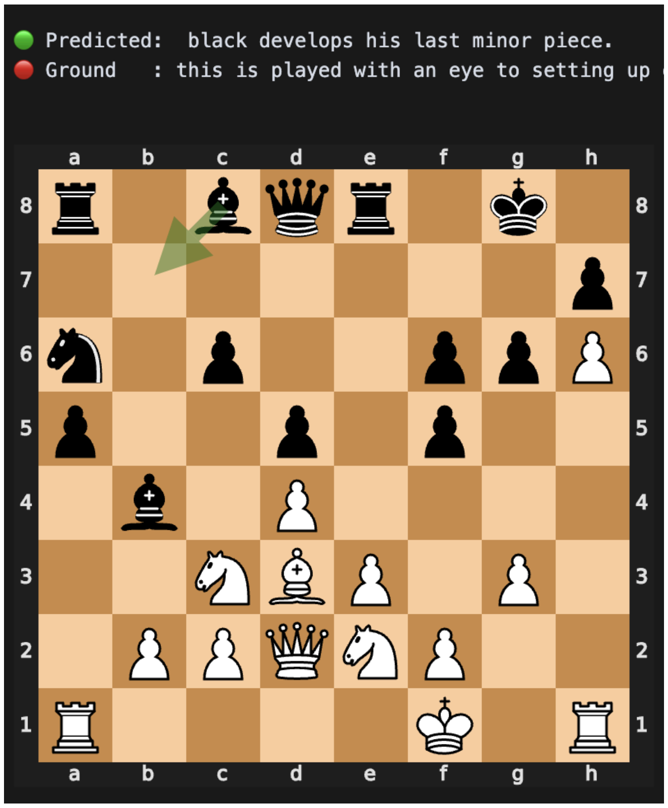
Author: Kerem Kazan
Title: Chess Commentator Transformer

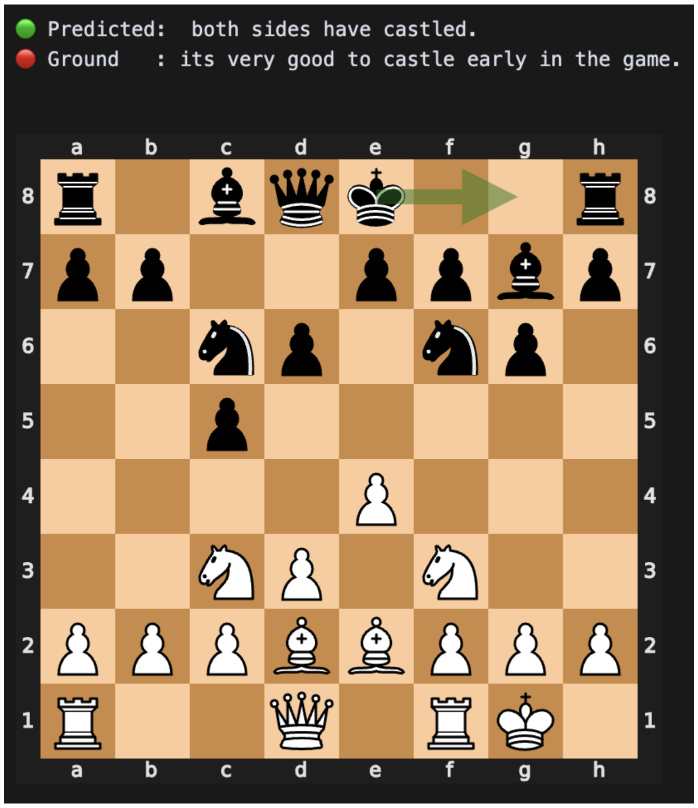
Motivation: Chess board analysis is an endeavor as old as time. There are plenty of analysis engines out there, such as Stockfish. These engines assign numerical scores to boards by simulating moves and attempting to understand which paths lead to certain victory, or certain defeat. These engines will never show you the idea, or the thinking behind these moves - like a teacher would. Some "coach bot"s offer rules-based hand-coded algorithms that can be reduced to a series of nested if-statements that will never capture the deep, interesting patterns that keep emerging in a game as complex as chess. So the issue remains: we need a natural language chess bot that can offer comments on chess boards and moves.

Dataset: Almost all chess datasets match boards and moves to winners or scores, but not comments. Upon inspecting prior art, many references to the Gameknot.com chess forum games came up, along with outdated code that was meant to scrape them. Solution: write the scaping code from scratch. The result was 12449 games and 278,000 rows of (board,move,comment) tuples. This dataset is not publicly available, so I'm only able to share a small sample of 500 rows: https://tinyurl.com/yck7twmc

Approach: Use an encoder-decoder architecture - specifically BERT and GPT2. Manually sequence the (board,move) inputs and train a compact tokenizer. Use a world-level tokenizer of size 1000 for the comments. Train on A100.

Results: The model is coherent, and able to make useful comments. However, it's prone to repeating common phrases from the dataset, such as "I think this is a good move". Below are 2 cherry-picked examples of good board understanding.


```
🟢 Predicted:  black develops his last minor piece.
🔴 Ground   : this is played with an eye to setting up
```


```
🟢 Predicted:  both sides have castled.
🔴 Ground   : its very good to castle early in the game.
```

Step 1: Data Mining

The data needed to run this project is not publicly available. I had to manually scrape it, sanitize it and preprocess it. Here's a 500 row sample of the data that I have made publicly available: https://docs.google.com/spreadsheets/d/1U_yUzUpn3PKREU6L0h5ixjgGPh3RDrTm4RnLjg7HrBA/edit?usp=sharing

If you want to access the 300k rows of data I have mined and pre-processed, just let me know and I'm happy to share it. If you instead want to scrape and build it from scratch, all the necessary steps are outlined in `Data_Mining.ipynb`, along with code and commentary that walks you through. This part makes use of `bs4` and `selenium` for headless browsing. The data was tricky to obtain, because it required javascript interactions and mouse clicks. The python script in the notebook uses `selenium` to achieve this, and then bs4 for html parsing.

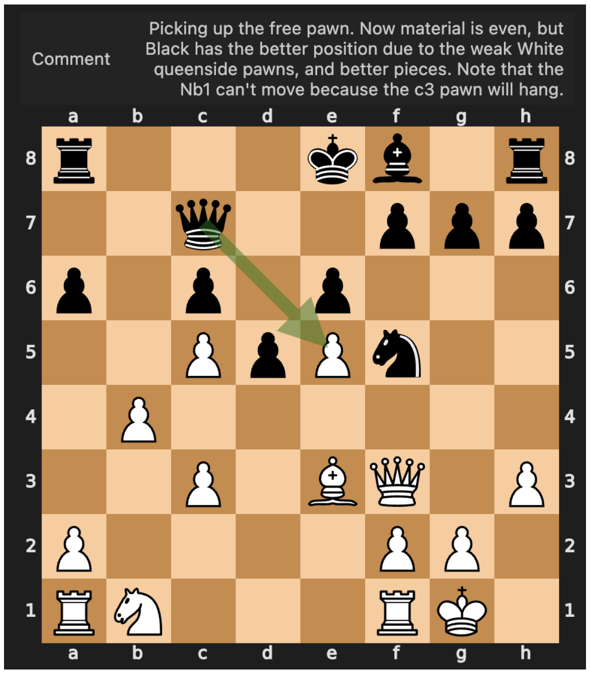
In the same notebook, we do a few quick rounds of data sanitization to remove unusable rows. A couple of examples:

- "The French Defence happens to be the only opening I have a book on, and there is a rather nice story behind it. I am a supply teacher - what our American cousins would call a substitute teacher. I go round all sorts of schools and colleges. I was in a staff room, browsing at this book, and someone said that the owner of the book had left, so if I wanted I could have it. At that time my girlfriend lived in London, and phoned her and mentioned that I had been given a book on The French Defence. The football world cup finals had just started and she said 'Isn't there anything about the French midfield or the French attack?' She shared by love of football, but not of chess!"
- "Und auf einmal taucht auch die Dame auf."
- "Can you believe? :-)"

We employ manual regex and filters to remove these bad rows. Language detection often produces false positives - which sometimes removes good rows - but we have too much data to worry about that.

Step 2: Data Quality and Inspection

We make extensive use of the python-chess library to visualize and analyze chess boards and moves. Here's an example:


Comment

Picking up the free pawn. Now material is even, but Black has the better position due to the weak White queenside pawns, and better pieces. Note that the Nb1 can't move because the c3 pawn will hang.

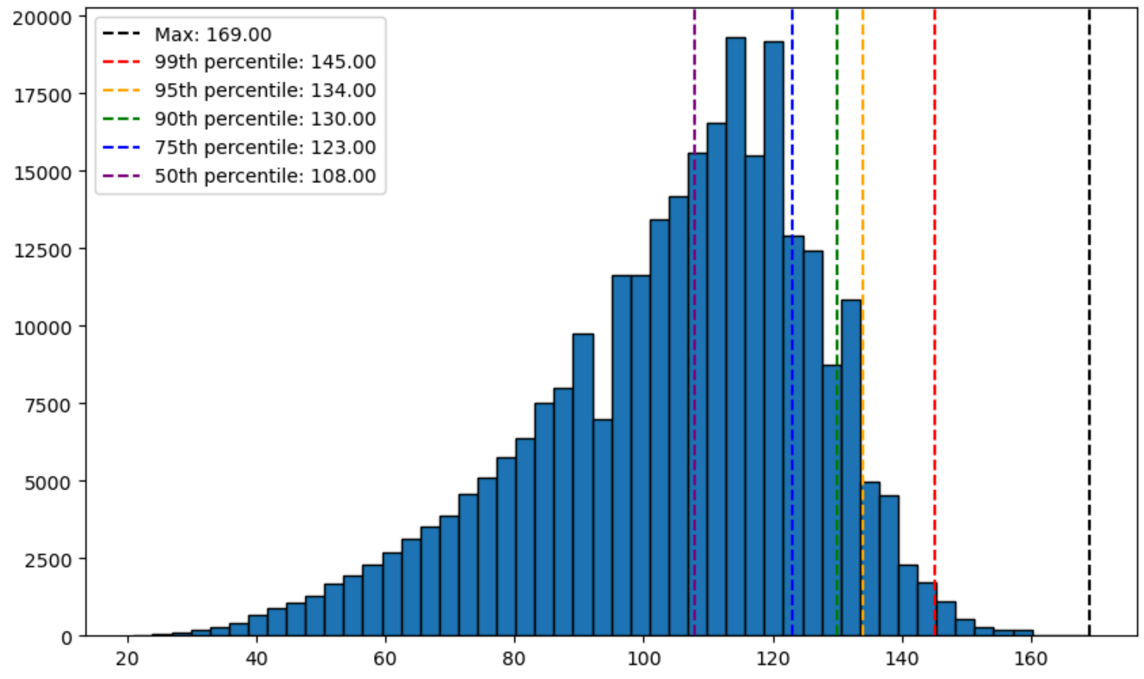
Step 3: Outlier Removal
We remove the top 1% of tokenized comment lengths to obtain a more uniform and workable dataset. More can be found here: `Data_Quality.ipynb`.

Step 4: Serialization
The chess data standards FEN and UCI are compact data formats that contain lots of information about the game and move, but they are hard for transformers to learn from. This is why we create our own serialization scheme that maps the input data into a more understandable sequence. We are essentially helping the transformer with prior domain knowledge here. Here's an example sequence:

> [CLS] WHITE_BISHOP ON SQUARE_E5 MOVES_TO SQUARE_G3 [SEP] WHITE_BISHOP ATTACKS BLACK_PAWN ON SQUARE_C7 [SEP] WHITE_BISHOP DEFENDS WHITE_PAWN ON SQUARE_F2 WHITE_PAWN ON SQUARE_H2 [SEP] WHITE_BISHOP CONTROLS SQUARE_F2 SQUARE_H2 SQUARE_F4 SQUARE_H4 SQUARE_E5 SQUARE_D6 SQUARE_C7 [SEP] WHITE_ROOK ON SQUARE_A1 WHITE_QUEEN ON SQUARE_D1 WHITE_KING ON SQUARE_E1 WHITE_BISHOP ON SQUARE_F1 WHITE_ROOK ON SQUARE_H1 WHITE_PAWN ON SQUARE_B2 WHITE_PAWN ON SQUARE_E2 WHITE_PAWN ON SQUARE_F2 WHITE_PAWN ON SQUARE_G2 WHITE_PAWN ON SQUARE_H2 WHITE_PAWN ON SQUARE_A3 WHITE_KNIGHT ON SQUARE_C3 WHITE_KNIGHT ON SQUARE_F3 WHITE_PAWN ON SQUARE_D4 BLACK_PAWN ON SQUARE_D5 WHITE_BISHOP ON SQUARE_E5 BLACK_KNIGHT ON SQUARE_H5 BLACK_KNIGHT ON SQUARE_C6 BLACK_PAWN ON SQUARE_F6 BLACK_PAWN ON SQUARE_A7 BLACK_PAWN ON SQUARE_B7 BLACK_PAWN ON SQUARE_C7 BLACK_BISHOP ON SQUARE_E7 BLACK_PAWN ON SQUARE_G7 BLACK_PAWN ON SQUARE_H7 BLACK_ROOK ON SQUARE_A8 BLACK_BISHOP ON SQUARE_C8 BLACK_QUEEN ON SQUARE_D8 BLACK_KING ON SQUARE_E8 BLACK_ROOK ON SQUARE_H8 WHITE_HAS_KINGSIDE_CASTLING_RIGHTS WHITE_HAS_QUEENSIDE_CASTLING_RIGHTS BLACK_HAS_KINGSIDE_CASTLING_RIGHTS BLACK_HAS_QUEENSIDE_CASTLING_RIGHTS [SEP]

This is the serialization of a single board and move. It looks like a lot, but a word-level tokenizer sees this as a very manageable sequence input. All of this can be seen in `Tokenization.ipynb`.

Step 5: Tokenization
Using the serialized input data from the previous step, we train a custom word-level tokenizer for our encoder. The result is a vocabulary size of 100, which is extremely compact. This is excellent though, because it will reduce the model size, increase the training speed, and remove unnecessary noise. One potential downside of small vocabularies is longer tokenized sequences - but this was not the case here. Here's a tokenized input length distribution. Once again, all of this can be seen in `Tokenization.ipynb`.

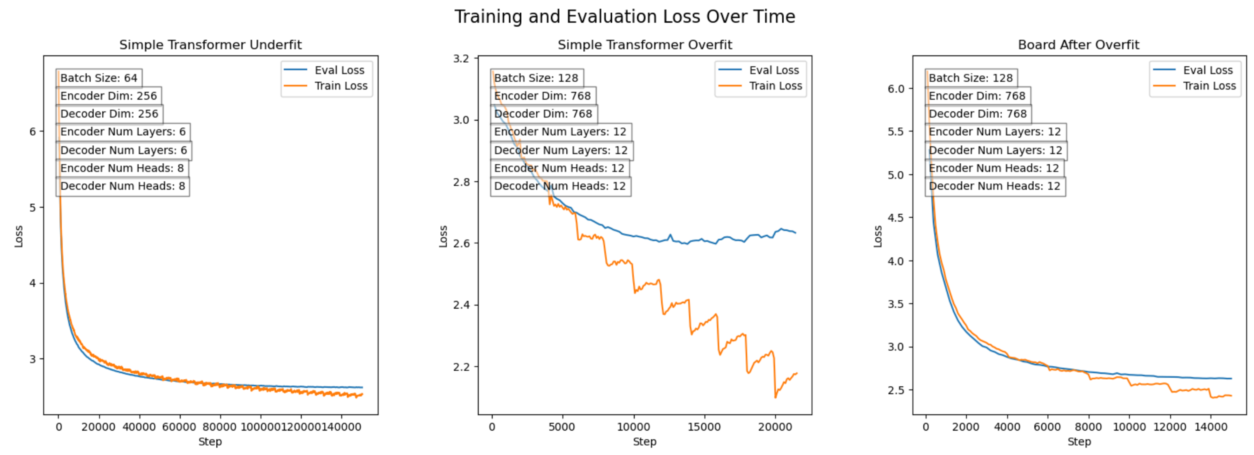


Step 6: Training

We use the Hugginface `transformers` library to setup an encoder-decoder model. We use BERT for the encoder, and GPT2 for the decoder. We supply both with our own trained tokenizers. The resulting network is 11M parameters-strong - so it's appropriate to categorize it as a "tiny" transformer. All of this can be seen in `Training.ipynb`.

The best performing model was trained on an A100 for 5 hours via google colab.

Step 7: Results



Run 1 - Alias: Simple Transformer Underfit

The first plot's original run can be found here: https://colab.research.google.com/drive/1tCtRpZRnNv6Ta6qG9ZFcOab-E2_0NbpW?usp=sharing
. This is the analysis of the training run you'll find in `Training.ipynb`. It shows a very healthy training curve - which took 5 hours and close to 150k steps to complete. We can see that training loss caught up to the eval loss at around 100k steps - but the difference isn't too big. I consider this run a success. Upon inspecting the actual generated comments, I noticed that there was room for improvement. So I ran the same model by basically doubling everything, which resulted in the second plot.

Run 2 - Alias: Simple Transformer Overfit

Original run: https://colab.research.google.com/drive/10rTpBn8lPTQl8jFBpgEjjwZj0viPjYFF?usp=sharing

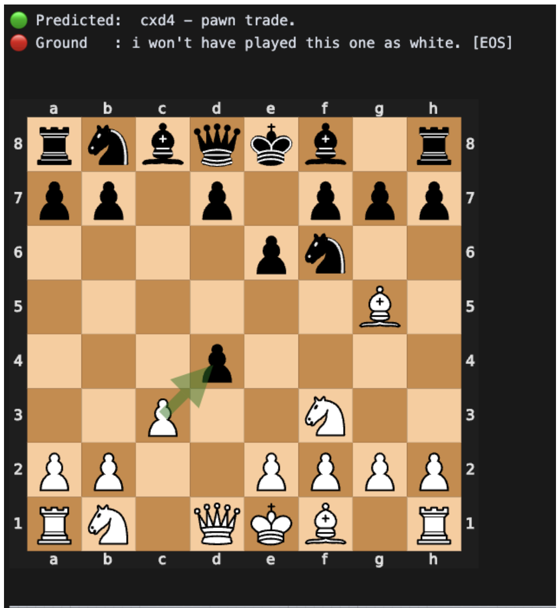
This one shows a dramatic overfit pattern that shows up pretty quickly. This run was an obvious failure - but it was worth a shot.

Run 3 - Alias: "Board-After" Overfit

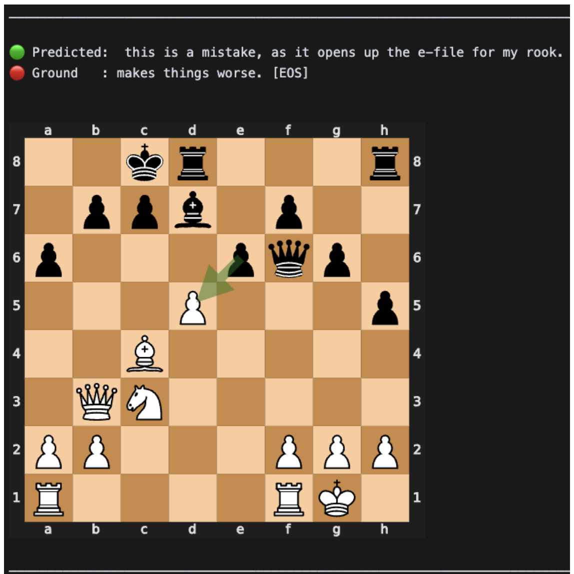
Original run: https://colab.research.google.com/drive/1JB2Bywfugo1LjaeJtRzLdGnkCGRagq4-?usp=sharing

This one uses a different tokenization scheme. Instead of tokenizing the input as BOARD + MOVE sequences, it ignores the move sequence and tokenizes the board both before and after the move - in an attempt to capture the shift in the board position. This too was an overfit - but it was much better than the previous one. This run was also a failure - but I believe there's merit to this strategy - and it should be explored further.

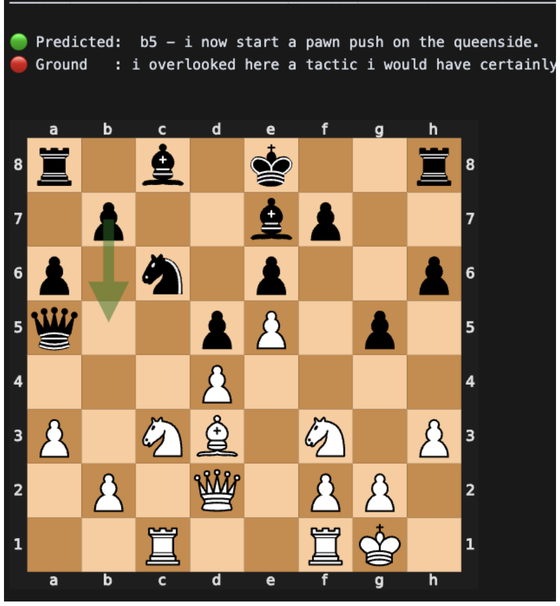
Step 8: Prediction Examples



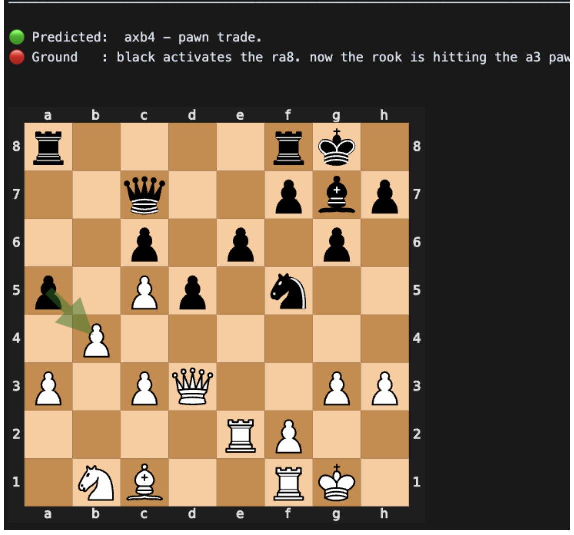
```
Predicted:  cxd4 - pawn trade.
Ground   : i won't have played this one as white. [EOS]
```



```
Predicted:  this is a mistake, as it opens up the e-file for my rook.
Ground   : makes things worse. [EOS]
```



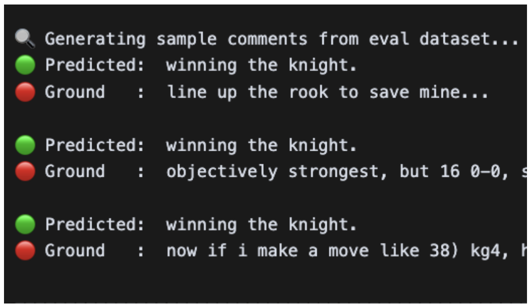
```
Predicted:  b5 - i now start a pawn push on the queenside.
Ground   : i overlooked here a tactic i would have certainly
```



```
Predicted:  axb4 - pawn trade.
Ground   : black activates the ra8. now the rook is hitting the a3 paw
```

Step 9: Critical Takeaways

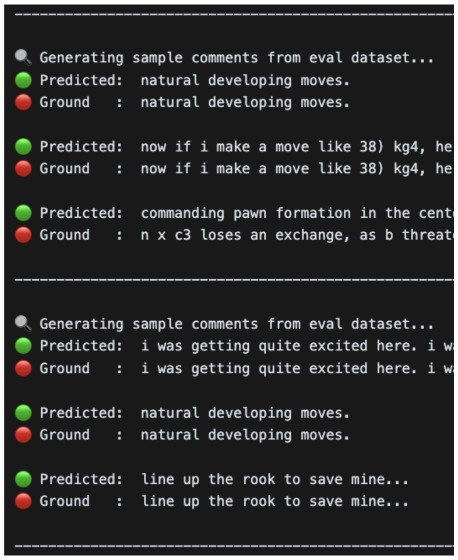
Don't just count on training loss and eval loss to tell you if your model is learning. Transformers can be very deceptive. There were many cases where the training loss approached 0 during my sanity-check training runs - but the model was still producing garbage. This is why it's important to have a mechanism to clearly show the output of the model - during training. Huggingface trainer library has a callback mechanism that allows you to run custom code at certain points in the training process. Here's an example where the network has clearly collapsed, and this visual aid was critical in detecting the issue fast.



```
Generating sample comments from eval dataset...
Predicted:  winning the knight.
Ground    :  line up the rook to save mine...

Predicted:  winning the knight.
Ground    :  objectively strongest, but 16 0-0, s

Predicted:  winning the knight.
Ground    :  now if i make a move like 38) kg4, h
```

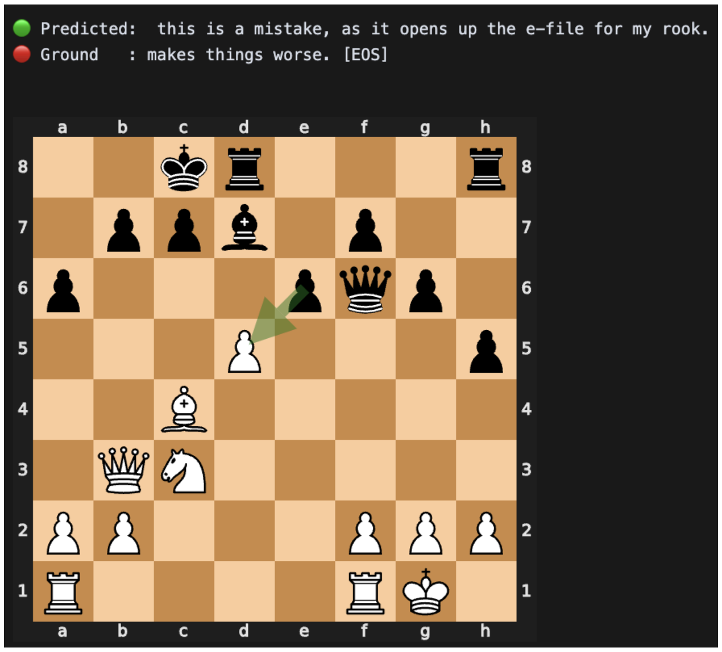
Another takeaway: In the earlier iterations of this project, my transformer would very quickly collapse and keep generating the same output - like "A very good move" - regardless of the input. It's much easier to debug issues like tensor size mismatches - the runtime will error and print the error message and a stack trace. But this is one of those more Byzantine issues that are hard to debug. Training a model is no easy task - especially for a beginner. You need to be patient and keep fixing issues one by one. That's why it's extremely important to have mental checkpoints and sanity checks that gradually become more complex.

Here's a very useful tool: Create a super small training dataset and attempt to overfit it. If your model can't even memorize a dataset of 10 rows, no amount of hyperparameter tuning will help. You have a systemic issue, and likely a bug. This was the case for me. And this technique helped me gradually pinpoint where my issue was. Once I fixed my issues, I was able to see the output here. We're still far from learning, but at least we can memorize.



```
--------------------------------------------------------

🔍 Generating sample comments from eval dataset...
🟢 Predicted:  natural developing moves.
🔴 Ground   :  natural developing moves.

🟢 Predicted:  now if i make a move like 38) kg4, he
🔴 Ground   :  now if i make a move like 38) kg4, he

🟢 Predicted:  commanding pawn formation in the cent
🔴 Ground   :  n x c3 loses an exchange, as b threat

--------------------------------------------------------

🔍 Generating sample comments from eval dataset...
🟢 Predicted:  i was getting quite excited here. i w
🔴 Ground   :  i was getting quite excited here. i w

🟢 Predicted:  natural developing moves.
🔴 Ground   :  natural developing moves.

🟢 Predicted:  line up the rook to save mine...
🔴 Ground   :  line up the rook to save mine...

--------------------------------------------------------
```

Conclusion:
The best performing model was able to produce comments that were both syntactually, but more importantly, semantically correct - on data it has never seen before. So a meaningful amount of learning must have occured. The predictions don't always match the ground truth in terms of vocabulary, formatting and syntax. But they are correct and insightful nevertheless. In fact, they sometimes do a better job than ground truth in explaining the situation. For example the ground truth here only mentions that this move makes things worse, whereas the prediction offers a correct insight as to why it's a bad move.



```
🟢 Predicted:  this is a mistake, as it opens up the e-file for my rook.
🔴 Ground   : makes things worse. [EOS]
```

2 Minute Video: https://youtu.be/fw4TVnBHX3U
15 Minute Video: https://youtu.be/TRrfl2bLbKY