BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
SPECIALIZATION Computer Science

DIPLOMA THESIS

Chess commentary generation using transformers architecture

Supervisor
Lect. Dr. Mircea Ioan-Gabriel

Author
Râpeanu George-Alexandru

2024

UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ

LUCRARE DE LICENŢĂ

Generarea comentariilor de șah folosind arhitectura transformer

Conducător științific
Lect. Dr. Mircea Ioan-Gabriel

Absolvent
Râpeanu George-Alexandru

2024

ABSTRACT

This thesis proposes an Artificial Intelligence(AI) based application, capable of generating commentary for chess games. This commentary aims to both bridge the gap between AI understanding and human reasoning, but also be a learning tool that makes chess more easy to understand.

This thesis starts with a brief introduction in chapter 1 and explores related works in chapter 2. In chapter 3, the dataset utilized is described, and a model is proposed with 8 different variants, all based on the transformer architecture. These variants are about how the model preprocesses the input data, what kind of encoder it uses, and how the loss should be computed. Chapter 4 describes the methods and frameworks used for developing and training the model, and explores some experiments which were done in order to get a more refined model and compares this model to the state of the art. Afterwards, chapter 5 describes the application that showcases the model's capabilities, describing it's architecture, the frameworks used, the testing that was done and how the user can interact with it. Finally, chapter 6 provides a brief conclusion of the paper and lays out potential future work that can be done.

This work is the result of my own activity. I have neither given nor received unauthorized assistance on this work.

Contents

Chapter 1

Introduction

Artificial intelligence (AI) has been a disruptive force in recent years, changing the technological landscape of our day to day life. Almost every tech application or product is enhanced, at least in part, with AI. But among the astounding progress, there is a significant obstacle to overcome: understanding the reasoning underlying AI decision-making.

One of the most influential moments in AI history is the chess showdown between Gary Kasparov and IBM's Deep Blue. For the first time in human history, it has been empirically proved that computers can outperform humans even at tasks that require some level of reasoning, such as the game of chess.

1.1 Motivation

Yet, the game of chess remains an enigma in its own right. With so many moves and possibilities, chess has been known to be one of the hardest games to grasp. The moves chosen by top experts in the field, such as chess grandmasters, can often seem counter-intuitive to average players. With the emergence of chess engines that are orders of magnitude better even than the top humans, the best chess moves can often look puzzling.

Recently, a new wave of AI's have began to be on the rise: the so called "transformers". Large language models(LLMs) such as ChatGPT have become essential AI assistants, capable of often generating reasonable responses to any question one might address them.

Motivated by these developments, this thesis attempts to create a tool that can be used to explain the counter-intuitive moves mentioned above. By using transformers, this thesis hopes to produce a web application capable of explaining the motivation, reasoning, and strategy behind chess moves, and be a good learning assistant for someone approaching the game of chess.

1.2 Objectives

The application aims to be a helpful tool to both beginners who just picked up chess, as well to more moderate players, who want to get some insight into their moves. Furthermore, it aims to offer more insight into how a model things, by attempting to see how well a model might be able to annotate given chess games, and seeing if it demonstrates a good enough logical reasoning, as well as verbal/grammatical coherence.

1.3 Structure

The thesis is structured in 6 main chapters.

- 1. Introduction: This chapter aims to brief the reader on the problem, the objectives and the motivation behind the application.
- 2. Related works: This chapter looks at what has been done in terms of generating chess commentary and what the current state of the art is.
- 3. Methodology: This chapter contains the description for our approach. It first talks about the dataset which was used, how it was recreated and how the data was pre-processed in order to turn it into useable data for the model. Afterwards, the problem is formally defined, and several models for approaching it are described.
- 4. Model development: This chapter talks about the tools used for implementing the model, as well as the various experiments which were done in order to find promising hyperparameters, and in the end it showcases the results of each experiment, when compared with each other. Afterwards, the best model is selected, and is compared to the state of the art models in the field.
- 5. Application development: This chapter talks about the requirements the app that showcases the model should have, what the architecture of it looks like, and provides useful diagrams for each usecase, together with class and sequence diagrams in order to get a better understanding of the inner workings of the application.
- 6. Conclusions: Finally, this chapter provides a conclusion of the paper, and lays out potential ideas where future work can be explored.

Chapter 2

Related works

2.1 State of the art in chess engines

Before discussing about the related work done in commentating games, a brief look into how are computers so good at playing complex games like chess is required. Currently, there are two main types of computer chess engines: traditional ones and neural-network based.

2.1.1 Traditional chess engines

Traditional chess engines work using the same algorithm as IBM's Deepblue[4]: the minmax algorithm[6], often enhanced by various other techniques, such as alpha-beta pruning [14]. This algorithm works by generating all possible moves from a given state, and exploring each one recursively up to some depth, where an evaluation function is used in order to determine the corresponding position's approximate strength. Afterwards, the algorithm chooses for each state the best move according to the current player, and thus it can generate strong moves for any chess position.

Currently, the best traditional chess engine is Stockfish [33] with an estimated ELO rating of 3634, which makes it have approximately $100 : 1$ odds at beating the best human chess grandmasters.

2.1.2 Neural-network based chess engines

In 2017, Google's Deepmind released AlphaZero[23], the first chess engine that was neural-network based, which taught itself how to play chess via trial and error, using reinforcement learning [3]. In a 100 game match against Stockfish, AlphaZero managed to win 28 games, and draw the remaining 72[13].

In order to decide which moves to play, AlphaZero has taught itself to assign values to each move(how strong it is), and the probability that the particular move leads to a win. Then, using Monte-Carlo tree search[37], it simulates some games using the assigned probabilities, in order to obtain even more realistic probabilities.

2.2 Rule based commentary systems

In the beginning, rule based systems were utilized in a variety of games and scenarios, due to their simplicity.

An early example of this is ”Computer generation of Chinese commentary on Othello games”[16]. The idea behind it was to first implement a move generation module, which would use a minmax[6] algorithm together with alpha-beta pruning [14] in order to generate the best moves for a given position. Afterwards, a rule-based commentary generation module was designed to take those moves and generate helpful commentary.

Another paper that approaches annotating games is ”Automated Chess Tutor” [21]. Manually crafted features, such as ”how safe is the king” and ”how central are the knights” were utilized, and differences between them as the game progressed were used to generate commentary. These differences would then be used in order to generate relevant commentary about what happened.

2.3 Neural-network based commentary systems

With the rise of neural networks, they have began to take over the AI field, attempts have been made to generate commentary using neural networks too.

One of the papers which explores this is ”Learning a Game Commentary generator with grounded move expressions” [12]. This paper proposes the idea of ”game trees” with the following motivation: commentators often talk about multiple sequences of moves, exploring variations in order to demonstrate similar ideas, or fully prove that a move is good or bad.

First, using a rule based system and regular expressions, the game tree is constructed from commentary. Afterwards, a multi-layer perceptron is trained to identify which words from the vocabulary will appear in the final commentary. In the end, each position in the output sentence has an associated weight matrix, that is learned, and is used to determine the next word based on the multi-layer perceptron’s output and the previous selected words.

2.3.1 RNN and LSTM based

In more recent times, sequence-to-sequence models have become increasingly complex. One of the popular types of neural networks which could achieve this are the RNN/LSTM based ones[22]. The idea behind them is to have a hidden state for some neurons, and each time an input is passed through them, they update the hidden state and the output using the previous hidden state and the input. Thus, a sequence of outputs can be generated from a sequence of inputs. LSTM's differ from RNN's by allowing the neurons to both "forget" old information, and chose what new information to "commit" to their memory.

These kinds of models have been used in multiple area, including generating commentary based on stock prices[17] and generating game tutorials[9]. However, one of the most influential papers when it comes to chess commentary generation is "Learning to Generate Move-by-Move Commentary for Chess Games from Large-Scale Social Forum Data"[11]. First, this paper proposes a training dataset that is collected from a large chess forum, GameKnot[26]. Then, they noticed that chess commentary falls into 5 different categories, and propose separating them by manually categorizing a few, and afterwards training a SVM classifier for each category. Afterwards, they manually extract features from a chess position, such as what attacks are available, what was the last move, if it was a capture or not, and so on. They then feed these features into a BiLSTM layer, in order to encode them, and decode them with a LSTM decoder, which attempts to predict the next token in the commentary at each step. This paper achieved good results, and it managed to offer some baselines and a standard dataset to be used by other papers which attempt the same task.

One of the papers which makes use of this is "Automated Chess Commentator Powered by Neural Chess Engine"[36]. Based on the previous paper, this one attempts to take everything a step further, by first training a model to play chess, in a similar manner to AlphaZero[23]. Afterwards, multiple sub-models are trained to generate commentary, each one being specialized in one of the identified commentary types from the previous paper.

2.3.2 Transformers

The "Attention is all you need"[34] paper will remain one of the most influential papers in the field of AI. This paper introduces the transformer architecture, which offer a way to reduce the shortcomings of RNN based approaches: as opposed to RNN/LSTM's, transformers are able to process inputs independently, and thus they are more suitable to be parallelized, and thus are able to be trained in a fraction of the time needed to train RNN/LSTM's.

They have managed to prove their potential in many areas, such as creating various chatbots, including ChatGPT[19]. On the commentary generation side of things, while applications of them in chess are limited, they were proven efficient in similar fields, such as commentating basketball games[8].

Chapter 3

Methodology

The problem can be modeled as an auto-regressive task, in which, given some input related to the current state of the game, the model produces commentary, in a token-by-token fashion.

3.1 Dataset

The dataset used is a recreation of the "Learning to Generate Move-by-Move Commentary for Chess Games from Large-Scale Social Forum Data"[11] dataset. Chess commentaries and positions are extracted using the $saved\_files$ links. Afterwards, 500 of them were manually categorized in the categories mentioned by the paper. Afterwards, a sklearn [15] SVM classifier was trained for each category. Table 3.1 contains examples for each.

Category	Example	% in data	val acc.
Direct Move Description	An attack on the queen.	32.4%	96.0%
Move Quality	A rook blunder.	8.8%	93.6%
Comparative	At this stage I figured I better move my knight.	8.0%	98.4%
Planning / Strategy	Trying to force a way eliminate d5 and prevent Bb5.	40.0%	98.8%
Contextual Game Info	Somehow, he game I should have lost turned around in my favor.	35.2%	96.0%
General	Protect Calvin, Hobbs	17.2%	94.0%

Table 3.1: Commentary types

The following descriptions are used, as described in [11]:

- **Direct move description(MoveDesc)**: Explicitly or implicitly describe the current move.
- **Quality of move(MoveQuality)**: Describe the quality of the current move.
- **Comparative**: Compare multiple possible moves.
- **Planning / Strategy**: Describe the rationale for the current move, in terms of gameplay, advantage, over other potential moves.
- **Contextual Game Information**: Describe not the current move alone, but the overall game state - such as possibility of win/loss, overall aggression/defence, etc.
- **General information**: General idioms and advice about chess, information about players/tournament, emotional remarks, retorts, etc.

3.1.1 Dataset description

The dataset D can be described as a set of games $G_i, i \in \{1, 2, \ldots, |D|\}$. Each game can be described as a pair of sequences, $G_i = (B_{i,j}, C_{i,j}), i \in \{1, 2, \ldots, |D|\}, j \in \{1, 2, \ldots, |B_i|\}$, where $B_{i,j}$ is the board state of game i at move j, and $C_{i,j}$ is the associated commentary of game i at move j. The commentary should be in a tokenized form, being represented as an array of tokens.

The model receives, as input, the last board states up to the current move, and the generated commentary so far, and should produce the next token of the commentary. More formally, the model should approximate the probability function $P(C_{i,curr,j}|B_{i,max(curr-count,1)}, \ldots, B_{i,curr}, C_{i,curr,1}, \ldots, C_{i,curr,j-1})$, where i is the current game, $curr$ is the current board state, $count$ is the number of past boards that are passed to the model, and j is the current token to predict.

3.2 Model

All models proposed in this thesis have a common feature: they consist of a model encoder, and a transformer model decoder. The decoder is modeled after the "Attention is all you need"[34] paper.

3.2.1 Input features

The input consists of the last $past\_boards$ chess boards, where $past\_boards$ is a model hyperparameter. If there are not enough past boards available(for example, if its the start of the game), then the features of the non-existent boards are filled with zeros.

Afterwards, we experimented with 2 ways of transforming the boards into input tensors:

Alphazero representation

Inspired by the success of Alphazero [23], an augmented version of the input features of AlphazeroChess was used, with the exception of using only the past 2 boards, instead of the past 8. The input is comprised of multiple 8×8 planes, each corresponding to different features.

There are two types of features: "Position features", which refer to only one chess position, and "State features", which refer to data which is not necessarily related to each individual position, such as the side to move. Tables 3.2 and 3.3 contain the description for each.

Feature name	Feature planes	Feature description.
P1 piece	6	For each piece type, an 8×8 plane is created, containing 1 in each cell where the corresponding white piece type is present, and 0 everywhere else.
P2 piece	6	For each piece type, an 8×8 plane is created, containing 1 in each cell where the corresponding black piece type is present, and 0 everywhere else.
Checks	1	An 8×8 plane is created, containing 1 in each cell that contains a piece that is checking the current player's king, and 0 everywhere else.
Strength	1	An 8×8 plane is created, containing in each cell the current engine evaluation, expressed in centipawns. If there exists a forced checkmate, a hyperparameter defined as $mate\_value$ is used, with the sign corresponding to the player which has the forced checkmate sequence.
Repetitions	1	An 8×8 plane is created, containing the number of times the current position has been reached before(including the current position).
Total	15	

Table 3.2: Position features

For each supplied board, its "position features" will be extracted. Afterwards, the "state features" are extracted. All of those are concatenated and flattened, re-

Feature name	Feature planes	Feature description.
Color	1	An 8×8 plane, containing 0 everywhere if it is white's turn to play, otherwise 1.
Total moves	1	An 8×8 plane, containing in each cell the number of the last played chess move.
Castling	4	8×8 planes, each containing in each cell 1 if its corresponding player can castle kingside/queenside.
No progress count	1	An 8×8 plane, containing the number moves played with no progress in each cell, as it is defined in the fifty moves rule.
Total	7	

Table 3.3: State features

sulting in a $Batch \times T \times dim_{input}$ tensors. With $count\_past\_boards = 2$, $dim_{input} = 15 \times (2+1) + 7 = 52$

In order to be transformed to $Batch \times T \times dim_{encoder}$ tensors, the model uses a 1×1 $Conv2d$ layer, followed by $BatchNorm2d$ and $ReLU$.

$X_{input} = RElU(BatchNorm2d(Conv2d(X_{board})))$

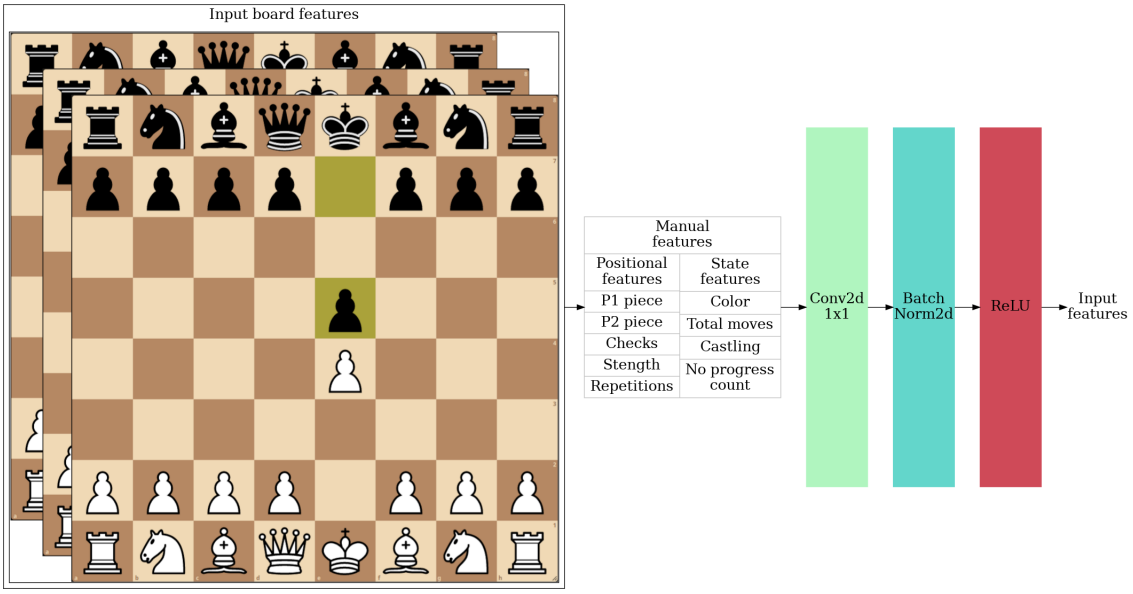


Figure 3.1: Alphazero input features

Actual Board representation

Inspired by the way in which transformers manage to learn embeddings for words, we decided to try delegating the transformation to $Batch \times T \times dim_{encoder}$ tensors to

the model. Thus, we feed the following into the model:

- Color-Piece array: A 64-length array, which contains on each position values from 0 to 12, each one corresponding to a unique piece and color, plus an additional value indicating that no piece is present.
- Strength tensor: A tensor consisting of $past\_boards + 1$ values, each being the strength of its corresponding board, as defined in the previous approach.
- Repetitions tensor: A tensor consisting of $past\_boards+1$ values, each being the repetitions of its corresponding board, as defined in the previous approach.
- State tensor: A tensor consisting of "state features", as defined in the previous approach.

In order to transform these features into a similar $Batch \times T \times dim_{input}$ tensor, the model has to first preprocess them with learnable embeddings:

- Color-Piece embedding($piece\_embedding$): a learnable embedding layer, containing 13 learnable tensor of $dim_{encoder}$ shape.
- Cell positional embedding($pe\_cell\_embedding$): A learnable positional embedding layer, that operates on the flattened boards.
- Board positional embedding($pe\_board\_embedding$): A learnable positional embedding layer, containing $past\_boards + 1$ positions, corresponding to each board that is fed into the model. For correct tensor broadcasting, it will have the shape of $(count\_boards + 1) \times 1 \times dim_{encoder}$
- Strength linear layer($strength\_linear$): a $1 \times dim_{encoder}$ linear layer for strength features.
- Repetitions linear layer($reps\_linear$): a $1 \times dim_{encoder}$ linear layer for repetitions features.
- State feed-forward network($state\_ffn$): a Multilayer perceptron, that consists of:
  - $7 \times dim_{encoder}$ linear layer, with ReLU activation function.
  - $dim_{encoder} \times dim_{encoder}$ linear layer, with no activation function.

Assuming that the batch dimension is present, the layers are applied in the following way:

$X_{embedded\_boards} = piece\_embedding(X_{boards}) + pe\_cell\_embedding + pe\_board\_embedding$

$X_{position\_features} = unsqueeze(concat(strength\_linear(X_{strength}), reps\_linear(X_{reps})), 2)$

$X_{state\_features} = state\_ffn(X_{state})$

$$X_{input} = concat(flatten\_middle(concat(X_{embedded\_boards}, X_{position\_features})), X_{state\_features})$$

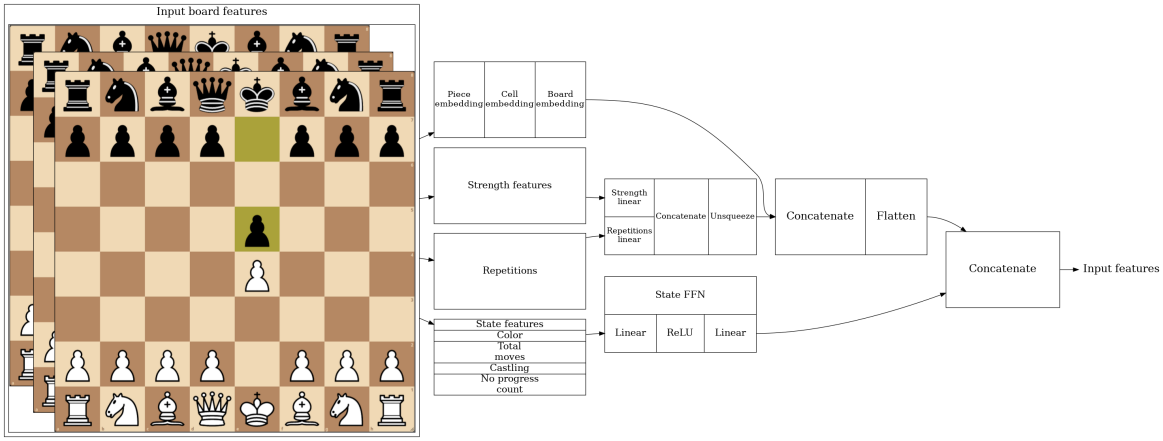


Figure 3.2: Actual board input features

3.2.2 Commentary tokenization

Tokenization of the commentary was done using sentencepiece[27], an unsupervised text tokenizer and detokenizer developed at Google. The tokenizer was trained on the train split of the commentaries from the dataset. A $< bos >$(beginning of sequence) token is prepended to the commentary, and a $< eos >$(end of sequence) token is appended to the tokens. Additionally, the input may be padded towards the end in order to batch inputs. This utilizes an additional $< pad >$ token.

Afterwards, a positional encoding is added to each token, as mentioned in "Attention is all you need"[34].

3.2.3 Model Encoder

Both input feature extraction methods offer $Batch \times T \times dim_{encoder}$ tensors. The encoder consists of multiple $EncoderBlocks$, stacked on top of each other. There are 2 types of $EncoderBlocks$ which were experimented with:

- Resnet encoder block[10]: inspired by Alphazero Chess, $ResnetEncoder$ blocks were used in order to learn about the input. Figure 3.3 illustrates their architecture(without the BatchNorm2D layers).

  $ResnetEncoderBlock(X) =$
  $ReLU(X + BatchNorm2D(Conv2D(ReLU(BatchNorm2D(Conv2D(X))))))$

- Transformer encoder block: the classic $TransformerEcnoder$ block, specified in [23]. Figure 3.4 illustrates its architecture.

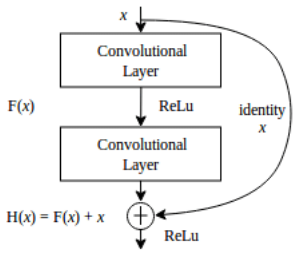


Figure 3.3: Resnet block [18].

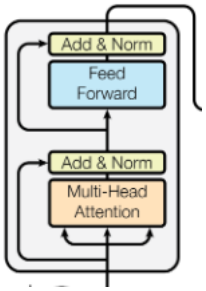


Figure 3.4: Transformer encoder block[34].

3.2.4 Model Decoder

The decoder is the classical transformer decoder. It consists of multiple $Transformer$-

$DecoderBlocks$. Figure 3.5 illustrates their architecture.

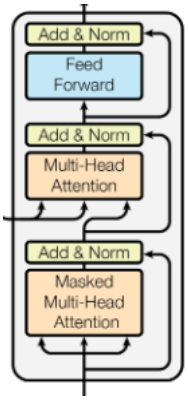


Figure 3.5: Transformer decoder block[34].

$XText_{i+1} = TransformerDecoder(X_{i+1}, XText_i)$. In the end, the last decoder layer's outputs are fed through a linear layer, and the output logits are
$Logits = Linear(XText_{decoder\_blocks})$

3.2.5 Model overall architecture

First, board features are extracting using either the $Alphazero\ representation$ or $Actual\ board\ representation$. A positional embedding is added to the commentary up to the current token, and fed as initial input to the decoder of the model. Let X_{boards} be the features received from the boards, and X_{text} be the features received from the commentary.

The model will consist of an equal number of $EncoderBlocks$ and $DecoderBlocks$, each being a variant of the ones specified above. Let $EncoderBlock_i$ and $Decoder - Block_i$ be the encoder and decoder blocks from the i'th level. Let num_blocks be the number of total blocks. Let $X_{encoder,i}$ be the input of the $EncoderBlock$ on level i, and $X_{decoder,i}$ be the input that comes from the previous block of the $DecoderBlock$ from level i. Then:

- $X_{encoder,0} = X_{boards}$
- $X_{decoder,0} = X_{text}$
- $X_{encoder,i} = EncoderBlock(X_{encoder,i-1}), i \in \overline{1, \ldots, num\_blocks}$
- $X_{decoder,i} = DecoderBlock(X_{decoder,i-1}, X_{encoder,i}), i \in \overline{1, \ldots, num\_blocks}$

Last, the final $DecoderBlock$'s output is fed through a $LinearLayer$, in order to get logits.

$logits = linear(X_{decoder,num\_blocks})$

Figure 3.6 illustrates this architecture.

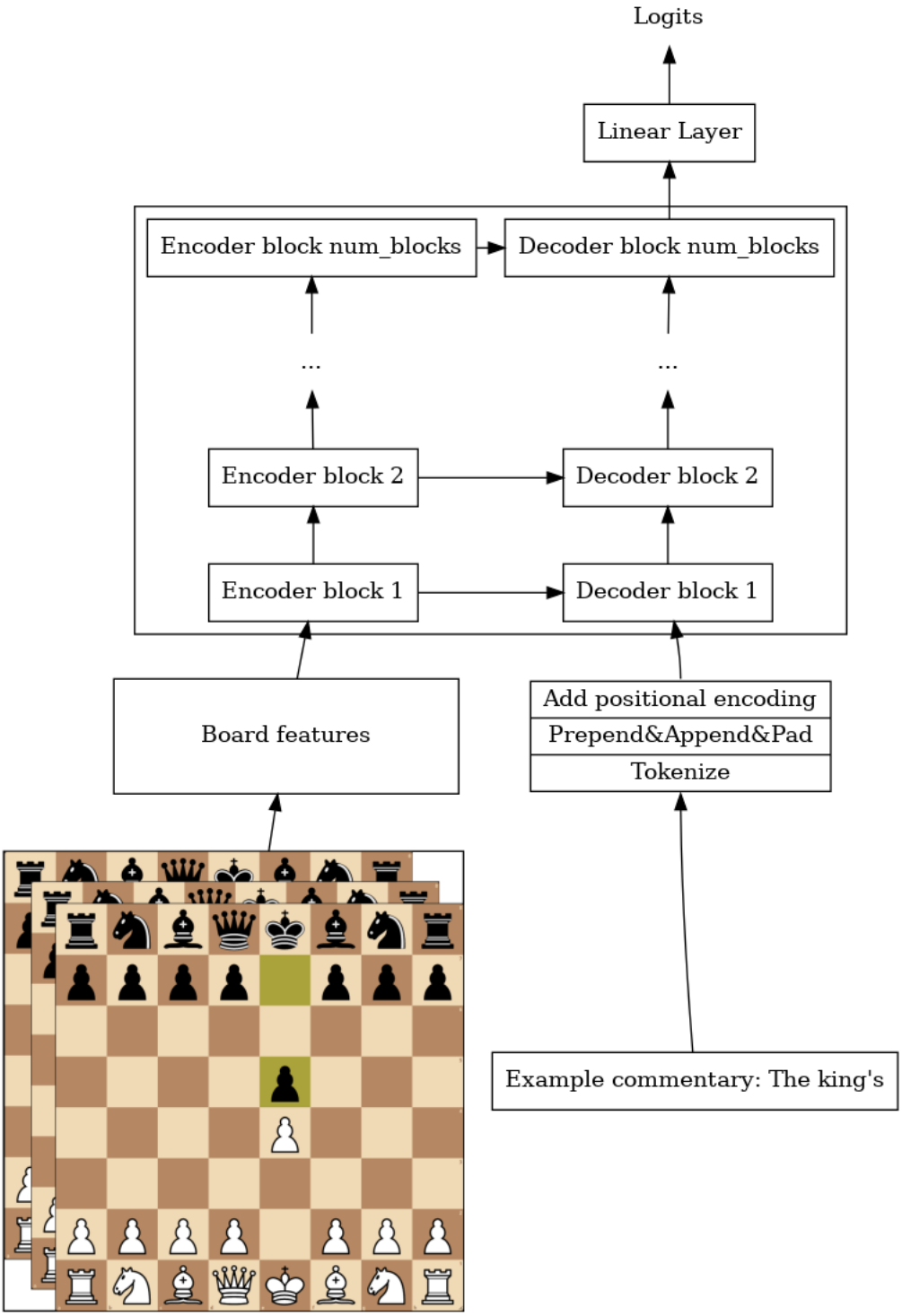


Figure 3.6: Model

3.2.6 Computing model loss

There are two ways of calculating the loss function, the second one seems to provide quicker convergence, but does not improve the accuracy:

- $Loss = CrossEntropy(logits, y)$
- Multiple heads: For each type of commentary, a linear layer is configured to take the output from the decoder block at a specific level. This linear layer should also provide the logits for all inputs which are of that type. Let $Linear-Type_i$ be the corresponding linear layer, and $DecoderType_i$ be the output of decoder block responsible for the current type. In this case, the loss becomes $Loss = CrossEntropy(logits, y)+Crossentropy(LinearType_i(DecoderType_i), y)$, where $type(y) = i$. Figure 3.7 illustrates this.

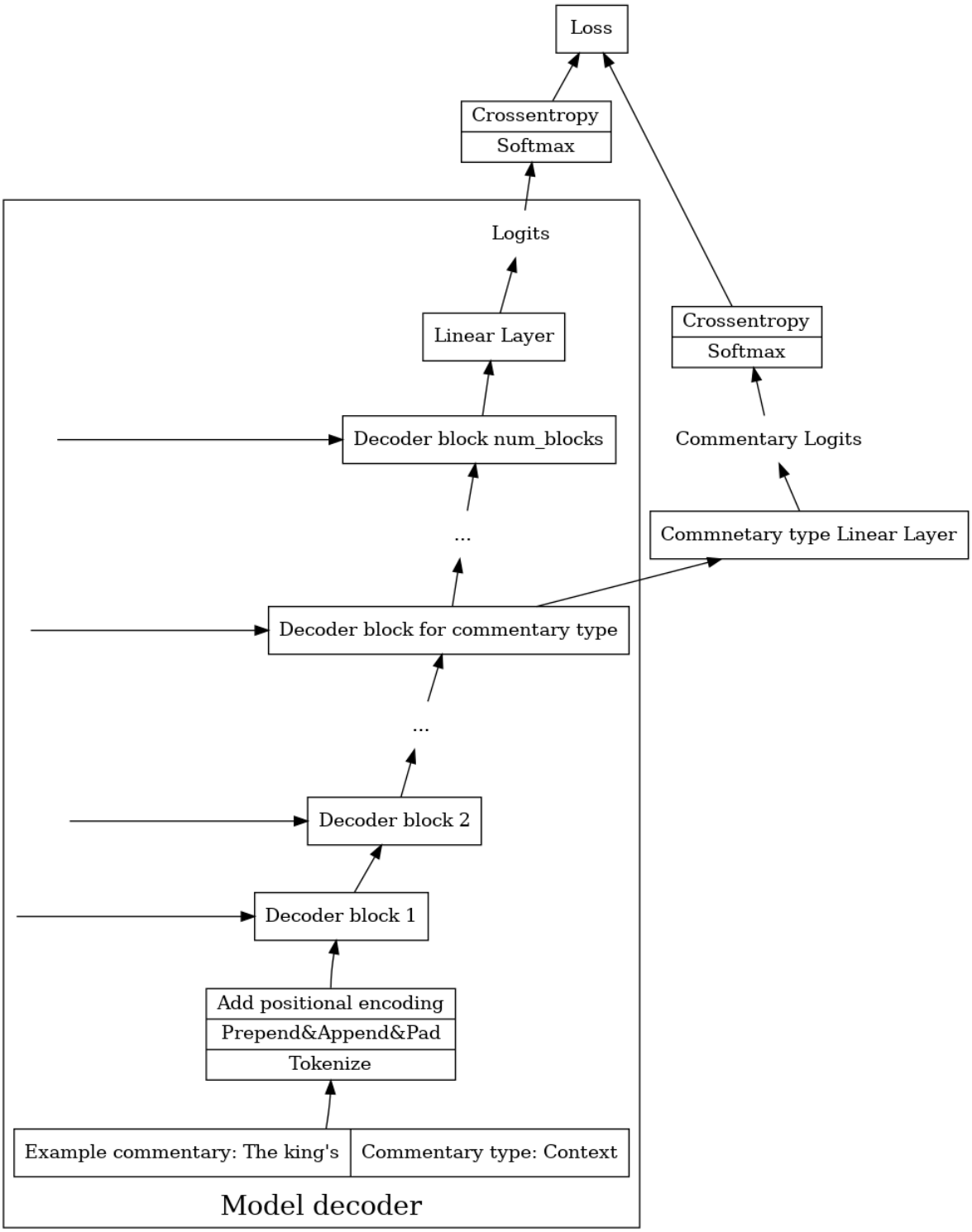


Figure 3.7: Multiple heads loss

Chapter 4

Model development

4.1 Implementation

The model was implemented using PyTorch Lightning[32], a flavour of the PyTorch[31] python library. PyTorch Lightning builds on top of what PyTorch offers, providing a more standardized experience. A few of the advantages that the library provides over vanilla PyTorch are:

- Modular design: a standardized, modular approach to model design, training, and evaluation, offered through the *LightningModule* base class.
- Code reduction: the library handles a lot of common deep learning tasks, such as making the code GPU/TPU compatible, model checkpointing and distributed training.
- Flexibility: despite its abstractions, PyTorch Lightning retains the flexibility and control of native PyTorch, the user being able to mix vanilla and lightning code as they see fit.

The Hydra[29] framework was also utilized in order to provide flexibility when it comes to configuring the model and training processes. Being a configuration management framework, hydra offers a composition-based approach. This is useful for training models, being able to experiment with model settings and hyperparameters by specifying different configuration files.

4.2 Weight and Biases

Weights and Biases(Wandb[35]) is a machine learning platform for developers to build better models faster. Some features which it provides to developers include:

- Artifacts: version assets, which can include model weight files, datasets, configs, and so on
- Tables and Reports: provide a way to visualise the model's performance as it is training, together with statistics such as correlations between hyperparamters and final model loss/accuracy
- Sweeps: A tool which allows searching for optimal hyperparameters for the model

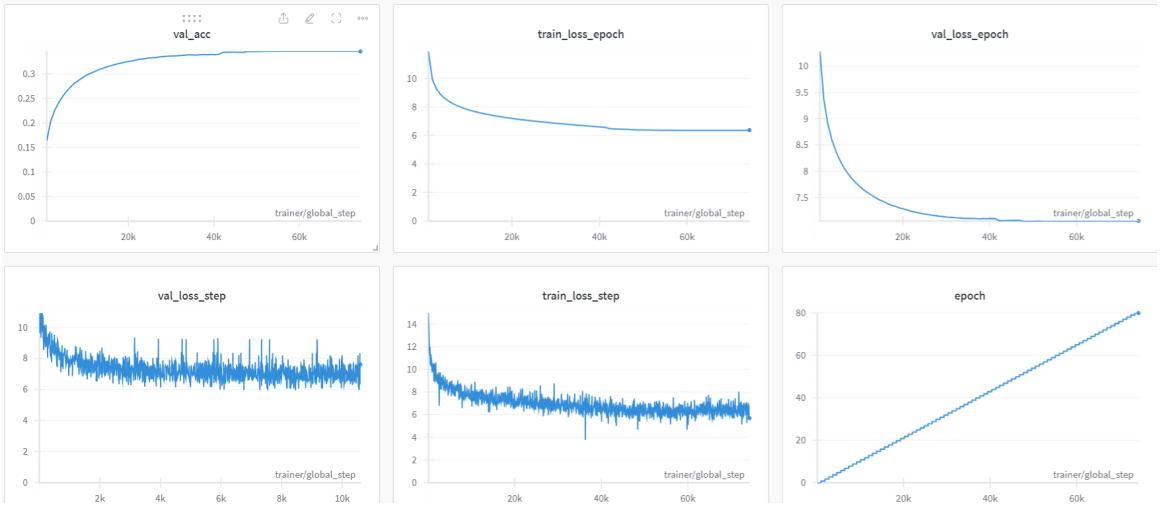


Figure 4.1: Wandb reports

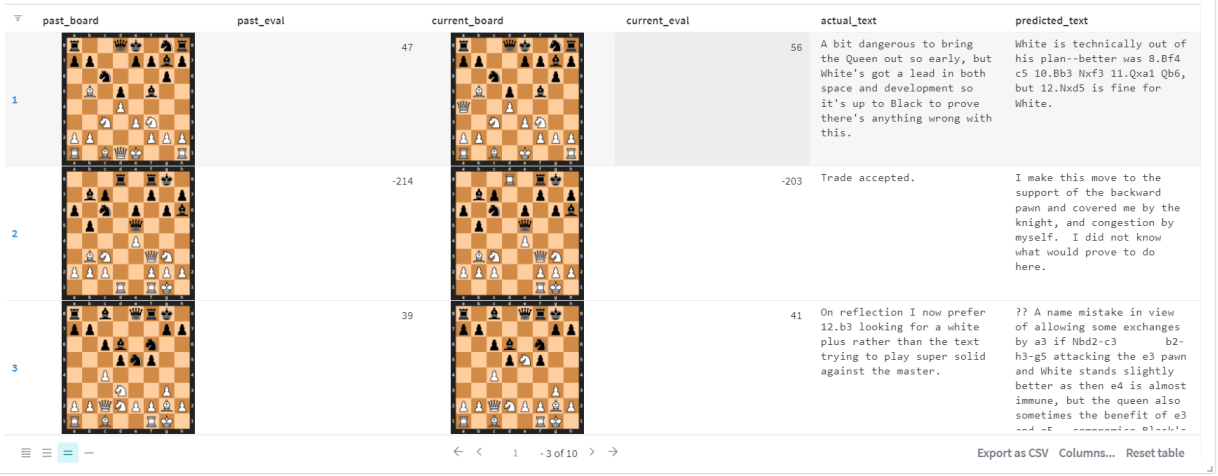


Figure 4.2: Wandb tables

Wandb was utilized in order to keep track of the model's various versions and to see the model's metrics as it is training. Comparing different versions of the model is also made easy by Wandb. An initial wandb sweep was run in order to narrow down the optimal hyperparameters(figure 4.4).

From it, the following values have been selected for further exploration:

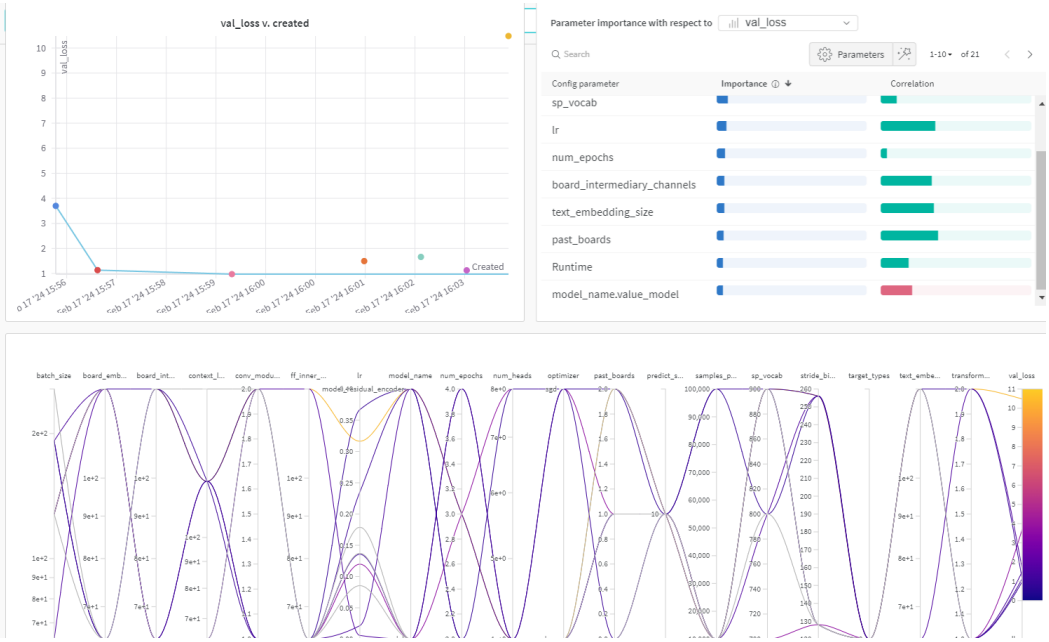


Figure 4.3: Wandb sweeps

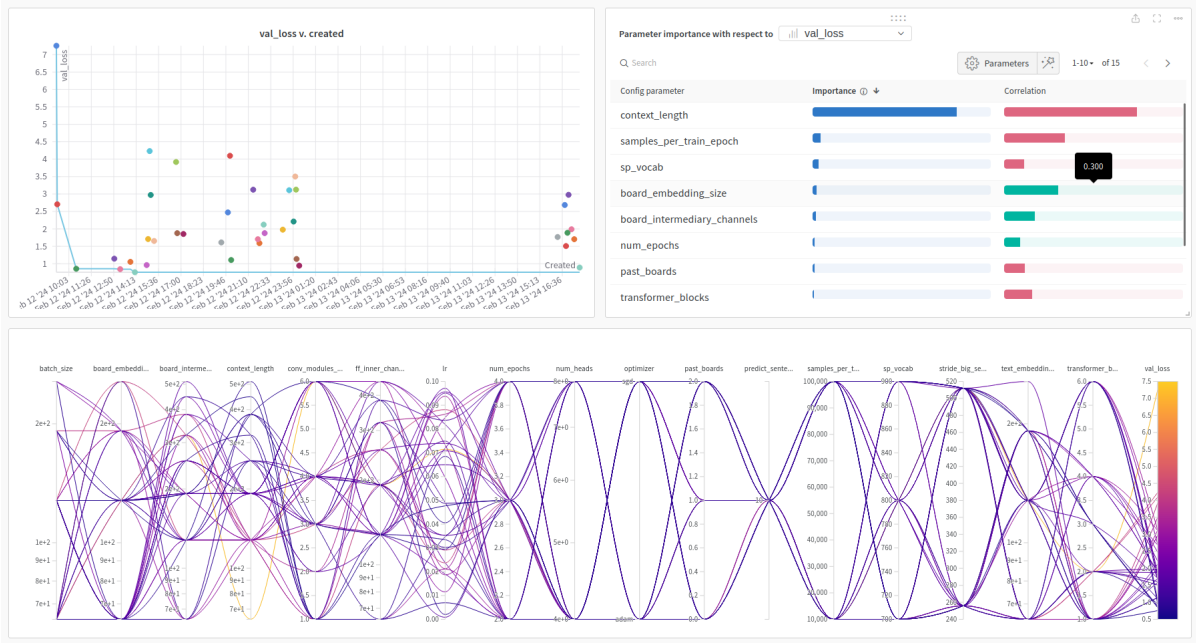


Figure 4.4: Wandb initial sweep

- $context\_length = 512$: The number of maximum tokens that the model receives as far as previous commentary tokens are concerned.
- $sp\_vocab \in \{1000, 2000\}$: The number of tokens available in the vocabulary.
- $num\_blocks(transformer\_blocks) = 6$: The number of encoder-decoder blocks.
- $dim\_encoder(board\_embedding\_size) = 256$: The dimensionality of the encoder features.
- $ff\_inner\_channels = 512$: the inner channels of each $FeedForward$ network used in the transformers.
- For sgd, a learning rate of 0.08 was found to be best. For $adam$, a learning rate of 0.0001 was found to be best.

A couple of model variation were run until convergence, with $EarlyStopping$ and $ReduceOnPlateau$. Table 4.1 illustrate their results.

	Input features	Encoder type	Loss type	sp_vocab	optim	val_loss	val_acc
1	Alphazero	Resnet	multiple heads	2000	sgd	7.255	33.06%
2	Alphazero	Resnet	multiple heads	1000	sgd	2.778	37.77%
3	Actual board	Transformer	normal	2000	sgd	3.171	33.73%
4	Actual board	Transformer	normal	2000	adam	3.263	32.27%
5	Actual board	Transformer	normal	1000	adam	2.672	39.98%
6	Alphazero	Resnet	multiple heads	1000	adam	5.903	40.91%
7	Actual board	Transformer	multiple heads	1000	adam	5.942	40.64%
8	Actual board	Transformer	multiple heads	2000	adam	7.054	34.61%

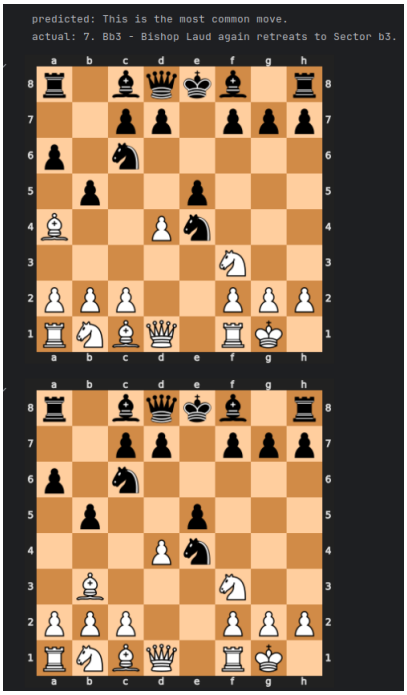
Table 4.1: Model experiments results

From a first look at the table, it seems that the models with $multiple\ heads$ loss type perform worse in terms of loss. However, this is misleading, since they use a different loss formula. The accuracy is a much better indicator of their performance.

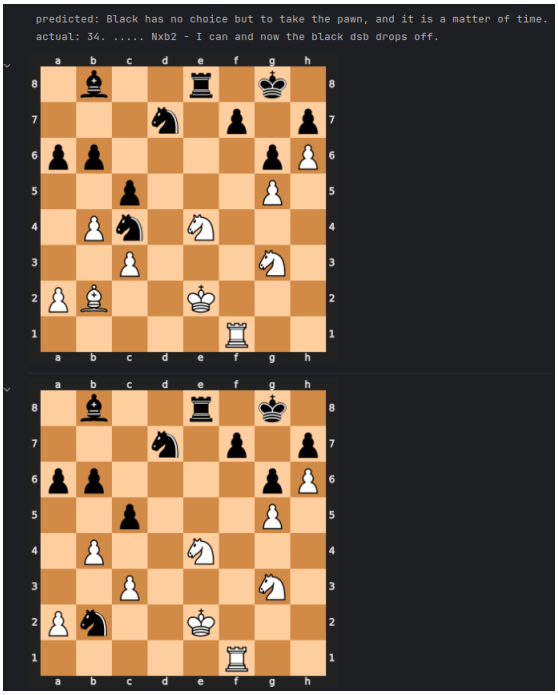
There seems to be two clear candidates for the best model: model 6(the best accuracy from the models with $sp_{vocab} = 1000$), and model 8(the best accuracy for the models with $sp_{vocab} = 2000$).

Even though model 8 has a much lower accuracy than model 6, it works with twice as many tokens, and overall was found to perform better than model 6, producing commentaries which sounded more human, and which made more sense.

Thus, we decided to proceed with this model. Figure 4.5 and 4.6 show some examples of what the model is able to generate, together with what the actual commentary was for the example. The first board in all of them is the previous chess position, and the second one is the chess board after the move.4.5a shows the model incorporating some degree of domain knowledge, by commentating about the move's popularity. 4.5b shows the model commentating about the last move, even though it mistakes the bishop for a pawn. Figure 4.6 shows the model commentating on a positional advantage that black has gained, known as a "passed pawn".



(a) Commentary about the move's popularity.



(b) Commentary about the move.

Figure 4.5: Commentary examples.

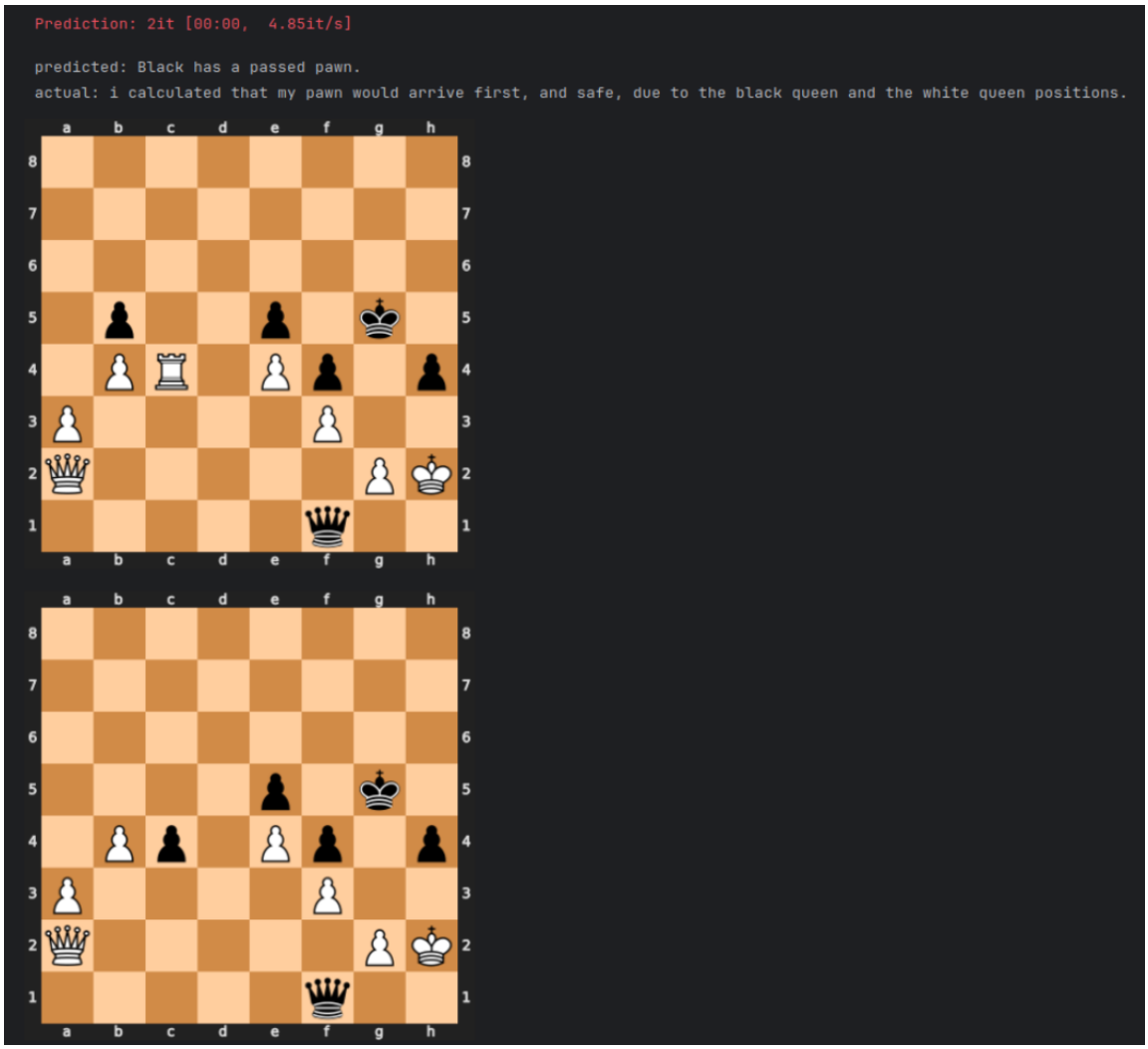


Figure 4.6: Commentary about the position after the move.

4.3 Results

With this model, we generate commentaries for each of the categories established in 3.1. The commentaries are generated both by sampling according to the distribution predicted by the model and by taking the token with the highest probability.

As suggested in [36], the evaluation metrics used are *BLEU-2* and *BLEU-4* [20], along with *METEOR*[2]. The models from [36] are used as baseline for this model. The results are illustrated in the table 4.2 as *model-topk* for highest probability token sampling and *model-sample* for sampling with respect to the predicted probabilities.

BLEU-4(%)	**Temp**	**Re**	**KWG**	**GAC**	**SCC-weak**	**SCC-strong**	**SCC-mult**	**model-topk**	**model-sample**
Description	0.82	1.24	1.22	**1.42**	1.23	1.31	1.34	0.09	0.42
Quality	13.71	4.91	13.62	16.90	16.83	18.87	**20.06**	4.17	0.00
Comparison	0.11	1.03	1.07	1.37	2.33	3.05	2.53	**5.47**	3.52
Planning	0.05	0.57	0.84	N/A	**1.07**	0.99	0.90	0.66	0.53
Contexts	1.94	2.70	4.39	N/A	4.04	**6.21**	4.09	0.23	0.59
BLEU-2(%)	**Temp**	**Re**	**KWG**	**GAC**	**SCC-weak**	**SCC-strong**	**SCC-mult**	**model-topk**	**model-sample**
Description	24.42	22.11	18.69	19.46	23.29	**25.98**	25.87	0.33	3.17
Quality	46.29	39.14	55.13	47.80	58.53	61.13	**61.62**	9.90	6.46
Comparison	7.33	22.58	20.06	24.89	24.85	**27.48**	23.47	11.78	10.46
Planning	3.38	20.34	22.02	N/A	22.28	**25.82**	24.32	3.78	3.73
Contexts	26.03	30.12	31.58	N/A	37.32	**41.59**	38.59	1.41	3.58
METEOR(%)	**Temp**	**Re**	**KWG**	**GAC**	**SCC-weak**	**SCC-strong**	**SCC-mult**	**model-topk**	**model-sample**
Description	6.26	5.27	6.07	6.19	6.03	6.83	7.10	6.45	**8.51**
Quality	22.95	17.01	22.86	24.20	24.89	25.57	25.37	**35.82**	21.30
Comparison	4.27	8.00	7.70	8.54	8.25	9.44	9.13	**18.55**	16.55
Planning	3.05	6.00	6.76	N/A	6.18	7.14	7.30	9.34	**10.00**
Contexts	9.46	8.90	10.31	N/A	11.07	**11.76**	11.09	7.48	8.81

Table 4.2: Model scores. Based on [36].

When commentating without giving the model a specific category, the *COMB*, *GAC* and *CAT* models from [11] are used as a baseline. Only the *BLEU-2* and *BLEU-4* are available for these models. Table 4.3 shows the results in this scenraio.

Model	BLEU-4	BLEU-2	METEOR
COMB (M)	2.07	20.13	N/A
COMB (M + T)	**2.43**	25.37	N/A
COMB (M + T + S)	1.83	**28.86**	N/A
GAC-all (M)	1.69	20.66	N/A
GAC-all (M + T)	1.94	24.11	N/A
GAC-all (M + T + S)	2.02	24.70	N/A
CAT (M)	1.90	19.96	N/A
model-topk	0.54	2.85	8.03
model-sample	0.53	3.42	**8.80**

Table 4.3: Model scores without specific category. Based on [11].

It can be seen that our model outperforms the established state of the art in some categories, but is worse in the majority of them. It also has a higher *METEOR* score, which indicates that the model generates text with more similar meaning to the original, but deviates from it when evaluating n-grams, like *BLEU* does. When generating commentary without a specified commentary type, the model generates slightly worse commentaries in the majority of cases, which is expected.

Chapter 5

Application development

The application aims to augment the model, allowing the user to explore it's capabilities in a straight forward manner.

The main objective of the application is to allow the user to fully grasp the capabilities of the model. For this, the user should be able to both explore commentaries generated by the model for any chess game, as well as see concrete probabilities if he wishes to.

A secondary objective, yet as significant as the main one, is for the application to be intuitive to use and allow the user to tinker with the model without fully understanding neither chess nor how the model works.

5.1 Domain specific objects

The web application should allow the user to explore chess games and positions in a straight forward way. Thus, the following chess "objects" will be defined:

- Piece: a chess piece, such as a pawn, rook, knight, bishop, king or queen; in each color variant(black and white)
- Move: a valid chess move, containing a starting square and a target square, and optionally a promotion
- Game: A sequence of valid moves. Each chess game has an equivalent PGN (Portable Game Notation [7])
- Position: A chess position refers to the state of the game/board at a given move.

5.2 Functional requirements

The following functional requirements have been designed in order to demonstrate the model's ability:

- (F1): The application should provide a board explorer feature, in which the user can play and undo valid chess moves, such that he is able to explore any chess game possible.
- (F2): The user should be able to seek a game's chess position at any given move.
- (F3): The application should provide an "import game feature", where the user should supply a valid PGN, and the application should load the corresponding chess game in the game explorer.
- (F4): The user should be able to request commentary from the model.
- (F5): A menu should be available for the user, from which he should be able to change the settings with which the model operates. The initial values for these and other metadata should be initialized at the start by querying the backend. In case this fails, a retry connection should be presented to the user. The following parameters should be available in the menu:
  - $temperature$: A strictly positive float number, which is used for sampling. Increasing this number should produce more diverse results, but they will also be more inaccurate.
  - do_sample: A boolean value, which determines the sampling mode. If set to true, the model will sample each token given the probability distribution computed by it. Otherwise, it will take the token with the highest probability.
  - $target\_type$: A string which can take the following categories: $MoveDesc$, $MoveQuality$, $Comparative$, $Strategy$, $Context$. When present, this model will produce commentary that is of the requested type. A $General$ option should also be available, in which case the model does not have to generate commentary tailored to a specific category.
  - max_new_tokens: An integer which determines the maximum number of tokens the model should produce at a given time.
  - $prefix$: A string which can be used in order to utilize the model in autocomplete mode.

- (F6): The user should be able to see, for a given chess position together with the settings mentioned above, what the model suggests for the next token, together with the probability for it.
- (F7): The user should be able to see the strength of the current position.

5.3 Use case diagram

Figure 5.1 represents the features as usecases.

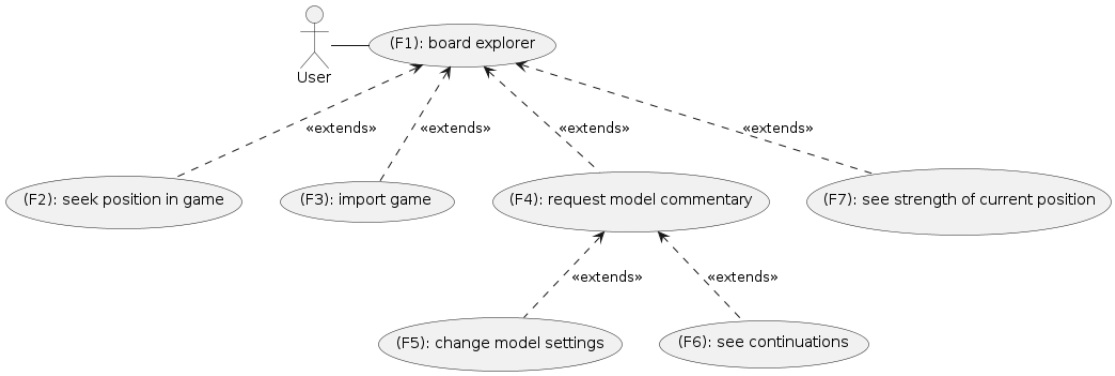


Figure 5.1: Use case diagram

5.3.1 Usecases description

F1

- **Description**: A chessboard should be available for the user to explore various chess positions. The user should be able to move pieces(by clicking on them and their target square or via drag and drop).
- **Primary actor**: User
- **Preconditions**: The board is in a valid state.
- **Postconditions**: The board allows the changes if and only if they also provide a valid state.
- **Flow**:
  - The user clicks on a piece he wants to move
  - The user clicks on one of the available target squares, which provide legal moves.

  - If the user click on something that is not a valid move, the move is canceled.
  - The board executes the move. In case the move is a promotion, an extra dialogue/screen is used to ask for the piece the user wants to promote to.

F2

- **Description**: The user should be able to see the game's moves up to this point, and select any of them to view the game at that point in time. Also, he should be able to undo and redo moves.
- **Primary actor**: User
- **Preconditions**: The board is in a valid state.
- **Postconditions**: The board allows the changes if and only if they also provide a valid state.
- **Flow**:
  - The user clicks on a the last move of the position he wants to seek.
  - The app shows the chess position at that move. The user should be able to redo and undo moves starting from this point.
  - The user can, at any point, use the left arrow to undo the last move, and the right arrow to redo moves. Using the up and down arrows seeks to the beginning of the game, and to the end respectively.

F3

- **Description**: Editable fields should be provided, which display the current game's FEN(Forsyth-Edwards Notation [5]) PGN(portable game notation [7]). The user should also be able to edit these fields and provide a valid FEN or PGN to load other games.
- **Primary actor**: User
- **Preconditions**: The FEN / PGN provided by the user is valid.
- **Postconditions**: The game is changed to reflect the game provided by the user.
- **Flow**:
  - The user replaces the string from the textarea with his new valid FEN / PGN.

  - The app validates it, and loads the game with the corresponding FEN / PGN.
  - If the FEN / PGN is invalid, an error is displayed.

F4

- **Description**: The user should be able to request commentary from the backend model using a button.
- **Primary actor**: User
- **Preconditions**: The game is in a valid state.
- **Postconditions**: The commentary appears in a commentary textarea.
- **Flow**:
  - The user presses a button to request commentary from the model.
  - A request is sent to the backend which interacts with the model.
  - A stream response is sent back to the client.
  - The commentary is placed into the textarea, as it is being streamed.
  - In case any network exception occurs, an error is properly displayed to the user.

F5

- **Description**: The user should be able change settings for the commentary generated by the model.
- **Primary actor**: User
- **Preconditions**: The settings are valid.
- **Postconditions**: The new settings remain valid.
- **Flow**: If the backend was successfully queried for initialization:
  - The user changes any of the settings.
  - The app validates them, and updates them for future requests.

  If the backend was not successfully queried for initialization:
  - The user is presented with an error message and a retry button.
  - The user can retry the initialization phase.

F6

- **Description**: The user should be able to see the top 10 suggestions for continuing a prompt, and be able to select one of the in order to explore further.
- **Primary actor**: User
- **Preconditions**: The prompt is valid.
- **Postconditions**: The prompt remains valid.
- **Flow**:
  - The application sends a request to the backend on any game/model settings change.
  - The user sees possible continuations for a given prompt.
  - The user can select one of them. The application updates the current prompt and appends the given continuation.
  - In case a network error occurs, the request is retried at most 3 times. If this also fails, an error is dispalyed, together with a retry button.

F7

- **Description**: The user should be able to see a score for the given position, evaluated by an established chess engine.
- **Primary actor**: User
- **Preconditions**: The game is in a valid state. An engine chess web worker has been started successfully.
- **Postconditions**: None
- **Flow**:
  - The user interacts with the app in a way that changes the current chess position.
  - The application cancels the previous position's evaluation requests, and asks the engine web worker to evaluate the new position.
  - The responses from the web worker are streamed and placed by the app in the frontend.

5.4 Application design

The core of the application should be a minimal web API that exposes a way for the user to utilize the model. This core should be responsible for only one endpoint:

- $POST\ /get\_commentary\_execution$, which takes as parameters the current position, together with the past 2 positions, and the model settings described in F5 5.3.1. It should return a binary stream HTTP response, containing the "commentary execution", which is defined as such: at each step, the model samples the next token based on the model settings and the text generated so far, and returns the generated logits at the step, together with the selected token.

In front of this core, a proxy web application should be placed, responsible for caching the commentary executions. On top of this, it should also expose the following endpoints:

- $POST\ /get\_commentary$, which takes as parameters the same things as the model does, and returns a stream of strings, that is decoded from the information given by the model. The information is also cached.
- $POST\ /topk$, which is a special case of the previous endpoint. It takes as parameters the same things as the previous endpoint, except do_sample(since sampling is not needed for this feature), and injects $max\_new\_tokens = 1$. The information is also cached.
- $GET\ /info$, which takes no parameters, and returns general model settings and hyperparamters, such as the number of past boards it required, recommended initial values for the model settings, and ranges for some of them.

A frontend web application should make use of these endpoints, and should implement an interface for the functional requirements 5.2.

5.4.1 Model(core)

Flask[25] is a light web framework designed for python applications. Since the model was developed in python, it integrates well with flask. The model application exposes the $/get\_commentary\_execution$ endpoint, which returns a streamed response of binary data, containing the logits from the model output, together with the sampled one. Logits are returned instead of probabilities since it is easier to modify the logits on temperature changes requests. Listing 5.1 shows the code snippet responsible for creating such stream.

```
with torch.no_grad():
    for i in range(max_new_tokens):
        X_text = X_text if X_text.size(1) < self.__cfg.context_length
    else X_text[:, -self.__cfg.context_length:]
        logits, _ = self.__model(X_board, X_strength, X_reps, X_state,
    X_text,
                            (torch.zeros(1, X_text.size(1)) == 1).to(
    X_board.device), target_type=target_type)

        text_next = sampler.execute(logits[:, -1, :])
        X_text = torch.cat([X_text, text_next], dim=1)
        yield struct.pack(f"!{self.instance.__sp.vocab_size()}fI", *
    logits[0, -1, :].tolist(), text_next[0, -1].item())
        if text_next == self.__sp.eos_id():
            break
```

Listing 5.1: Streaming logits and sampled tokens

5.4.2 Proxy

Flask was used in the writing of the proxy too, since it shares a lot of common objects with the model. It decodes the binary stream received from the model and caches the logits of a given context(game, model settings) using a LRUCache, so they can be reused at a later time. It returns a plain text stream as a HTTP response. Listing 5.2 shows the code snippet responsible for transforming the stream from binary to text.

Decoupling the Model from the Proxy was done in order to ensure easy scalability, by adding multiple model instances, and running a load balancer between them. While this has not been done in this application, supporting this should require little to no changes.

Caching model responses has also been shown to be responsive, reducing the response time by a $5\times$ factor at best, and $2\times$ on average.

```
s = requests.Session()

with s.post(self.__model_url + "/get_commentary_execution", json=data,
    stream=True) as resp:
    for (logits, token) in self.consume_bytesio_stream(resp.raw):
        count_cache_miss += 1
        key = self.request_to_key(
            past_boards=past_boards,
            current_board=current_board,
            target_type=target_type,
            prefix=data['prefix']
        )
```

```
        self.__cache.set(key, logits)
        if token == self.__sp.eos_id():
            break
        token = self.__sp.IdToPiece(token)
        token = token.replace("_", " ")
        if len(data['prefix']) == 0:
            token = token.strip()
        data['prefix'] += token
        yield token
```

Listing 5.2: Streaming text from binary stream

5.4.3 Frontend

An Angular[1] application was developed to serve the frontend. It implements components for the features mentioned above in section 5.2, and displays them in a single page application.

5.5 Docker

In order to facilitate easy deployment and scalability, docker compose[24] was utilized. The following containers are created:

- *model*: A gunicorn[28] based container, that serves the model application.
- *proxy*: A gunicorn based container that sits between the model and frontend.
- *nginx*: A NGINX[30] container, responsible for serving the static files generated from Angular. It also sits as a reverse proxy between the browser and the backend proxy.

Figure 5.2 showcases the app architecture.

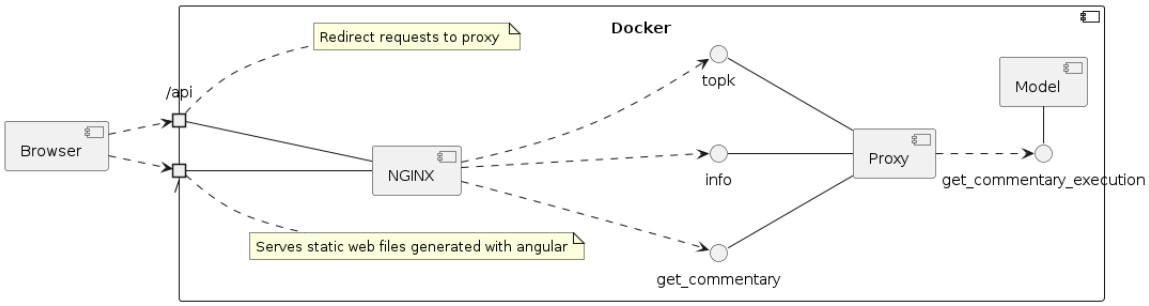


Figure 5.2: App component diagram

5.6 Class Diagrams

The application was developed in three main steps, each of them having its own class diagram.

The first step was setting up the data pipeline and training the model. Figure 5.3 displays the data pipeline class diagram. The two variants of input features described in 3.2 and 3.1 have their own dataset and datamodule implementation. The datamodules are responsible not only for preparing the data for the model, but also crawling and pre-processing it if it is missing or the user opts for a forced re-initialization.

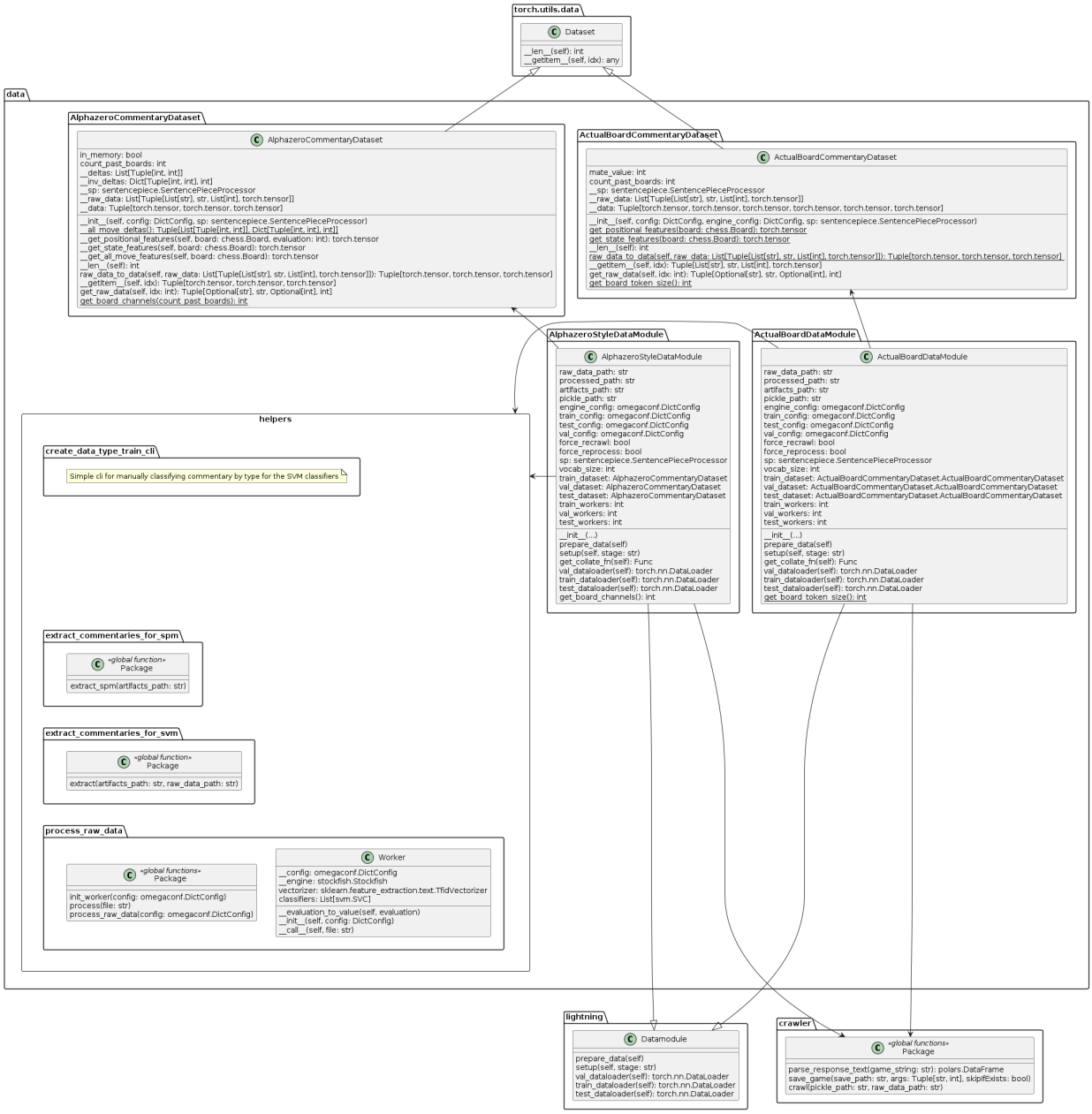


Figure 5.3: Data pipeline class diagram

Figure 5.4 shows how the model classes interact with each other. A list of modules is defined that represent building blocks in all models. Predictor classes are created for both input feature variants, and commentary models are implemented based on the modules. They inherit from lightning modules, which make them automatically take advantage of any tensor accelerator device available on the machine.

The second step was serving the model. Figure 5.5 shows how the classes interact for serving the model. The strategy design pattern was utilized in order to determine how the next token of the commentary should be sampled by the trained model. Chain of responsibility was also utilized, allowing for chaining multiple types of validators in order to ensure the validity of a request before it got to the model. Two singleton and facade classes were implemented to manage both strategies and validators, and be available to the Flask routes.

A frontend was built in order to display the model outputs. Figure 5.6 shows how the components and services of the frontend interact with each other. Being implemented with Angular [1], a model view controller architecture is automatically imposed, with services being the controllers and components being the views. The controllers each take care of a crucial part of the frontend:

- *GameStateService*: this service is responsible for keeping track of the current chess game, executing moves and undos/redos/seeks on behalf of the user, while updating other services and components on any such changes.
- *ModelBackendService*: this service is responsible for executing backend requests, whenever they are needed. It also contains automatic retry logic and makes requests to the backend when the game state changes in order to fetch the new top 10 next tokens, as specified in feature F6 5.3.1.
- *ChessEngineService*: this service is responsible for managing a chess engine which is run in the form of a webworker, and returning observables with evaluations from the engine on request.

Figure 5.4: Model class diagram

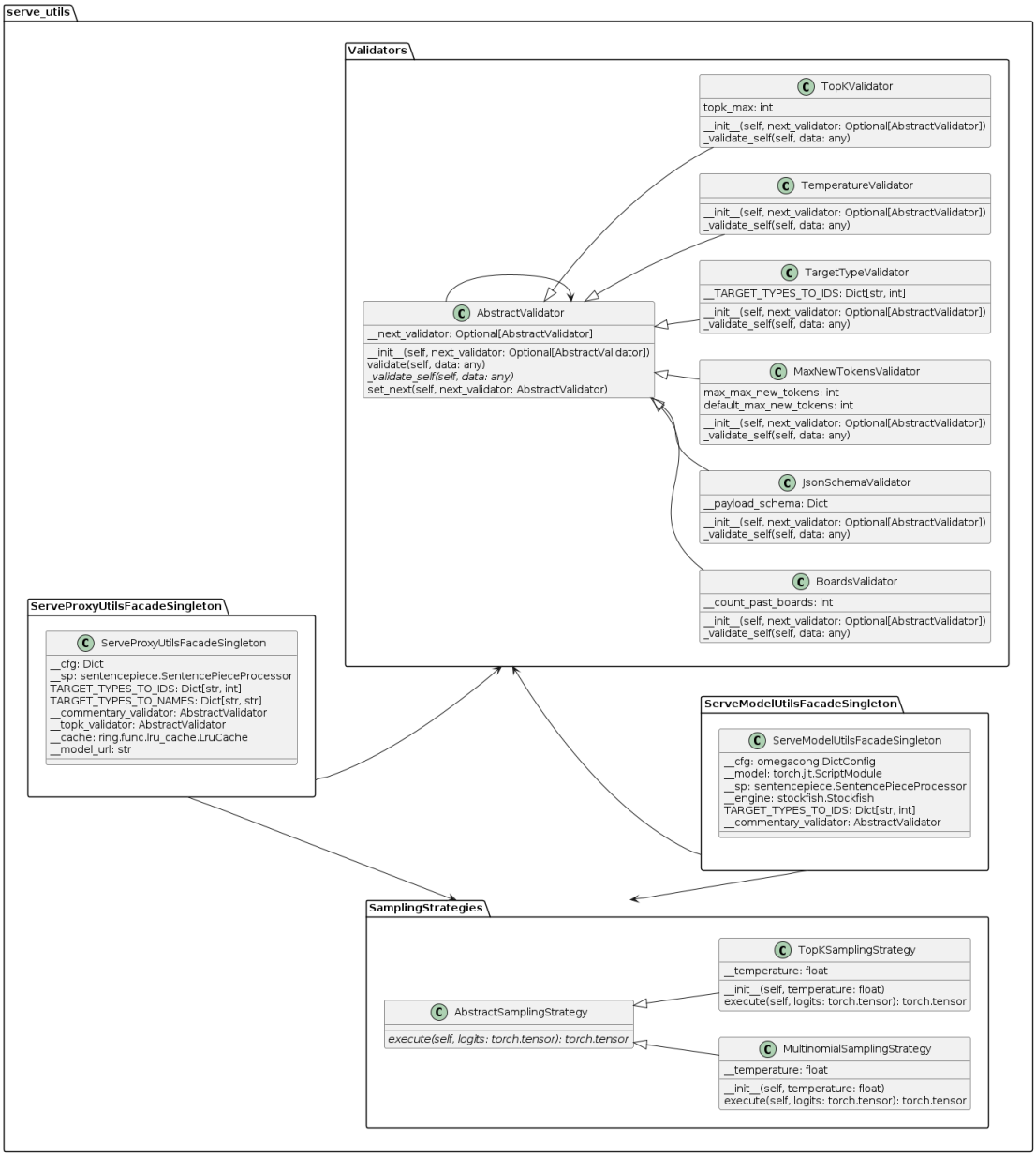


Figure 5.5: Serve model class diagram

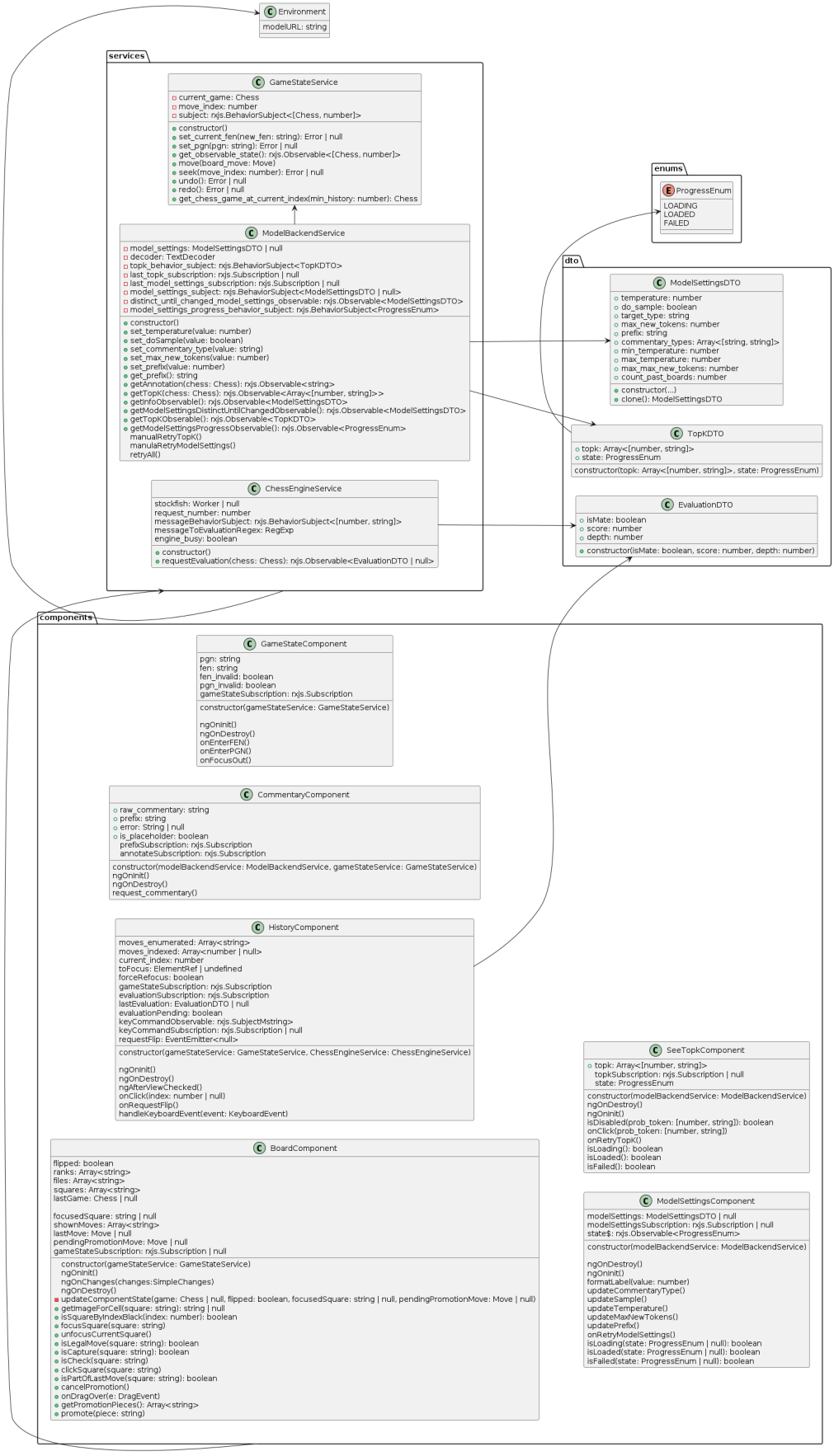


Figure 5.6: Frontend class diagram

5.7 Sequence Diagrams

In this section, sequence diagrams will be provided for all the usecases described earlier.

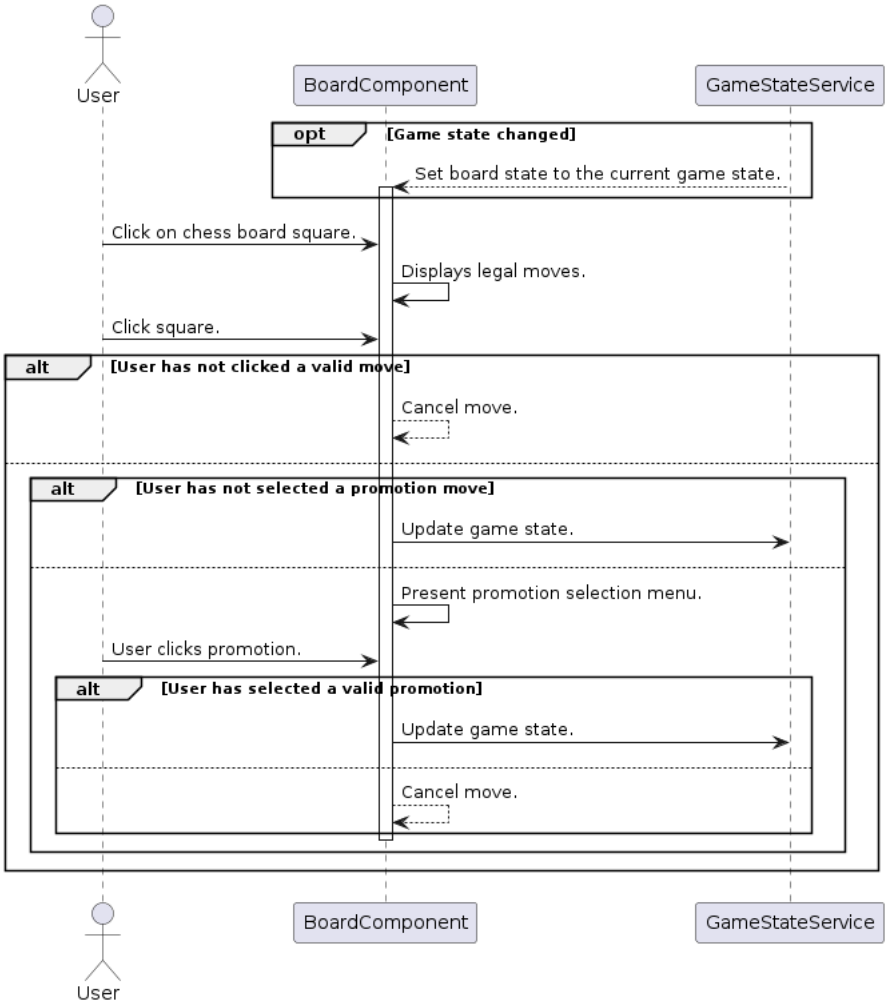


Figure 5.7: F1 5.3.1 sequence diagram

Feature F1 happens completely in the browser. The user can click on a chess board square, and legal moves will be displayed. On a click on a second square, the user can cancel the move by clicking an invalid target square, or continue it by selecting a legal move. If the move is a promotion, an additional promotion menu is presented to the user, from which they can cancel the move by clicking outside of it, or select the correct promotion by clicking on it.

Feature F2 happens completely in the browser. The user can seek the chess board after a particular move by clicking on the corresponding move in the history component. They can also undo the current move by pressing the left arrow; redo it by pressing the right arrow; and seek the beginning or the end of the chess game by pressing the up or down key respectively.

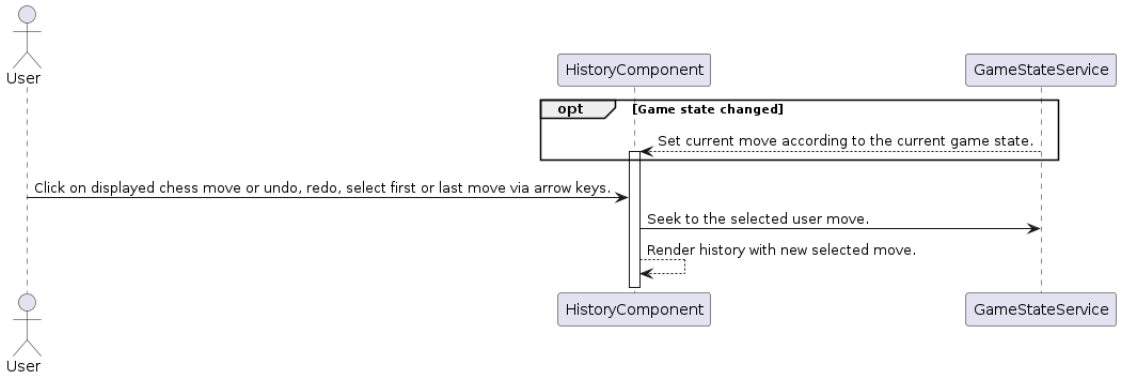


Figure 5.8: F2 5.3.1 sequence diagram

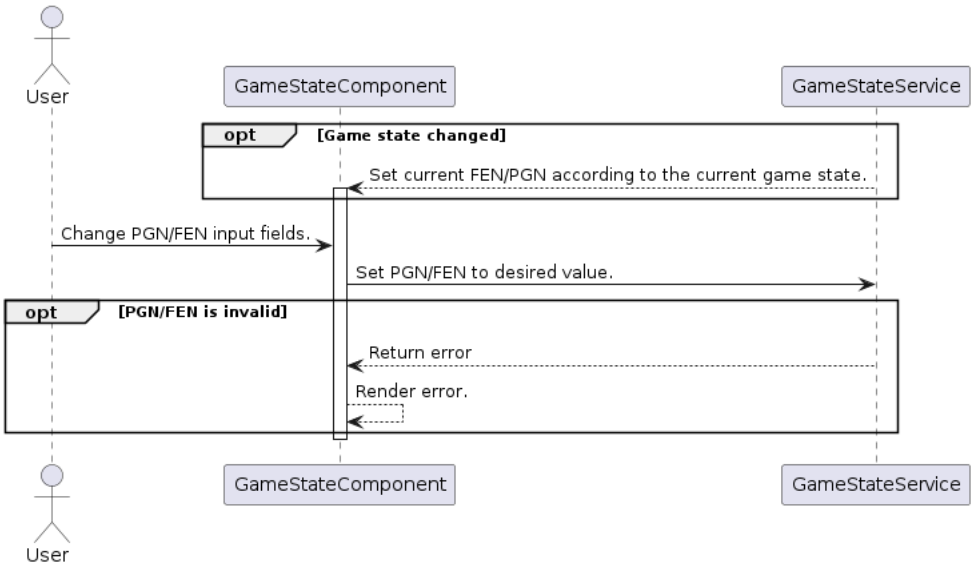


Figure 5.9: F3 5.3.1 sequence diagram

Feature F3 allows the user to change the PGN/FEN of the chess game by pasting it into two text areas. The component validates it, and displays an error under the text field if necessary. If it is valid, the new chess game is loaded.

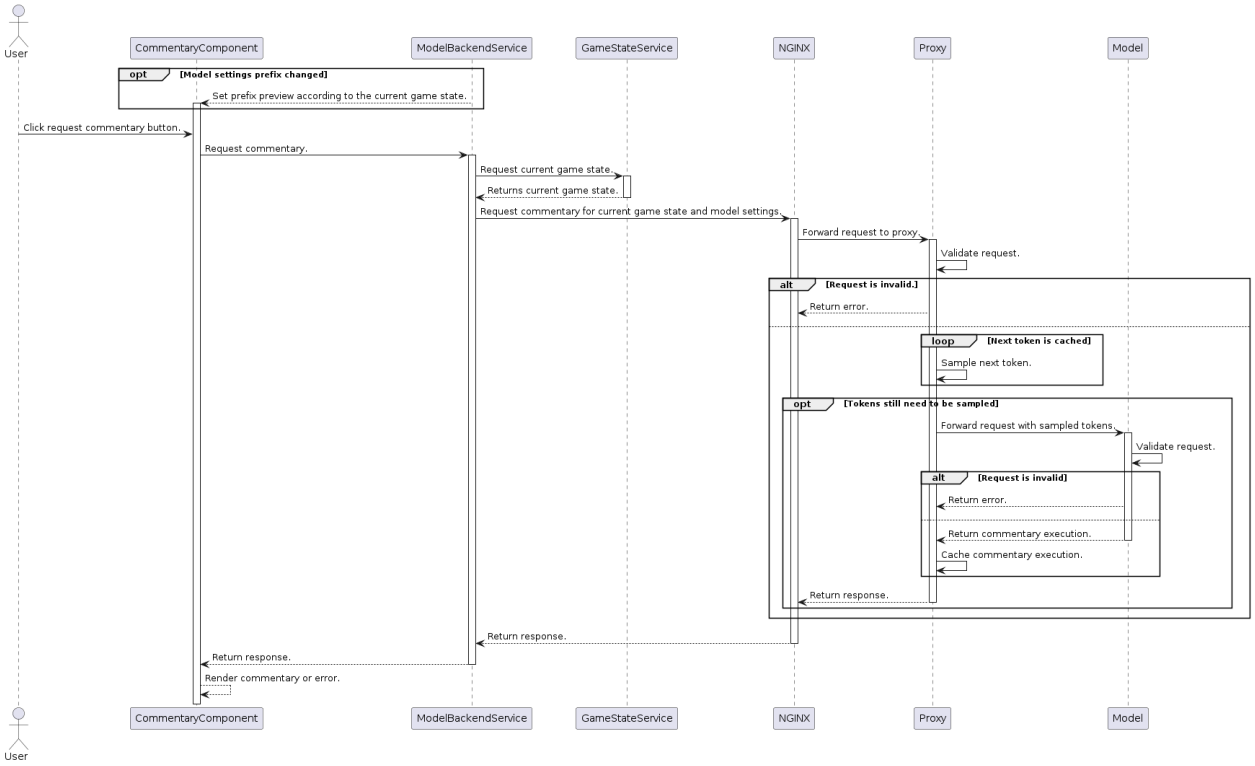


Figure 5.10: F4 5.3.1 sequence diagram

Feature F4 allows the user to request commentary from the model. Upon clicking the request button, the request goes through the *ModelBackendService*, which requests the current game information from the *GameStateService*, and makes a request to the backend. The request first goes through a *NGINX* layer, which forwards it to the *Proxy*. The *Proxy* validates the requests, and returns an error if it is invalid. Otherwise, it checks how much of the response can be generated from its cache, and forwards a modified request to the *Model* in order to get the rest of the commentary. The *Model* also validates this request, to ensure that the modifications performed by the proxy are valid. The *Model* returns an error in the case the request was invalid, or a binary stream containing the commentary execution details. The proxy then decodes it, caches it, and forwards a string stream all the way to the frontend, where the commentary can be rendered in the *CommentaryComponent*.

Feature F5 allows the user to change the model settings from the *ModelSettingsComponent*, by adjusting sliders, filling text fields or selecting a radio button. This feature is only available if the component has been successfully initialized by a successful call to the backend $/info$ endpoint. Otherwise, an error message is presented and a retry button is available for the user to manually retry the request.

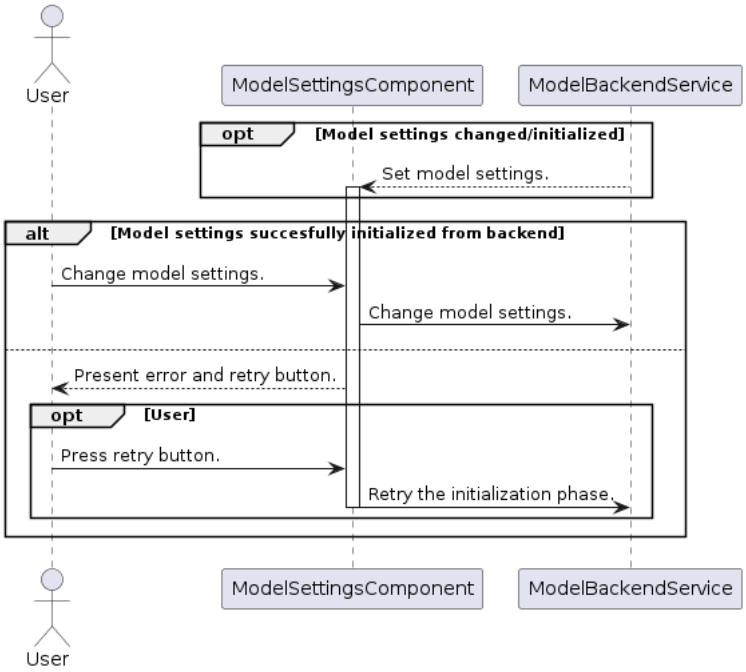


Figure 5.11: F5 5.3.1 sequence diagram

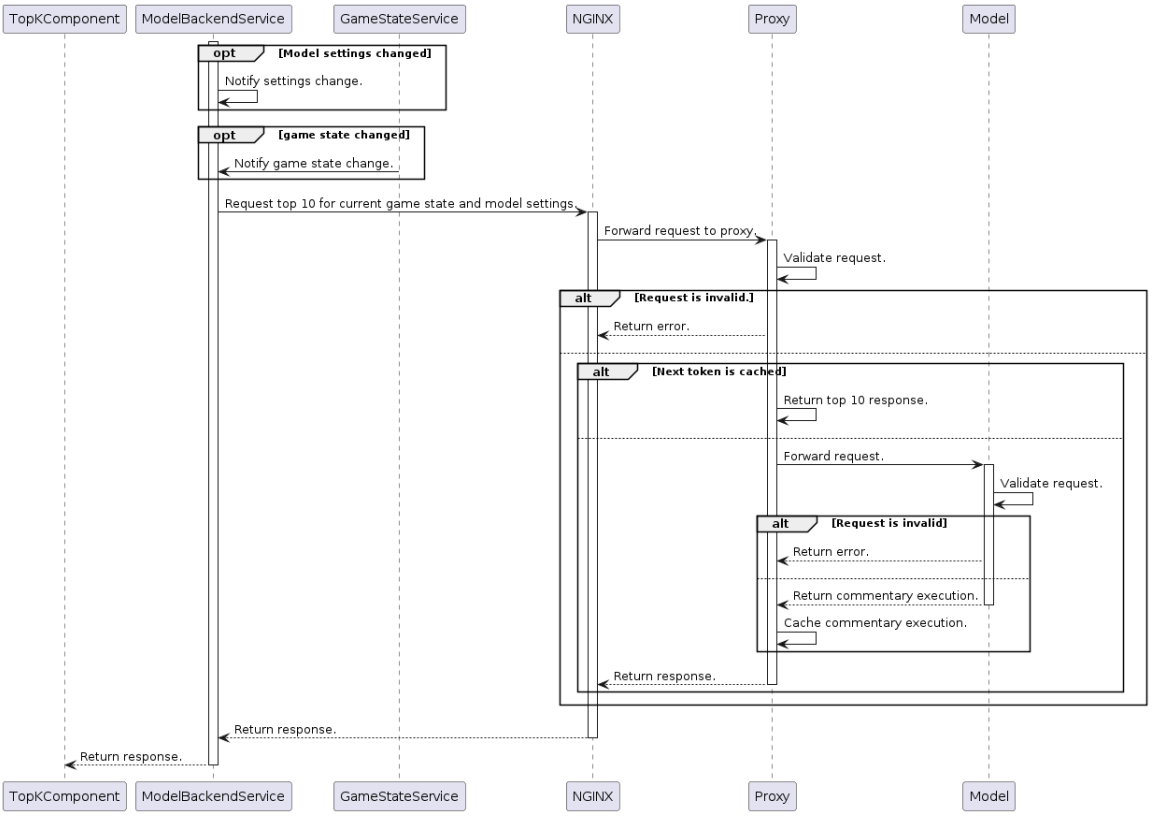


Figure 5.12: F6 5.3.1 sequence diagram

Feature F6 provides the user with a continuous feed of information regarding the probability for the next commentary token, based on the current game state and model settings. When it is notified of any changes that would modify the response of the request, it automatically sends a request to the backend, which goes through *NGINX* and the *Proxy*. At the proxy, the request is validated and an error is returned if it is invalid. If the *Proxy* contains the necessary information to respond to the request based on its cache, it returns the top 10 possible continuations. Otherwise, the request is modified and forwarded to the $Model$'s $/get\_commentary\_exection$ endpoint, where the request is validated again and an error is returned in case the request is invalid, or the binary commentary execution. The *Model* caches this response, and sends it back all the way to the frontend, where the *TopKComponent* can render the response.

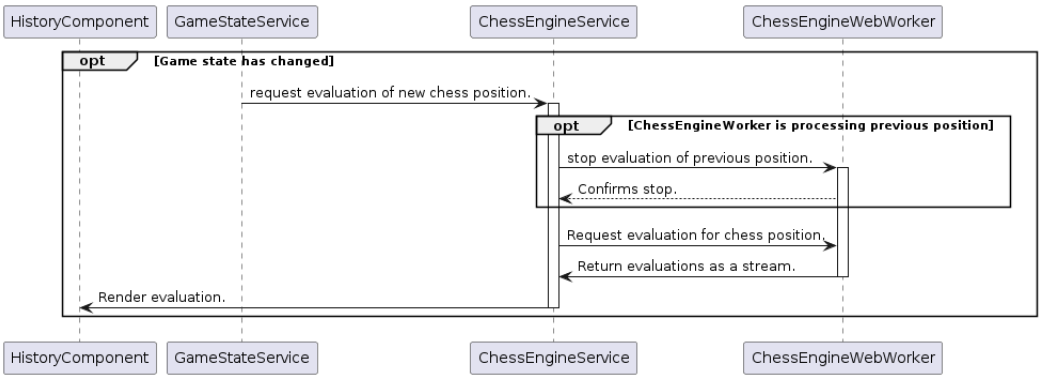


Figure 5.13: F7 5.3.1 sequence diagram

Feature F7 allows the user to gain insight into the strength of the current chess position by utilizing traditional chess engines. This is done automatically, when the *ChessEngineService* is notified by the *GameStateService* that the position has been changed. The *ChessEngineService* sends a stop request to the web worker, and after it is confirmed sends an evaluation request for the new position. An observable containing the feed provided by the webworker is returned to the *HistoryComponent*, where the evaluation can be displayed for the current position.

5.8 Application testing

The application was tested thoroughly via manual testing, together with automated testing, achieving more than 95% line coverage.

5.8.1 Model testing

The AI model has been constantly monitored during training, tracking stats such as $accuracy$ and training $loss$. Outputs from the $test$ dataset split have been sampled throughout training, in order to manually confirm that the model converges to some reasonable commentaries.

All of the components that were utilized in the training of the model were unit tested, from the crawler and data pre-processing to the prediction generation based on the model's output.

5.8.2 Proxy and Model testing

The proxy and the model components are just wrappers around the model. They offer some web endpoints, which the rest of the application can utilize in order to communicate with the model. They have been tested using unit tests, mocking both incoming and outgoing requests.

5.8.3 Angular frontend testing

All of the frontend components and services have been unit tested. Mocking was heavily used especially in order to test the frontend's resposne to various backend responses, but also to test how components react to messages from each other.

5.9 User Interface/User Experience

The user interface was designed in a way that offers a good user experience, being able to interact flawlessly with the model. They were modeled to follow the usecase specification.

5.9.1 Overall application

At startup, the app attempts to query the backend in order to properly initialize settings and hyperparameters. Figure 5.14 shows what the app looks like on startup if backend was reached successfully.

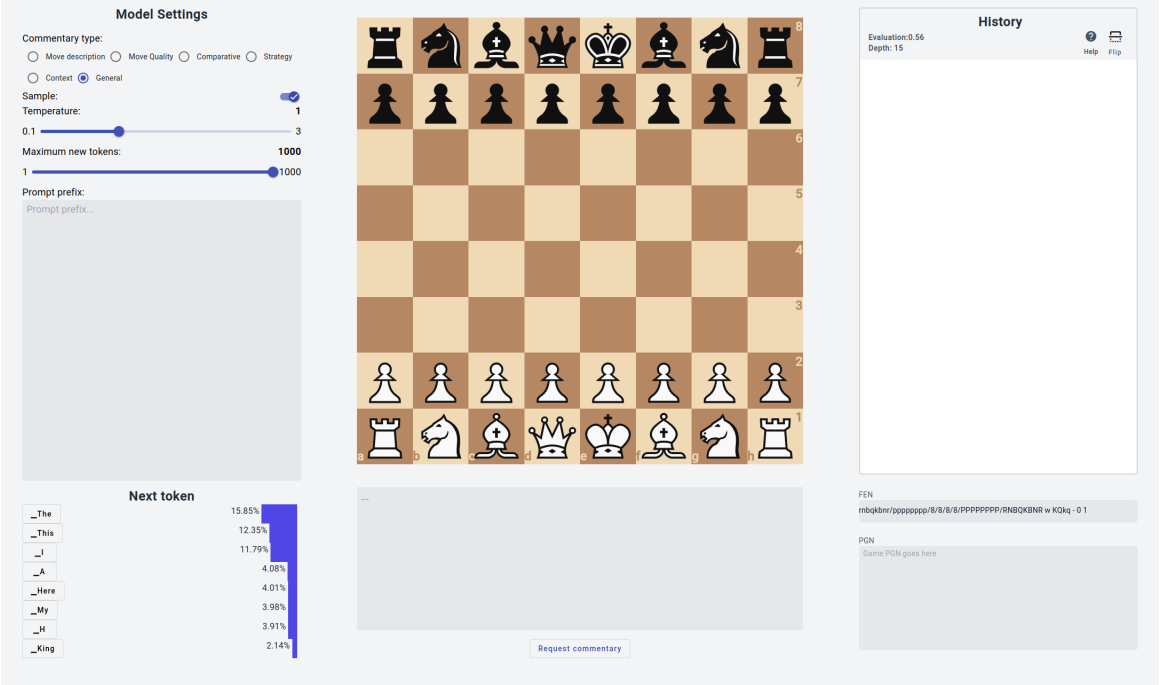


Figure 5.14: App if backend has been reached successfully.

Figure 5.15 shows what the application looks like in case the backend is not able to be reached. Pressing any of the two retry buttons present retries both of the backend resources needed. Figure 5.16 shows the app in a more arbitrary state, after the model settings have been changed(for example, the prefix, which is shown as a preview in the commentary textarea too), and after some moves have been made.

5.9.2 Board component

The board component is at the heart of the frontend. This component is responsible for showing the state of a chess game as an image(the actual chessboard), and allows the user to explore any possible chess game by playing moves on it. The user can move pieces by either clicking the source and target square(and optionally a promotion option), or by drag and dropping pieces from the source square to the target.

Figure 5.17 shows the board in an arbitrary position. On top of the chess position, it also shows what the last move was(by highlighting the source and target square of the last move).

Figure 5.18a shows a chess position where the black king is in check. The app highlights this by adding a red gradient on the king that is in check. Figure 5.18b shows the board when a piece is selected, and how it hints towards legal moves by inserting a green circle on the possible target squares. If the move is a capture, the corners of the square are highlighted instead. Figure 5.18c shows the board

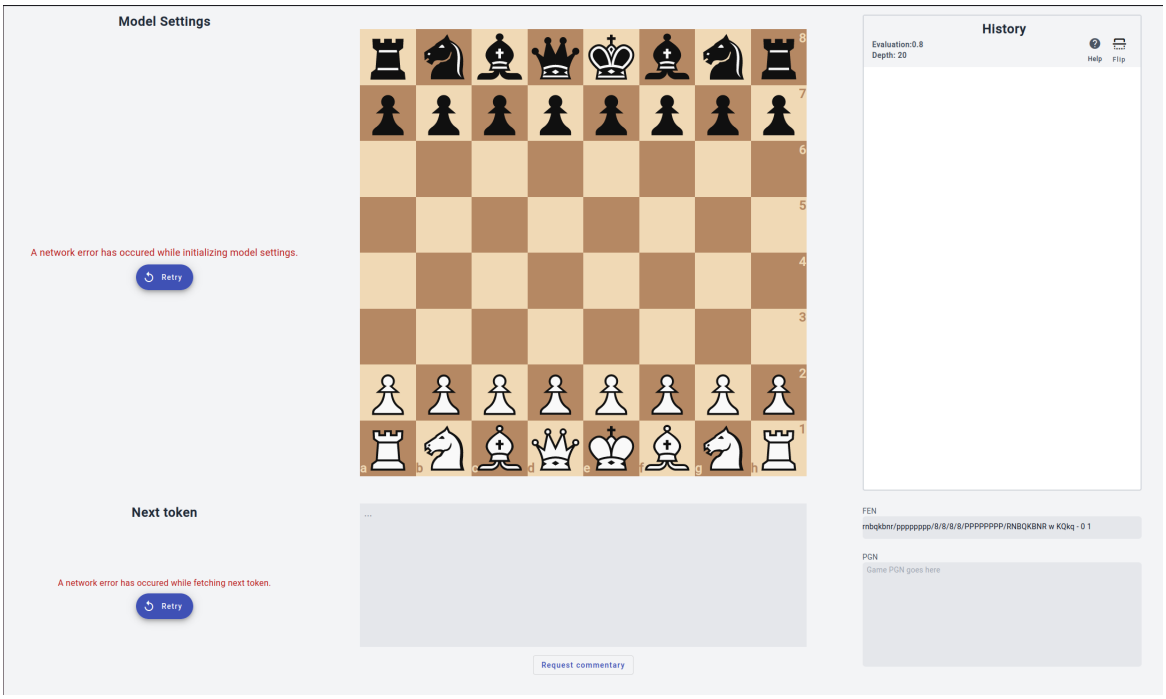


Figure 5.15: App if backend has not been reached.

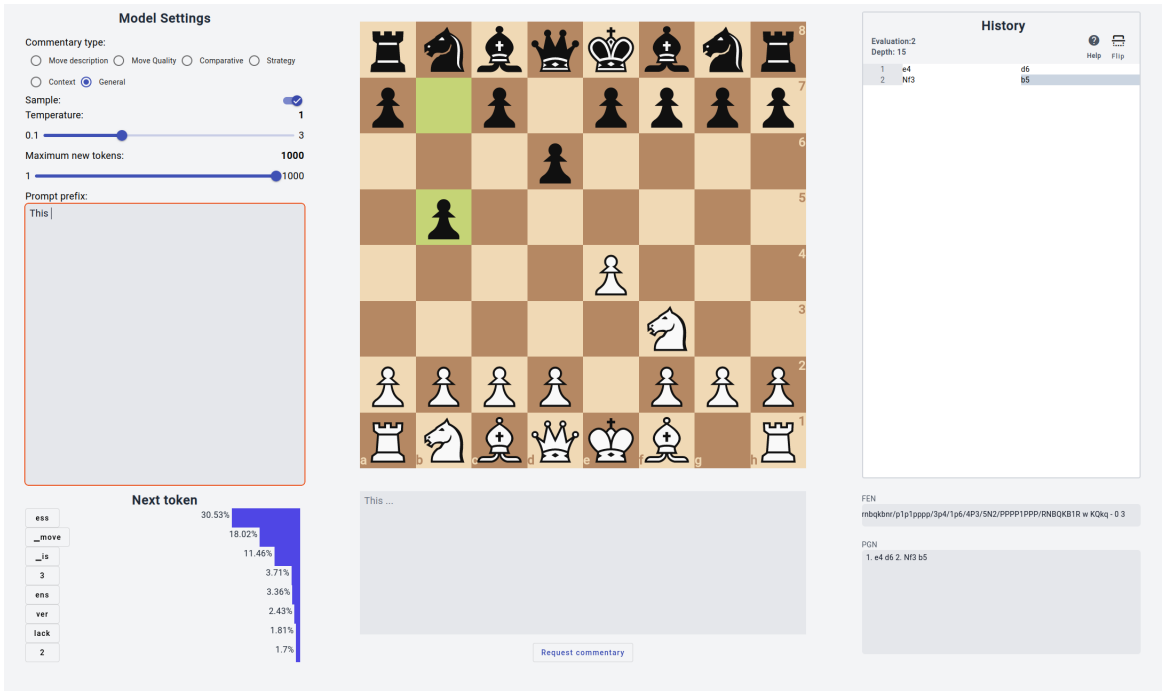


Figure 5.16: App initialized successfully with in arbitray state.

Figure 5.17: App initialized successfully with in arbitrary state.

from a flipped perspective, and Figure 5.18d contains the board component when a promotion needs to be selected.

5.9.3 Game state component

The game state component allows the user to import chess games from other places, such as chess websites, books, and so on. Figure 5.19 shows the component. This component has two functions: the first, as stated, is to import games by pasting their PGN [7] in the ”PGN” field, or by pasting their FEN [5] in the ”FEN” field; the second, is that the user can copy either of the fields in order to export the game too. They are updated as the game progressed, and also respond to undo, redo, or seek position changes. Figure 5.20 shows the component when invalid values are pasted. The error go away and the fields are reset when the user un-focuses the fields.

5.9.4 History component

The history component is responsible for allowing the user to seek the chess game to any previous(or future, if some moves have been undone) position. On top of that, it contains some quality of life features, such as a ”help” tooltip, signaling the fact that arrow keys can also be used for undo, redo, and seeking the first or last chess position; a flip button, that flips the board perspective, like seen in 5.18c. In the top-left corner of it, a live engine evaluation is displayed for the current position,

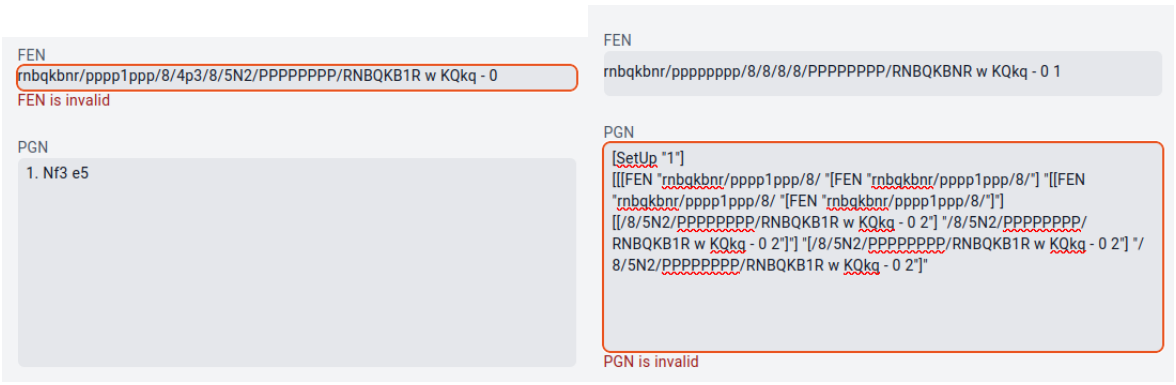
(a) Board in check.

(b) Board showing legal moves.

(c) Board from flipped perspective.

(d) Board showing promotion menu.

Figure 5.19: Game state component.



(a) Game state invalid FEN. (b) Game state invalid PGN.

Figure 5.20: Game state invalid values.

with both it's strength and the depth up to which the engine has explored. This evaluation is updated as the engine reaches further depths or the position changes. The evaluation is performed on the clientside, using the Stockfish chess engine [33]. Figure 5.21 contains the history component.

5.9.5 Model settings component

The model settings component is responsible for tinkering with the model settings, tuning in order to receive the desired commentary type, and exploring the autocomplete feature of the model. It initializes it's default values and category options by querying the backend at the start. Figure 5.22 shows the component, both in the case that the query succeeded or that it failed. If it failed, a retry button is present, that allows the user to retry the connection. Being a component whose state depends on a backend request, a spin-loader variant of it also exists.

5.9.6 Commentary component

The commentary component is responsible for displaying commentary generated by the model. In figure 5.23, the component can be seen with no commentary present, but with a preview of the prefix used from the model settings. Figure 5.24 shows the component, both in the case that the request succeeded and in the case that it failed. The button from this component is pressed when the user desired commentary, and the stream response is placed in the component, token by token.

5.9.7 Next token component

The next token component displays the top 10(a scrollbar is available if they do not all fit in the component) continuations for the supplied prefix that the model

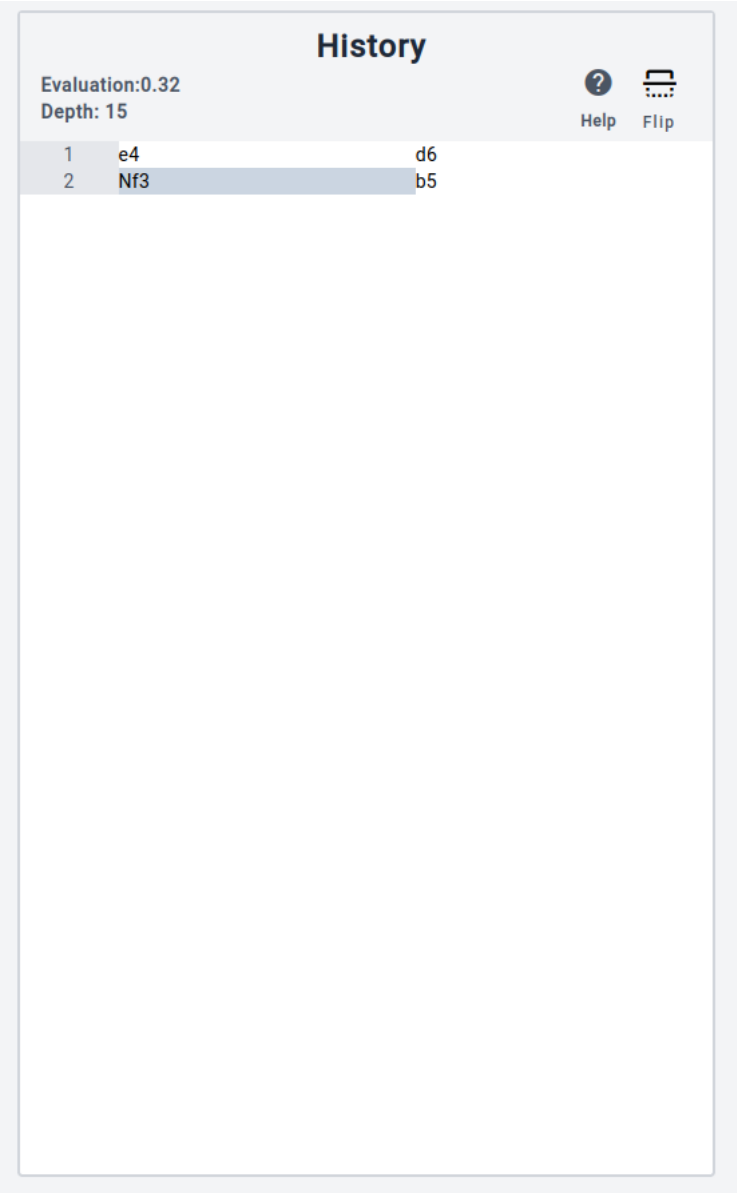
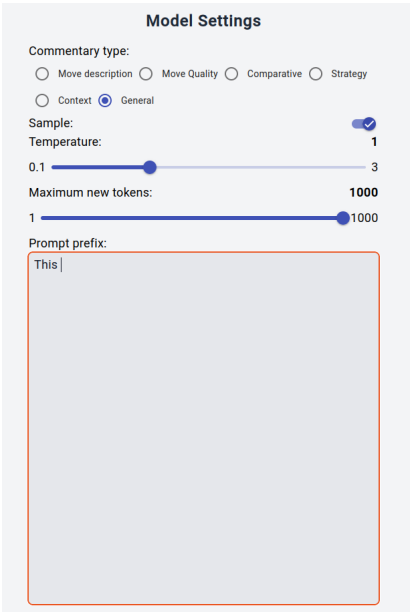
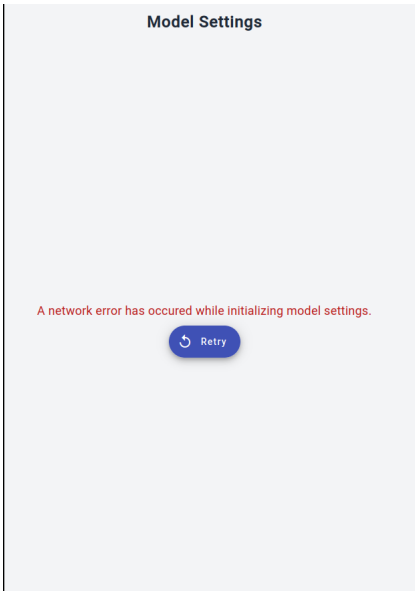


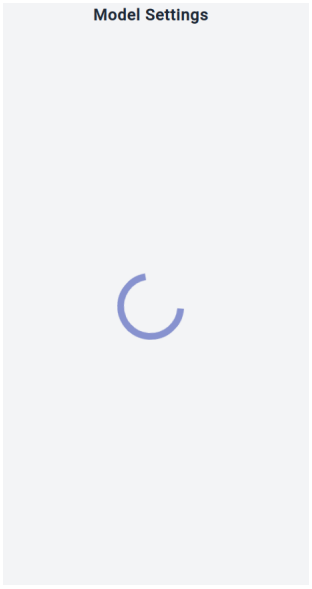
Figure 5.21: History component.



(a) Model settings.



(b) Model settings if backend query failed.



(c) Model settings loading.

Figure 5.22: Model settings component

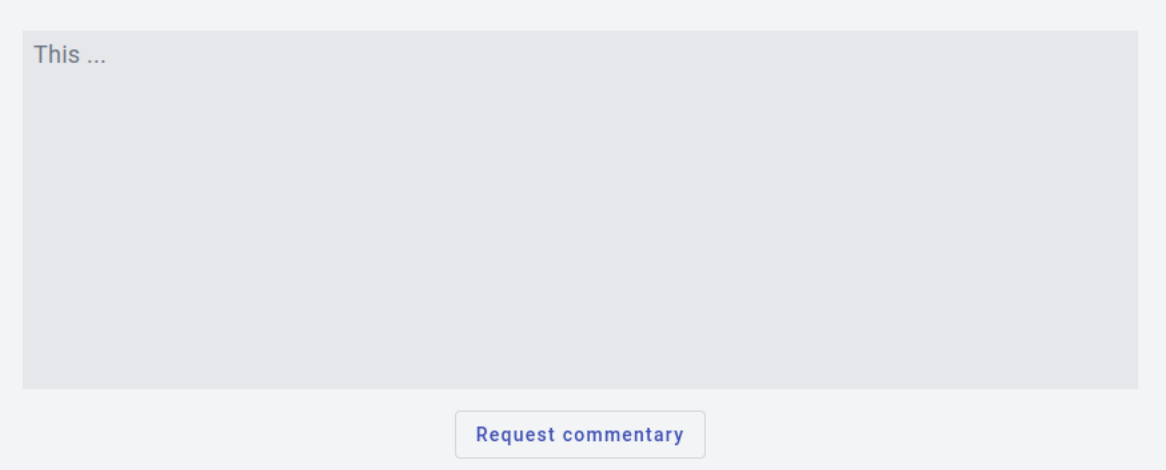
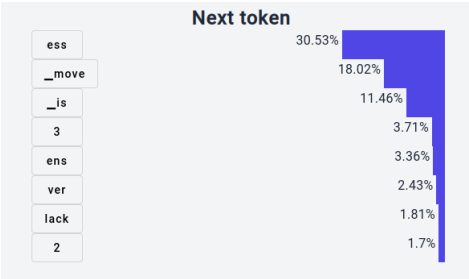


Figure 5.23: Commentary component.

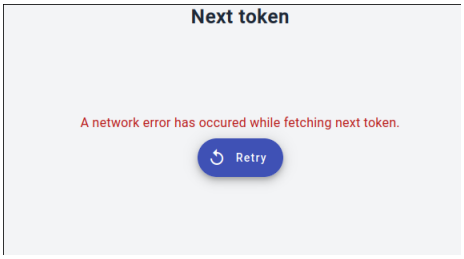
(a) Commentary request was successful. (b) Commentary request was unsuccessful.

Figure 5.24: Commentary component request variants.

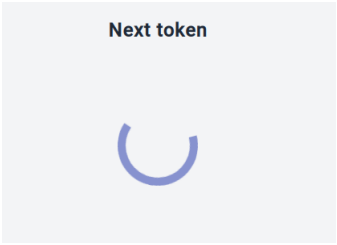
suggests, together with their probabilities. It updates automatically on game state changes and model settings changes. In order to get the continuations, it queries the backend, and this request can fail. Figure 5.25 shows the component in both cases. In the case that it failed, the retry button can be pressed to manually retry the request. In the case it succeeds, the continuations are buttons that can be pressed in order to automatically add the continuation to the prefix. Because this component queries the backend, a spin-loader variant of it also exists.



(a) Next token successful.



(b) Next token unsuccessful.



(c) Next token loading.

Figure 5.25: Next token variants.

Chapter 6

Conclusions and future work

Overall, we explored the problem of generating chess commentaries for games, with the motivation of creating a chess tutor for beginner/average players, but also in order to see how much of a complex game like chess it can understand. For this, the transformer architecture was utilized, due to its performance in natural language processing tasks. This produced a model which is able to sound human, and produce sentences which could pass as written by humans. However, even though in some cases the model produces valid and useful commentary, in most it shows a lack of understanding of the game, confusing pieces and colors between them.

6.1 Future work

One approach which might prove useful is integrating a neural-network based chess engine with the transformers architecture. A classic neural network chess engine like Alphazero might already help a lot, but a reimplementation of Alphazero to use transformer encoders and self attention blocks might cooperate better with a transformer decoder, rather than a Resnet Alphazero chess engine. Afterwards, fine-tuning the chess engine to produce outputs which are more useful to commentary generated might also help.

Another point of improvement could be the dataset. While this is the best one available, and used in the papers which make up the state of the art, the fact that it is based on a social chess forum, where any user can annotate games provides commentary which might differ completely in style from other users, and thus make it harder for the model to grasp what is happening. Also, the commentary sometimes looks highly subjective(with sentences like ”this looked good to me” or ”i didn't expect to win”), which does not offer that much insight into the position. A dataset annotated by chess players with a minimum rating requirement, and with stricter rules about how to commentate positions would definitely improve the accuracy of

the model.

As far as the application is concerned, it can be improved on both the backend and the frontend. For the backend, having some sort of automatic scaling of the services would improve the latency of the responses. Kubernetes could be used for this. On the caching side, a redis instance could be used in order to accommodate multiple proxy services, which would share their executions through it.

As for the frontend, quality of life changes such as supporting premoves, saving evaluations and displaying engine suggested attacks could help the user see the true best approaches of the position and cross-check the commentary with the actual best move.

Bibliography

[1] Google angular team. Angular. https://angular.io/. accessed 27.05.2024 13:29 UTC+03:00.

[2] Satanjeev Banerjee and Alon Lavie. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In Jade Goldstein, Alon Lavie, Chin-Yew Lin, and Clare Voss, editors, *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*, pages 65–72, Ann Arbor, Michigan, June 2005. Association for Computational Linguistics.

[3] Olivier Buffet, Olivier Pietquin, and Paul Weng. Reinforcement learning, 2020.

[4] Murray Campbell, A.Joseph Hoane, and Feng hsiung Hsu. Deep blue. *Artificial Intelligence*, 134(1):57–83, 2002.

[5] Wikipedia contributors. Forsyth–edwards notation. https://en.wikipedia.org/wiki/Forsyth%E2%80%93Edwards_Notation. accessed 10.05.2024 12:14 UTC+03:00.

[6] Wikipedia contributors. Minmax. https://en.wikipedia.org/wiki/Minimax. accessed 27.05.2024 12:16 UTC+03:00.

[7] Wikipedia contributors. Portable game notation. https://en.wikipedia.org/wiki/Portable_Game_Notation. accessed 12.04.2024 16:10 UTC+03:00.

[8] Li Gong, Josep Crego, and Jean Senellart. Enhanced transformer model for data-to-text generation. In Alexandra Birch, Andrew Finch, Hiroaki Hayashi, Ioannis Konstas, Thang Luong, Graham Neubig, Yusuke Oda, and Katsuhito Sudoh, editors, *Proceedings of the 3rd Workshop on Neural Generation and Translation*, pages 148–156, Hong Kong, November 2019. Association for Computational Linguistics.

[9] Michael Cerny Green, Ahmed Khalifa, Gabriella A. B. Barros, Andy Nealen, and Julian Togelius. Generating levels that teach mechanics. In *Proceedings*

of the 13th International Conference on the Foundations of Digital Games, FDG '18. ACM, August 2018.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[11] Harsh Jhamtani, Varun Gangal, Eduard Hovy, Graham Neubig, and Taylor Berg-Kirkpatrick. Learning to generate move-by-move commentary for chess games from large-scale social forum data. In Iryna Gurevych and Yusuke Miyao, editors, *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1661–1671, Melbourne, Australia, July 2018. Association for Computational Linguistics.

[12] Hirotaka Kameko, Shinsuke Mori, and Yoshimasa Tsuruoka. Learning a game commentary generator with grounded move expressions. In *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 177–184, 2015.

[13] Mike Klein. Google's alphazero destroys stockfish in 100-game match. `https://www.chess.com/news/view/google-s-alphazero-destroys-stockfish-in-100-game-match`. accessed 27.05.2024 14:36 UTC+03:00.

[14] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[15] Scikit learn team. Scikit-learn. `https://scikit-learn.org/stable/`. accessed 27.05.2024 13:52 UTC+03:00.

[16] Jen-Wen Liao and Jason S. Chang. Computer generation of Chinese commentary on othello games. In Von-Wun Chang, Jason J. andMike Klein Soo, editor, *Proceedings of Rocling III Computational Linguistics Conference III*, pages 393–415, Taipei, Taiwan, September 1990. The Association for Computational Linguistics and Chinese Language Processing (ACLCLP).

[17] Soichiro Murakami, Akihiko Watanabe, Akira Miyazawa, Keiichi Goshima, Toshihiko Yanase, Hiroya Takamura, and Yusuke Miyao. Learning to generate market comments from stock prices. In Regina Barzilay and Min-Yen Kan, editors, *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1374–1384, Vancouver, Canada, July 2017. Association for Computational Linguistics.

[18] Researchgate Olarik Surinta. Resnet block. `https://shorturl.at/FTfVB`. accessed 12.04.2024 16:10 UTC+03:00.

[19] OpenAI. Chatgpt. `https://chatgpt.com/`. accessed 27.05.2024 13:55 UTC+03:00.

[20] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics*, ACL '02, page 311–318, USA, 2002. Association for Computational Linguistics.

[21] Aleksander Sadikov, Martin Možina, Matej Guid, Jana Krivec, and Ivan Bratko. Automated chess tutor. In H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. (Jeroen) Donkers, editors, *Computers and Games*, pages 13–25, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[22] Alex Sherstinsky. Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. *Physica D: Nonlinear Phenomena*, 404:132306, March 2020.

[23] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm, 2017.

[24] Docker team. Docker. `https://www.docker.com/`. accessed 27.05.2024 13:30 UTC+03:00.

[25] Flask team. Flask. `https://flask.palletsprojects.com/en/3.0.x/`. accessed 27.05.2024 13:50 UTC+03:00.

[26] Gameknot team. Gameknot. `https://gameknot.com/`. accessed 27.05.2024 13:23 UTC+03:00.

[27] Google team. Sentencepiece. `https://github.com/google/sentencepiece`. accessed 12.04.2024 16:10 UTC+03:00.

[28] Gunicorn team. Gunicorn. `https://gunicorn.org/`. accessed 27.05.2024 13:28 UTC+03:00.

[29] Hydra team. Hydra. `https://hydra.cc/docs/intro/`. accessed 03.06.2024 16:32 UTC+03:00.

[30] Nginx team. Nginx. `https://nginx.org/en/`. accessed 27.05.2024 12:28 UTC+03:00.

[31] PyTorch team. Pytorch. `https://pytorch.org/`. accessed 03.06.2024 16:31 UTC+03:00.

[32] PyTorch Lightning team. Pytorch lightning. `https://lightning.ai/docs/pytorch/stable/`. accessed 03.06.2024 16:31 UTC+03:00.

[33] Stockfish team. Stockfish. `https://stockfishchess.org/`. accessed 27.05.2024 12:19 UTC+03:00.

[34] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[35] Weight and Biases team. Weight and biases. `https://wandb.ai`.

[36] Hongyu Zang, Zhiwei Yu, and Xiaojun Wan. Automated chess commentator powered by neural chess engine. In Anna Korhonen, David Traum, and Lluís Màrquez, editors, *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 5952–5961, Florence, Italy, July 2019. Association for Computational Linguistics.

[37] Maciej Świechowski, Konrad Godlewski, Bartosz Sawicki, and Jacek Mańdziuk. Monte carlo tree search: a review of recent modifications and applications. *Artificial Intelligence Review*, 56(3):2497–2562, July 2022.