puzzle. Additionally, many tactical puzzles share common tactical motifs, and repeated solving helps a competitive chess player reinforce their understanding of tactical ideas. Often, chess tactical skills have direct results in competitive tournament games, such as when one player spots a combination while their opponent misses it. Given the efficiency

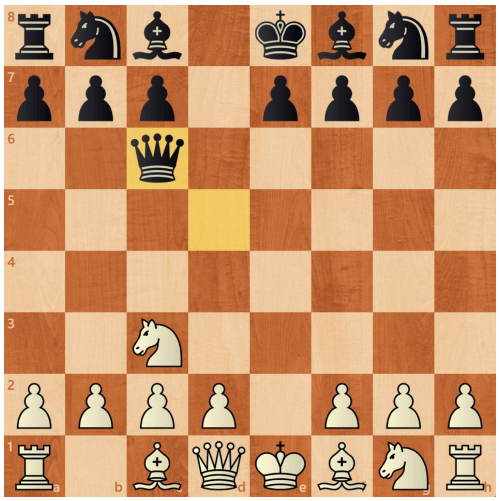


Figure 1: An example of a chess tactic.

of tactical training, it is unsurprising, then, that popular and universal chess websites have tactical training resources.

The following list contains three major tactical training sources for players to currently use:

- Chess.com
- Lichess
- Chess Tempo

Chess.com's tactic trainer is excellent and personalized, offering high-quality tactical puzzles which are grouped by motifs. However, it is not a free service—free-tier members can only solve a limited number of tactical puzzles a day.

Lichess, on the other hand is a completely free service. However, a couple months ago, Lichess did not group tactics by motif and offered tactical puzzles to the user only based on their tactic difficulty rating. In December 2020, however, they completely overhauled their tactical training system and introduced a new system with tactical motif classification and personalized training. Their willingness to change their tactical system shows the importance of the tactical training system to a professional chess player.

Finally, Chess Tempo is also mostly free and offers an unlimited number of tactical puzzle solving to free users. However, it does not offer a personalized training experience to free users, and additionally, its tactical motif grouping is based on user votes and is not automatically generated.

Given the fact that all three of these major tactical training sites have or have had specific drawbacks, I decided to take on the project of building a tactical training application that would automatically generate tactics, classify them according to tactical motif, automatically generate a difficulty rating for each tactic, and present tactics to the user

using a personalized algorithm. Once the project is completed, I plan to release it to the chess community for free.

2 Methods and Technical Implementation

I will now discuss my research process in the following order:

1. Tactic Generation
2. Tactic Motif Classification
3. Tactic Difficulty Calculation
4. Web Application and User Interface

During the implementation of the project, I had actually started with the Web Application and User Interface, and added functionality to the application as I continued working on the project. For explanation purposes, however, I will discuss this step last.

2.1 Tactic Generation

To generate a set of chess tactics, I needed a database of chess games to work with and a chess engine to evaluate board positions and calculate best moves. For the chess engine, I used the open source and free chess engine Stockfish, which is currently one of the best chess engines in the world.

For the database, my initial choice was Caissabase, a large database of more than 4 million international over-the-board chess games. However, I soon realized that it would take a very long time to process the database and generate a number of meaningful tactics, which I will explain below. I switched to Lichess's open-source databases of all online chess games played on its server. Importantly, these databases contain Stockfish board evaluations calculated at depth 22 for 6% of chess games. Below is the pseudocode of the algorithm I used to generate tactics. The database was in Portable Game Notation (PGN) format, and I used the library `python-chess` for iterating over the database, handling Stockfish calls, and chess functionality.

The algorithm consists of two main steps: identifying the possibility of a tactic based on position evaluations, and building the sequence of moves, or variation, once a tactic is discovered.

Algorithm 1: Generate list of tactical puzzles

```
1  tactics ← list()
2  prevEvaluation ← 0
3  foreach game in database do
4      foreach position in game do
5          newEvaluation ← position.evaluation
6          if significantDifference(prevEvaluation, newEvaluation) then
7              continuation ← copy(position)
8              while singleBestMove(continuation) do
9                  moveSequence ← list()
10                 moveSequence.append(continuation.bestMove)
11                 continuation.makeMove(continuation.bestMove)
12                 moveSequence.append(continuation.bestMove)
13                 continuation.makeMove(continuation.bestMove)
14             end
15             if moveSequence is not empty then
16                 moveSequence.removeLastElement()
17                 tactics.append(tuple(position, moveSequence))
18             end
19         end
20         prevEvaluation ← newEvaluation
21     end
22 end
23 return tactics
```

The Lichess databases were much faster to process, as I only iterated through the games which had Stockfish evaluations. As a result, obtaining the position's evaluation (line 5) was a simple lookup. With Caissabase, this would require a Stockfish evaluation call, which would be a large bottleneck when processing several thousands of positions.

Two particularly important areas of the above algorithm are the functions SIGNIFICANTDIFFERENCE and SINGLEBESTMOVE. The former uses several parameters to determine if the change in evaluation when iterating through each position in the game constitutes a mistake by one side, while the latter determines if the player to move has only one best move in the position. The SINGLEBESTMOVE function is the primary bottleneck of this algorithm, as it relies on the Stockfish engine to calculate the evaluations for every possible move in the position, and then uses a method similar to the SIGNIFICANTDIFFERENCE function to ascertain if second best move is comparable to the first. If the position passes both of these checks, a tactic is found, and the algorithm starts building the move sequence.

The move sequence starts with the player's best move (which is returned separately from SINGLEBESTMOVE to minimize cost) and adds the opponent's best reply. The algorithm repeats this while the player has a single best move, as this would indicate the combination of moves must continue for the puzzle to be fully solved. After the player has more than

one good move, the tactic sequence stops, the previous opponent reply is removed, and the tactic is stored.

An important consideration I made with this algorithm was the evaluation depth of Stockfish used in the singleBestMove function and the calculation of the opponent's best reply in line 12. Higher depth resulted in more accurate and potentially challenging tactics but required more computation and thus more time, whereas lower depth was faster but occasionally resulted in inaccurate tactics. I ended up choosing a depth of 12-ply as a balance. The algorithm was still slow, however, as another factor was the interface between the Python script and the UCI protocol of the Stockfish engine executable.

In total, I generated 17,000 tactical puzzles from two Lichess databases (July 2016 and July 2019), and after refining the significantDifference and singleBestMove functions, I added 7,000 of these into the web application.

2.2 Tactic Motif Classification

For this problem, I had to find a way to automatically tag tactical puzzles with tactical motifs. This would allow users of the application to see the tactical motifs corresponding to each tactic without the need for crowdsourcing.

Initially, I had been planning to use a form of Machine Learning to associate motifs with tactics. However, I soon realized that this was impractical for two reasons: (1) there was no readily available dataset containing both tactics and canonical tactical motif classifications, and (2) there was not a clear way to represent tactics as feature vectors to a model which would encapsulate all the required attributes. I attempted to get around issue 1 by using an unsupervised K-Means clustering approach. However, I noticed that the tactic positions contained in the cluster after convergence were not similar in solutions, but only board attributes like piece location and number of pieces, which would make sense given that the algorithm minimizes Euclidean distance in each cluster. Instead of a Machine Learning approach, I decided to implement algorithms to detect individual motifs which take as input the initial board position and the tactic variation.

Some tactical motifs, are simple to calculate as they rely on tangible parameters associated with the initial board position or tactic move sequence. For example, to ascertain if a tactic should contain the *endgame tactic* motif, the algorithm can just count the number of pieces in the board and check if it is below a certain threshold. However, many tactical motifs are harder to determine algorithmically, as the threat of the motif itself is enough to add the motif to the tactic. An example of this is the *back rank* tactic, which is defined as checkmate along the top or bottom of the board. In Figure 2, Black checkmates White using the back rank motif. In Figure 3, however, Black wins a rook by the threat of back rank checkmate if White takes back the rook. Both of these tactics contain the back rank motif, but only one of them ultimately results in checkmate.

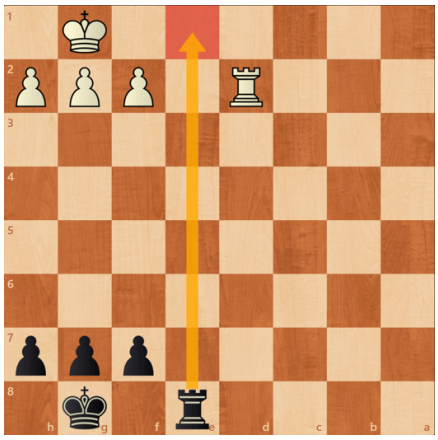


Figure 2: Back rank checkmate

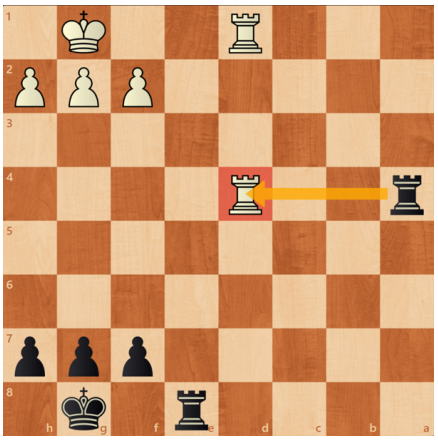
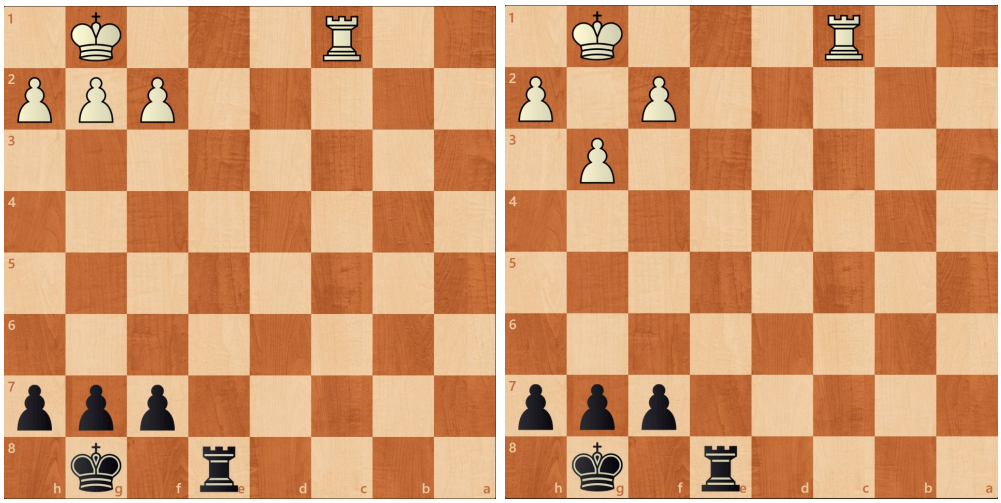


Figure 3: Back rank threat

I addressed this by implementing an algorithms which detect the possibility of a back rank checkmate given board position and color (see FIGURE 4) and construct a search tree of depth 3 containing player-opponent moves continuing the variation. As illustrated in FIGURE 5, if any nodes in the tree continuing the variation are terminal states (checkmate) and satisfy the back rank possibility check, the algorithm infers that a back rank threat exists in the position, and so assigns the motif to the tactic. This algorithm is able to detect back rank threats in many situation, but occasionally also generalizes the motif to other puzzles where back rank is not relevant. These kind of algorithms could potentially be required for similar tactical motifs, where only a threat would be enough for classification of that motif.



(a) True (b) False

Figure 4: Back rank heuristic from Black's perspective

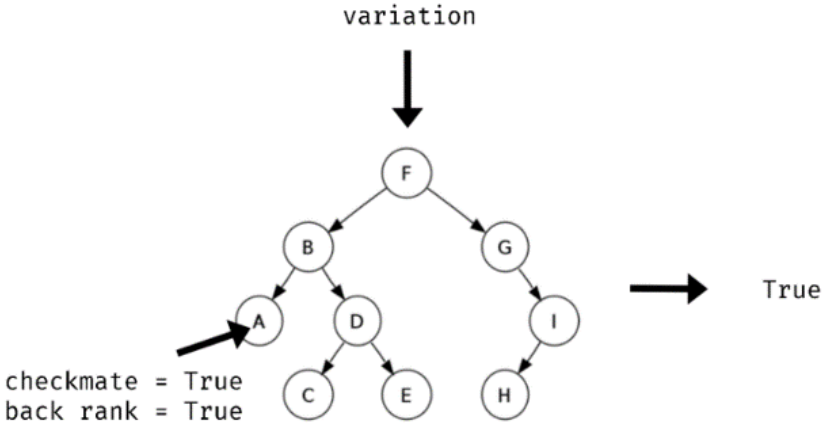


Figure 5: Back rank detection algorithm

Other tactical motifs were relatively simpler to calculate, but still required some level of inference from the tactic move sequence. For example, the *hanging piece* tactical motif is assigned when the tactic move sequence involves capturing an undefended piece or a piece of higher value. Below is the pseudocode of the motif's detection algorithm:

Algorithm 2: Detect *hanging piece* tactical motif

```
i ← 0
for move in variation do
    pieceMoved ← board.pieceAt(move.fromSquare)
    if board.isPieceAt(move.toSquare) and i % 2 == 0 then
        pieceCaptured ← board.pieceAt(move.toSquare)
        oppositeColor ← pieceCaptured.color
        if notDefended(move.toSquare, oppositeColor) then
            if pieceCaptured.value ≥ 3 then
                return True
            end
        end
        if pieceMoved.value < pieceCaptured.value then
            return True
        end
    end
    i ← i + 1
end
return False
```

Ultimately, I implemented detection algorithms for 10 other tactical motifs before moving on to the final portion of the project: *checkmate*, *discovered check*, *windmill*, *sacrifice*, *endgame tactic*, *opening tactic*, *perpetual check*, *defense*, *stalemate*, and *attacking f2/f7*.

2.3 Tactic Difficulty Calculation

This was the most interesting and seemingly difficult portion of the project. To illustrate why I needed to find a way to predict a tactic difficulty rating automatically for each tactic, I will provide a bit of background information.

Chess websites with tactical training resources start tactics out at a fixed rating, for example 1500. As their users attempt tactics, their server keeps track of results, changes tactic rating, and decreases the variability of the tactic rating as more users attempt the tactics. FIGURE 6 below illustrates how tactic ratings change as more users attempt them. In the case of my web application, however, there would not be enough users for tactic ratings to stabilize—for this reason, tactics should start out at a rating similar to their true difficulty.

This seems like a very abstract problem, but I found a similar research paper that aimed to learn tactic difficulty using Machine Learning classification. Stoiljkovikj, Bratko, and Guid in 2015 approached this problem, and managed to achieve 80 percent accuracy with classifying chess tactics as either "Easy," "Medium," or "Hard."[1]

For each tactic, the researchers constructed a *meaningful search tree*, a variant of the full two-player search tree with only nodes relevant to the tactic itself. These search trees did not consider any moves which were clearly worse than the best moves by both the player to move in the tactic and potential opponent replies. After acquiring search trees for each tactic, the authors gathered several heuristics in each search tree (shown in FIGURE 7), and then input those heuristics as features into several Machine Learning classifier models. Using several different models (ranging from a neural network to a random forest), they were able to get above or close to 80 percent accuracy with the classification task.

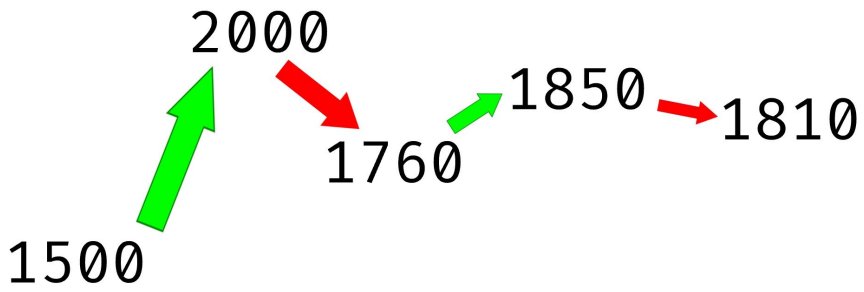


Figure 6: Tactic rating change illustration

Table 1: Attributes and their basic descriptions.

#	*Attribute*	*Description*
1	Meaningful(L)	Number of moves in the meaningful search tree at level L.
2	PossibleMoves(L)	Number of all legal moves at level L.
3	AllPossibleMoves	Number of all legal moves at all levels.
4	Branching(L)	Branching factor at each level L of the meaningful search tree.
5	AverageBranching	Average branching factor for the meaningful search tree.
6	NarrowSolution(L)	Number of moves that only have one meaningful answer, at level L.
7	AllNarrowSolutions	Sum of NarrowSolution(L) for all levels L.
8	TreeSize	Number of nodes in the meaningful search tree.
9	MoveRatio(L)	Ratio between meaningful moves and all possible moves, at level L.
10	SeeminglyGood	Number of non-winning first moves that only have one good answer.
11	Distance(L)	Distance between start and end square for each move at level L.
12	SumDistance	Sum of Distance(L) for all levels L.
13	AverageDistance	Average distance of all the moves in the meaningful search tree.
14	Pieces(L)	Number of different pieces that move at level L.
15	AllPiecesInvolved	Number of different pieces that move in the meaningful search tree.
16	PieceValueRatio	Ratio of material on the board, player versus opponent.
17	WinningNoCheckmate	Number first moves that win, but do not lead to checkmate.
18	BestMoveValue	The computer evaluation of the best move.
19	AverageBestMove(5)	Average best-move value of all best moves occurring at level 5.

Table 2: The values of multi-level attributes for the example in Fig. 4.

#	*Attribute*	*level 1*	*level 2*	*level 3*	*level 4*	*level 5*
1	Meaningful(L)	1	1	1	5	5
2	PossibleMoves(L)	47	22	39	23	178
4	Branching(L)	1	1	1	5	1
6	NarrowSolution(L)	1	1	1	0	5
9	MoveRatio(L)	0.021	0.045	0.026	0.217	0.028
11	Distance(L)	3	1	2	13	16
14	Pieces(L)	1	1	1	2	2

Figure 7: Heuristics (from Stoiljkovikj et. al)

The authors' methods seemed very understandable and a fitting solution to the task, although I needed to predict a specific difficulty *rating* instead of a classification. I implemented the author's methodology to generate the heuristics, but input features into a neural network for regression instead of classification. I started by building the meaningful search tree to a depth of 5, similarly to the researchers. The following pseudocode demonstrates the algorithm I created to generate these trees:

Algorithm 3: Generate *meaningful search tree*

```
   Data: board, variation: list()
 1 globalNodeList← list()
 2 root ← Node()
 3 board.makeMove(variation[0])
 4 layerOneNode ← Node(board, variation[0], board.evaluation, 1)
 5 root.addChild(layerOneNode)
 6 globalNodeList.append(root, layerOneNode)
 7 foreach depth in [1, 3] do
 8     foreach node in globalNodeList do
 9         if node.isLeaf() and node.depth == depth then
10             board ← node.position
11             board.makeMove(board.bestMove)
12             bestEvaluation ← board.evaluation
13             board.undoMove()
14             foreach legalMove in board.legalMoves do
15                 board.makeMove(legalMove)
16                 if board.evaluation.closeEnough(bestEvaluation) then
17                     opponentNode ← Node(board, legalMove, board.evaluation,
                        depth+1)
18                     node.addChild(opponentNode)
19                     globalNodeList.append(opponentNode)
20                     board.makeMove(board.bestMove)
21                     evaluation ← board.evaluation
22                     board.undoMove()
23                     foreach playerMove in board.legalMoves do
24                         board.makeMove(playerMove)
25                         if board.evaluation.closeEnough(evaluation) then
26                             playerNode = Node(board, legalMove, board.evaluation,
                                depth+2)
27                             opponentNode.addChild(playerNode)
28                             globalNodeList.append(playerNode)
29                         end
30                         board.undoMove()
31                     end
32                 end
33                 board.undoMove()
34             end
35         end
36     end
37 end
38 return root, globalNodeList
```

In line 16, the algorithm determines if an opponent move is similar enough to the opponent's best move to be worth including in the search tree by calling the function CLOSEENOUGH. This function uses two parameters, w and m. w determines if the position is won, while m determines the acceptable difference between two evaluations. I used values $w = 300$ and $m = 50$, where the units are in *centipawns*, or cp. In this measurement system, 100cp is roughly equivalent to one pawn.

If the check passes, the algorithm adds a node corresponding to the move, and then repeats this process for the player's response to the opponent move. This occurs in lines 23-30. FIGURE 8, originally from Stoiljkovikj et. al., provides an example of a meaningful search tree for a difficult tactic.

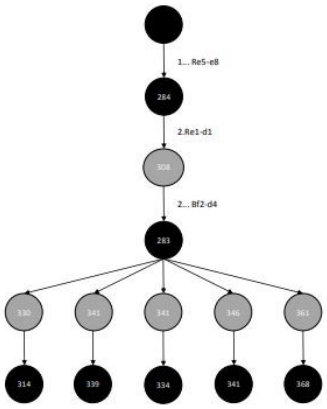


Figure 8: Meaningful search tree (from Stoiljkovikj et. al)

The tree was built for a tactic in which Black is to move. In layers 3 and 5, there is always only one response from Black to each of White's responses, demonstrating Black needs to find a precise response for each of White's responses all the way to depth 5. In simpler tactics, for example, Black might have several good responses at depth 5, or there may fewer options for White at depth 4.

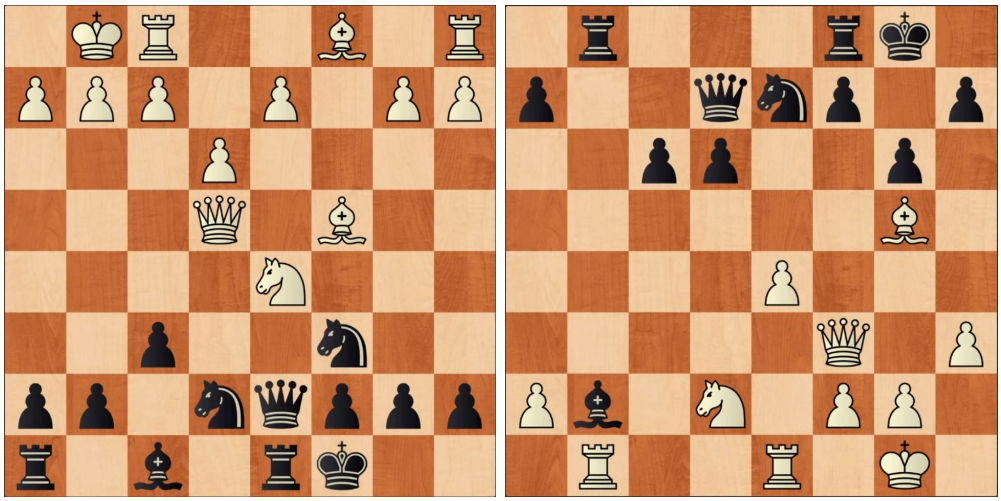
In my implementation of the tree-generation algorithm, I noticed that the simpler tactics were, the longer generation took for them. This was understandable, as often simple tactics might only require one move (such as simple piece capture). The algorithm would then find that almost all of both sides' moves are equivalent to each other, as the tactic would be completed. Due to the fact that the algorithm uses Stockfish to calculate the best move and evaluations for all nodes, the algorithm takes quite a while for these tactics. As a result, I removed any tactics which were one move long and had the *hanging piece* classification attached (see Section 2.2) to make the generation process faster, and assigned these tactics a rating of 600. I also removed any tactics that were checkmate-in-two moves, because a search tree of depth 5 was impossible to generate, and assigned these a rating of 1200. Since the data input to the model had to be uniform, it would not be possible to include trees of depth 3 when calculating heuristics.

I then implemented the heuristic calculation algorithms for the first 10 heuristics, as the authors had mentioned that the first 10 heuristics had been enough for 80 percent accuracy with the classification task. It's important to note, however, that some of the heuristics are calculated at each level of the tree, for example *NarrowSolution* (see FIGURE 7). The feature vector containing all the first ten heuristics appended together had size 30.

I then gathered a set of roughly 1100 rated chess tactics from lichess.org to use as a dataset. 200 of these were separated into the test set. After gathering feature vectors for each tactic, I then built a neural network and trained it on the data.

The final neural network architecture is a combination of Dense layers and Rectified Linear Unit activations. As the neural network performs regression, the output layer consists of one node, and the loss is Mean Squared Error. The current architecture is [50, 125, 100, 50, 25, 1] with ReLUs between each layer.

After training for 10 epochs on the entire training set, I found that the model didn't seem to improve past a certain point—roughly 140,000 mean squared error on the test set. This corresponds to an average rating deviation of roughly 370 rating points, but I noticed that the model still demonstrates a general understanding of tactic difficulties relative to each other. This is consistent with the results of Stoiljkovikj et. al, as they used a classification task which generalized the difficulty of each tactic. FIGURE 9 shows an example of two tactics—one relatively simple and one hard. The model predicts a rating of 1098.5 for the simpler tactic, and a rating of 1915.99 for the harder tactic, which is quite a large difference. This was enough for my needs, as I needed difficulty ratings that were accurate relative to each other; they did not need to be precise.



(a) Black plays Nxd5 and wins a piece for free. Model predicts a rating of 1098.5.

(b) White plays the sacrifice Rxb2 followed by Qf6, which wins material through a mating net. Model predicts rating of 1915.99.

Figure 9: Two different tactics. (a) is a relatively simple tactic, while (b) is much harder

2.4 Web Application and User Interface

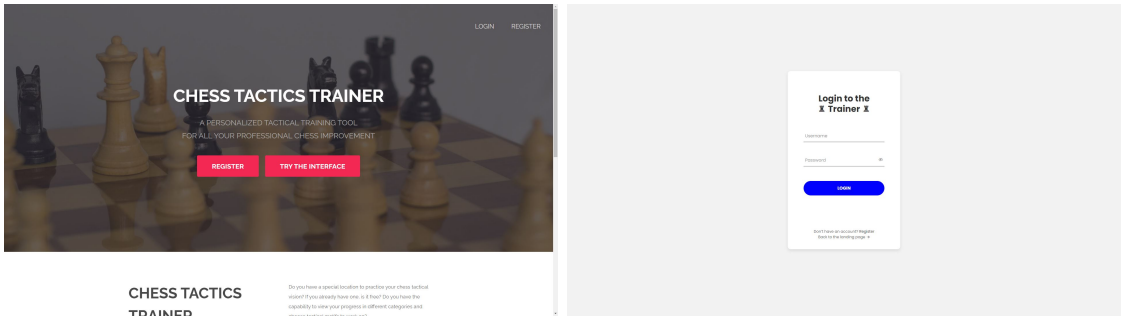
While I worked on tactic generation, motif classification, and difficulty calculation, I simultaneously worked on updating and editing the web application, where the trainer is located.

I began by creating a base application using the Django web framework, which uses Python. I had initially planned to host the site on TJ's Director, but I found that Director's Docker containers were too slow in installing packages and running files (this was in fall 2020, so changes have likely been made). Instead, I developed the application on my local machine (localhost) for the first half of my project, and deployed the application to the free service Heroku in February 2021. You can access the production website here: chess-tactics-trainer-5478tjsr.herokuapp.com.

The user login/register system uses Django's built-in authentication system. I configured Django to use a SQLite database, which stores all information in a single file with a `.sqlite3` extension. The user is stored in a parent class called `Player`, which holds information about the piece set to render (which I will discuss later), the player's rating, the player's dark mode setting, and the user login information.

The database also stores tactics in a separate table, which is associated with the `Tactic` class. This stores the initial tactic board position, the Stockfish evaluations corresponding to the tactic, the best move, the move sequence, motif classifications, tactic rating, the side to move, and the source of the tactic (which was useful when I had processed different sets of tactics over the course of the project).

FIGURE 10 contains screenshots of the landing and login pages. The register page is similar.

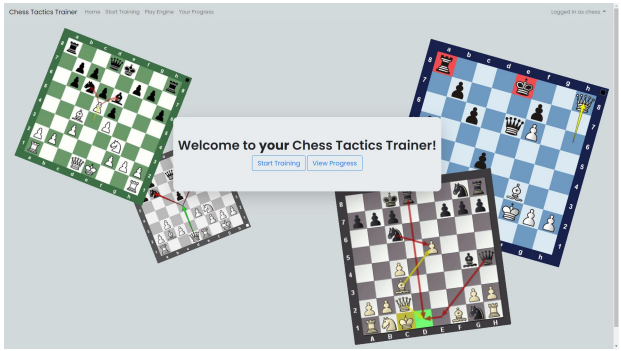


(a) Landing page (b) Login page

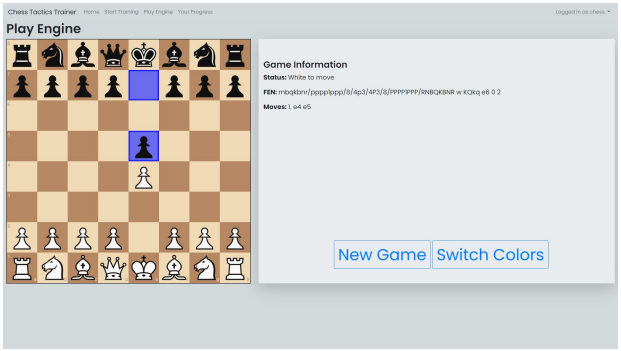
Figure 10: Landing and login pages

The landing page of my application is a modified version of a free HTML/CSS template online. The login/register pages were also based on another free template. Both templates have been cited in the GitHub repository of the project (see Section 3).

FIGURE 11 contains screenshots of components of the main application.

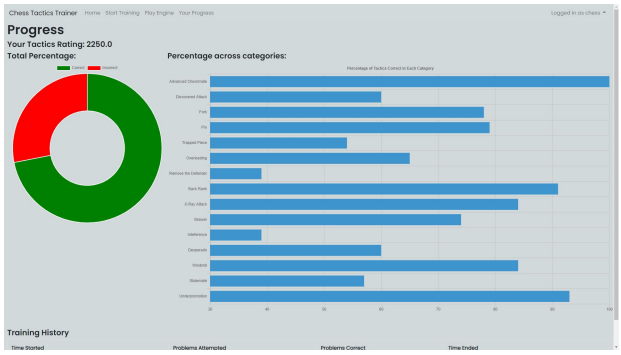


(a) Home page

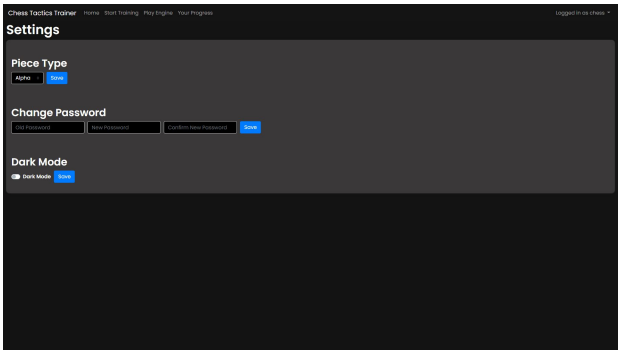


(b) Play Engine page

(c) Training page.



(d) Progress page



(e) Settings page (in dark mode)

The chessboard on the Play Engine and Training pages is rendered using the `chessboard.js` library. I allowed the piece set to be changed via the settings page. The Django server

stores the piece set preference in the `Player` model, and simply includes the correct piece file locations when rendering the board.

Some important areas of the Training page are the button array at the bottom of the chessboard, the analysis board at the bottom right, and the classifications text section. The four buttons directly below the chessboard allow the user to move back and forth through the *correct* move sequence. For example, if the user made an incorrect move, the tactic would end, and they would be able to view the correct answer. The analysis board at the bottom right is included via an `iframe` from lichess.org. As Lichess's analysis board can be configured to display a certain position by amending the URL to the analysis board, I was able to add the FEN (Forsyth–Edwards Notation) string of the initial tactic position to the URL. This allows the user to directly analyze the position without having to open an external application. Additionally, the classifications bar displays all the current motif classifications associated with a tactic. The "+" button allows the user to add or remove tactical motifs from this list by opening a modal.

The rating bar below the classifications bar displays the rating of the tactic predicted by the model. Finally, when the user selects "continue solving", the server randomly chooses a tactic from the database and displays it to the user. The connection between the server and the training webpage is maintained with `POST` requests and `JSON` responses between Django and JavaScript.

When aiming for fast request/response times between Django and JavaScript, it is important to make sure that there are no functions running in the response which take a couple seconds to process. For example, when I had initially been loading tactics from a pickled data structure object file, my code had loaded the tactic data structure every time a request was made, causing the board to freeze for two seconds before the tactic appeared. Such loading of data structures should occur when the server starts, so that each request only executes simple actions.

The Progress page aimed to contain detailed information about the user's overall statistics with solving tactics, along with information about the user's performance in each tactical motif. Right now, the page contains only static information, and does not dynamically update as users complete tactics.

Finally, the settings page contains settings for the piece set, password change, and dark mode. When preferences are updated, Django updates the player instance in the database. The screenshot shows the result of a user turning on Dark Mode.

3 Future Work and Conclusion

There are several ways this project could be developed in the future. Firstly, the tactic generation and motif classification algorithms could be improved and expanded for higher-quality tactics and motif classification. Additionally, the difficulty prediction model's precision could be improved by gathering more training data, implementing more heuristics, or optimizing the network's architecture. Finally, the application itself could be more dynamic and personalized by offering tactics to the user based on the user's past performance

as well as updating the user's statistics and ratings after a user attempts a tactic.

Ultimately, though, I'm very happy with the progress I've made over the course of this project, and I believe that the chess tactics trainer is well on its way to completion and potential introduction into the chess community.

I can be contacted at saigautambonam@gmail.com if you have any questions regarding my research process.

References

[1] Simon Stoiljkovikj, I. Bratko, and Matej Guid. A computational model for estimating the difficulty of chess problems. 2015.