Transforming Chess: Investigating Decoder-Only Architecture for Generating Realistic Game-Like Positions

William Pettersson

Autumn term 2023

Umeå University

Department of computer science

Degree of Master of Computing Science and Engineering, 300 credits

Supervisor: Henrik Björklund
Abstract

Chess is a deep and intricate game, the master of which depends on learning tens of thousands of the patterns that may occur on the board. At Noctie, their mission is to aid this learning process through humanlike chess AI. A prominent challenge lies in curating instructive chess positions for students. Usually these are either manually found by going through large numbers of real games, or handcrafted – a time-consuming process. For effective learning, it is often useful to collect many positions following the same theme, or exhibiting the same type of pattern. Curating such collections from real games is a challenging task.

This thesis investigates the transformer decoder-only architecture and its capability of generating realistic, game-like chess-positions. This investigation involved the development and training of a decoder model using Pytorch, and a simple web-based Turing test gaining larger understanding of testers experience.

The developed chess model successfully generates chess positions, with constraining possibilities of fixed pieces, score intervals, and fixed empty positions. Controlled re-generation ensures satisfaction of score intervals, while empty positions are handled by iterating over the model's probabilities. Based on the limited data provided by the Turing test, the model seems to fool players below 2000 rank-points on chess.com, where guess percentages land near the 50 percent mark, providing no clear indication that it deviates from randomness.

Keywords

Artificial intelligence, Transformer decoder-only, Generative architecture, Generation with constraints, Chess, Turing test
Acknowledgements

I am deeply grateful to Samuel Sonning at Noctie for providing me with the opportunity to conduct my master's work at their company, and making me feel supported and accommodated every step of the journey. This experience has been invaluable to my academic and professional growth, and I appreciate the support and guidance received during this period.

I would also like to express my sincere gratitude to my thesis supervisor, Henrik Björklund, university Lecturer at the Department of Computer Science, for dedicating time and effort to mentor me during the completion of this thesis. Without his support, this report would not have achieved its quality.
6.5 Vocabulary design and its effect 31
6.6 Choice of database 31
6.7 Future work 31
  6.7.1 Further training & testing 31
  6.7.2 Develop a model for GANS 32
  6.7.3 Investigate “empty” in vocabulary 32

List of references 33
1 Introduction

This chapter will introduce the thesis, giving the reader a background to the area and an understanding why this work is relevant. The overall aim of the thesis is explained in addition to clear scientific goals and milestones of the project. Lastly, a brief summary of the remaining chapters of the thesis.

1.1 Background and Objective

This project has been conducted in cooperation with a locally based company named Noctie. Chess is a deep and intricate game, the mastery of which depends on learning tens of thousands of the patterns that may occur on the board. At Noctie, their mission is to aid this learning process through humanlike chess AI. A prominent challenge lies in curating instructive chess positions for students. Usually these are either manually found by going through large numbers of real games, or handcrafted – a time-consuming process. For effective learning, it is often useful to collect many positions following the same theme, or exhibiting the same type of pattern. Curating such collections from real games is a challenging task.

1.2 Overall aim

This project seeks to develop an AI capable of generating realistic and diverse chess positions, for this purpose, specifically two primary architectures will be taken into consideration: decoder-only Transformers and (conditional) Generative Adversarial Networks (GANs/cGANs), where one will be chosen to implement.

1.3 Problem definition & Research questions

The problem which the project seeks to solve is how to generate instructive chess positions following the same theme or exhibiting the same type of patterns using artificial intelligence. The research questions this thesis aims to answer are the following:

1. Which of the proposed architectures is better suited for the task at generating realistic chess-positions with a possible extension of constraints & external vectors.

2. To what extent can this generative architecture generate realistic, game-like chess-positions as if they were to occur in a real match.

3. To what extent do the generated positions conform to the provided constraints.
2 Theory

In this chapter things required to understand the work will be explained and additionally introduce some papers on related work which this work will stand upon. During the theoretical investigation at the start of the project, the transformer decoder-only architecture was argued better suited for the task, and therefore will be more thoroughly described. The reasoning behind the choice of architecture can be viewed in section 3.2.

2.1 Generative adversarial networks

Generative adversarial networks, also called GAN’s have risen in the world of AI as a cutting-edge machine learning architecture. The framework has gained a lot of publicity and popularity from its powerful generative capabilities, generating realistic data, such as images natural and text which has blown up in the mainstream media. GAN’s can be simply described as two neural networks playing a min-max game where one acts as the counterfeiter(generator) and the other acts as the appraiser(discriminator). The counterfeiter tries to generate as realistic data samples as possible in order to fool the appraiser, while the appraiser tries to sort out the generated samples from the real ones. GAN’s are in this manner unsupervised, where the two neural networks continuously improve the other, by the generator generating data samples and feeding it together with real samples to the discriminator. If the generator fools the discriminator the discriminator back-propagates the error, and if the discriminator manages to catch the fakes, the generator back-propagates the error.¹ Further reading on GANs.²

2.2 Decoder-only transformers

Decoder-only transformers are a variation within the transformer architecture designed specifically for sequence-to-sequence tasks. The model generates its symbols in an autoregressive manner, where each generated symbol acts as an additional input for the next generation. A decoder is generally composed of a positional encoding layer followed by masked multi-head attention layers and a feed-forward layer. There are residual connections around the attention and feed-forward layers which are then followed by normalization.\(^3\)

---
\(^3\) “How does the (decoder-only) transformer architecture work?”, AI Stack Exchange, 2024-01-03, [https://ai.stackexchange.com/questions/40179/how-does-the-decoder-only-transformer-architecture-work](https://ai.stackexchange.com/questions/40179/how-does-the-decoder-only-transformer-architecture-work).
Having normalization layers help optimize deep learning models training process by boosting stability and efficiency. It does this by scaling the vector gradients but preserving the tokens relations.\footnote{Hassan Daar, “What is the role of Layer Normalization in GPT models?”, educative, 2024-01-03, \url{https://www.educative.io/answers/what-is-the-role-of-layer-normalization-in-gpt-models}.}
The flowchart in figure 1 is the specific architecture used in this project. The process of a decoder-only transformer can be described as the input to the model is firstly fed through the input embedding and positional encoding layers, this step yields an encoded representation for each token within the sequence, in order to capture both the semantic meaning and positional value. The encoded vector then gets fed into the stack of decoder blocks, where it then enters the multi-headed attention. The method of multi-headed attention helps the model learn a more nuanced understanding for the tokens. The cumulative output from these attention heads are then combined back into one vector containing the information of the total attention score. This vector gets added with a residual connection from its input. The feed-forward layer is then applied to capture complex patterns in the data, this also has a residual connection. These residual connections' purpose is to counteract the vanishing gradient problem when training deep networks. The final output vector of these decoder blocks are then fed through another normalization layer, and lastly through a linear layer which results in the logits for each token in the vocabulary, presenting each tokens probability. Softmax is then used in order to get the most probable token.\footnote{Ketan Doshi, “Transformers Explained Visually (Part 3): Multi-head Attention, deep dive”, towardsdatascience, 2024-01-03, \url{https://towardsdatascience.com/transformers-explained-visually-part-3-multi-head-attention-deep-dive-1c1ff1024853}.}

Transformers have gained a lot of attention for their use in natural language generation where they perform exceptionally well. A very commonly mentioned example is ChatGPT which is built upon the transformer model.
2.3 Related work

In this section a selection of related work will be presented which give an insight into the architectures at hand, what they have been used for, how they perform, and some novel approaches to their traditional structures. These papers were the main source of information when searching for arguments to answer which seemed best suit the choice of architecture, presented in chapter.

2.3.1 Attention is all you need

This is a groundbreaking research paper in the field of deep learning and NLP which introduced the transformer model architecture to the world, and it has then become the foundation for many state of the art models. The paper goes over the transformer architecture in depth, and describes everything you might need to create your own model.\(^6\)

2.3.2 A survey of transformers

This research paper states itself to be the first comprehensive literature review on the transformer architecture variants. The paper firstly gives a good understanding about the background of transformers, in what areas it has been adopted and why. It then goes on to an in-depth explanation of the vanilla transformer, proposed in the main paper. This explanation thoroughly explains each individual module within the transformer, including formulas, complexity charts and analysis. The rest of the paper is a deep-dive into the wide variety of architecture modifications proposed since transformers adoption.\(^7\)

2.3.3 Improving Language Understanding by Generative Pre-Training

This thesis introduces the generative pre-trained model to the world, which was a huge breakthrough in the world of natural language generation as well as machine learning in general. The introduced GPT model is based upon the transformer architecture but does not have the standard requirement of task-specific training, instead relying on extensive pre-training on a large amount of language data, gaining a broad understanding of language patterns and structures, allowing it to perform well in a range of tasks such as question answering, semantic similarity assessment, entailment determination, and text classification.\(^8\)


\(^7\) Tianyang Lin, Yuxin Wang, Xiangyang Liu, and Xipeng Qiu, "A survey of transformers," School of Computer Science, Fudan University, Shanghai, China, Shanghai Key Laboratory of Intelligent Information Processing, 2022, 111-132, Available at [https://www.sciencedirect.com/science/article/pii/S2666651022000146](https://www.sciencedirect.com/science/article/pii/S2666651022000146).

2.3.3 Generative Adversarial Networks

This work is by some considered a foundational work in the field of generative modeling, introducing generative adversarial networks. The paper proposed a framework of two neural networks, via an adversarial process estimating generative models. These two networks, one generator, and the other a discriminator, constantly improving through competitive game of trying to beat its counterpart. Generative adversarial networks have shown great generative capabilities in generating realistic data, especially gaining traction in the world of image generation.  

2.3.4 SeqGAN: Sequence Generative Adversarial Nets with Policy Gradient

In summary this paper introduces a sequence generation framework, which they call SeqGAN, an interesting solution towards countering the limitations of traditional GAN’s in sequence generation. They incorporate reinforcement learning and Monte Carlo search, working around the issues constraining the model. The paper is considered a success and its novel approach of the traditional GAN has shown, according to the report, significant improvements in comparison over strong baselines.

2.3.5 Transformer playing chess

This study is extremely relevant for this thesis, using one of the proposed architectures within a very similar problem area. The thesis explores the use of a transformer model for understanding and generating strategies for games, with a large focus on chess. In this work they show that their model learned to play chess, effectively generating strategic meaningful moves.

---


2.4 Pytorch\textsuperscript{12} 

Pytorch is a commonly used open-source machine learning framework, simply put a python library for implementing AI solutions. Developed by the Facebook AI team, it has gained widespread adoption due to its ability to expedite development processes by simplifying the creation, training, and deployment of AI models. This is achieved through the provision of pre-built components that enable users to construct custom architectures effortlessly.

2.5 Google Colaboratory\textsuperscript{13} 

Google Colaboratory, often called Google colab, is a cloud-based platform that simplifies development and execution of python scripts. It provides free use of GPU and TPU resources as well as seamless integration with Google Drive, thereby handling both storage and execution of your projects. The platform is collaborative and interactive, allowing multiple users to simultaneously manipulate code and get live updates, streamlining cooperation. The platforms design of managing code in communicating blocks, makes testing separate components of your project faster and easier, bypassing unnecessary execution of non-relevant code. These combined properties have made google colaboratory a popular choice for researchers as well as students.

2.6 Firebase / Firestore\textsuperscript{14} 

Firebase and Firestore are services developed by Google. Firebase acts as a platform streamlining web development by handling essential parts of the process. The service provides authentication handling, hosting free of charge alongside detailed usage analytics. Firebase together with Firestore is a lethal combination, where Firestore acts as an NoSQL document-oriented database, simplifying the management and usage of cloud-based data storage. Firestore gives its users real-time data synchronizations, making it a suitable choice for dynamic web applications.

\textsuperscript{12} https://pytorch.org/
\textsuperscript{13} https://colab.google/
\textsuperscript{14} https://firebase.google.com/
3 Methodology

In this chapter the numerous components of the projects will be discussed, the process of their development alongside the reasoning behind the various design choices.

3.1 Theoretical analysis of the architectures

The first part of the project was a thorough analysis of the two architectures. At first youtube became the main source of information, watching various graphical guides and explanations, giving a rough understanding of the two architectures. This was not intended as a basis for the choice, rather a foundation to stand on, making understanding the coming research papers easier. Secondly google scholar became the main source of information on which the concluding arguments would be based upon. Here a large amount of papers were skimmed in order to find a selection found to be relevant and well written. These papers\textsuperscript{1, 2, 6-10, 15} were then read over several days, multiple times in order to fully grasp the content. Lastly, discussion with the company to argue the various properties of the two architectures.

3.2 Chosen Architecture

The Transformer architecture became the clear choice for this work after a theoretical analysis of the two architectures described in section 3.1, and their applicability to the issues at hand. Generative adversarial networks have achieved a state-of-the-art level in generation, and they have recently gained significant attention for their applications in image generation. With that said, the model has some limitations. In the paper SeqGAN: Sequence Generative Adversarial Nets with Policy Gradient\textsuperscript{15}, they explain the issue that GANs is primarily intended for generating continuous, real-valued data and faces challenges when attempting to generate sequences comprised of discrete tokens, this limitation does not provide a solid basis for generating chess sequences. Nevertheless, following this comprehension, both architectures remained under consideration. The standard transformer decoders are in their nature autoregressive, meaning they incorporate previously generated tokens, in combination with current token as input when generating new tokens. This allows us to simply set the wanted fixed positions, feed this set as input and the transformer, generating the next tokens in regular fashion without any significant modification to the model. There are non-autoregressive versions of this architecture, however, after this study I would argue that the standard autoregressive versions seems to be the better choice.

On the other hand implementing constraints with GANs is a considerably more challenging task, where even a simple constraint such as fixed positions would take considerable work and training. Training a GAN to generate within a fixed positions constraint, would require it to be trained on a large amount of positions containing that specific constraint, allowing it to connect the constraint-vector, with that type of position pattern.

While both models should be proficient in generating chess positions, and GANs, in all likelihood, would even provide a more streamlined approach for the initial implementation, it's crucial to note that the drawbacks of this method outweigh the benefits. The decisive factor became the availability and possibility for implementing constraints on model generation, which would make the end result significantly more useful for Noctie, as well as making the project substantially more interesting.

### 3.3 Data preparation

To train an AI model you need a significant amount of data for the model to learn any broader patterns, generally the more data the better, with the exception of problems such as overfitting, which isn't a problem in this scenario. This large amount of data was gathered from an online chess service which stores all games played on their webpage, which is publicly available for people to use freely. The company is called Lichess.\(^6\) Specifically the data used is gathered from January 2017. Python scripts were developed to prepare the data for training. Firstly only the positions were interesting for the training of the model, so we had to iterate over each and every move from the games in the database to get every occurring position. These positions were stored in a notation called PGN, short for “Portable Game Notation”, which is often used as a digital representation for chess positions. This felt unnatural for this project, and certainly restricting. The goal of the project was to be able to have a model generate realistic positions, with the option of giving constraints such as a given set of fixed positions. With this in mind the PGN were seen as unsuitable, making the implementation of these restrictions severely harder to implement. Instead a new notation for the positions were developed, where we stored each piece, paired with their individual position on the board as a sequence.

Still there was one issue with the representation. If we wanted to say, fix two rooks in the middle of the board, we had to give the model those two rooks as an initial input, but the representations were in the natural order of the board? This was solved by a simple shuffle. After the development of the new notation, the ordering of the pieces carried no actual value, since each and every piece had its type, color and position stored within its own token. And once the ordering lost its value, we lost the issue of giving fixed positions as input.

In figure 2, a sequence, together with its FEN-code and board representation is presented.

---

\(^6\) [https://lichess.org/](https://lichess.org/)
Since the script was working with large data, it utilized multi processing to enhance time efficiency, still the process took several hours.

The FEN-code describes the current position, working from top left, describing each row.

- Uppercase → White piece
- Lowercase → Black piece
- Slash ‘/’ → Next row
- Number X → X empty positions

Looking at figure 2, we start with 8 empty positions in row 8, a slash then indicates the start of a new row. On row 7 we start with 5 empty positions, followed by a lowercase “p”, meaning black pawn. Next comes a lowercase “k” meaning black king, followed by another black pawn and so on. Following the notation, you work your way through each row until the full board has been iterated.

The sequences instead have their positional data embedded in each token, by creating piece-position pairs. Looking at figure 2 we start off with a start token, followed by “black king on g7”, “white pawn on a2”, “white knight on c6” etcetera. The tokens have a randomized order to help training, if the positions had to come in order, applying constraints would be severely hindered since we could not for example give the model a white king on position b4 as initial input.
3.4 Creating the model

The largest component of the project was by far the development of the model, however a large amount of time was saved at the start of the project by finding related works and building upon their progress. The developed model is largely based on a model created by a developer named Andrej Karpathy who had, using pytorch, created a NL model for generating Shakespeare plays. This work was accessible through github, additionally Karpathy had created a youtube guide where he discussed how the model worked and how it was developed.17 His model followed the same principal architecture as this project aimed to investigate, only for a different purpose.18

Author/s, Title of video [format], (Publisher, Date created), URL, access date.

Firstly Andrejs model was transferred into a google colaboratory project, where it got split into natural blocks, meaning separating the fetching of the dataset, creating the vocabulary, declaring the architecture, initiating the model, training and generation. Breaking down the project into individual code blocks not only facilitates a better understanding of the overall structure, but also enables the iteration of specific code segments and dynamic code execution based on the memory and context retained from previously executed segments, enhancing the overall efficiency of the project. Running the project through google colaboratory also enabled working from nearly anywhere, not being restricted by the local hardware, nor storage since its seamless connection to Google Drive. During development of the model the standard given CPU was used to avoid depleting the free quota of GPU time on the service. However during training phases, and testing of training loops, the T4 GPU was the utilized hardware.

Later on when training became a bigger part of development, project advancement faced limitations due to these daily quotas quickly depleting. In order to overcome this issue, a Colab Pro subscription was acquired, providing sufficient GPU quotas and extended runtimes making longer periods of training less restrictive.

The main architecture of the model could be kept since the two problems differed in some aspects they were still fairly similar in nature. Figure 3 presents a flowchart describing the architecture.

17 Andrej Karpathy, “Let’s build GPT: from scratch, in code, spelled out.”, [Online video], Youtube, 2023-01-17, https://www.youtube.com/watch?v=kCc8FmEb1nY&t=3323s&ab_channel=AndrejKarpathy, accessed 2024-01-03.
The model was developed using the popular AI library pytorch. There was not any specific reasoning behind this other than the fact that Andrej Karpathy's related work used this library, in addition to its wide use giving me lots of sources and examples to go off, if trouble arised.

The first adaptation that had to be made to the model was to adapt how it passed data through its various functions. Originally the model worked with letters as tokens, minimizing the vocabulary, which obviously were unsuitable for generating chess positions. Instead we went with piece-position pairs as tokens, giving us a vocabulary size of 771. His model took a section of 128 characters, randomly out of natural text, and fed it through the network, and did this a large amount of iterations. This was adapted to instead work with chess positions, a sequence of 34 tokens which is the largest possible combinations of tokens for a chess board, including start and end tokens, giving us a block-size of 34. This blocksize had to be static and the sequences needed to always be this length, so padding tokens were implemented in sequences where pieces were eliminated. Figure 4 presents an example of a sequence, where “PAD” continues until 34 tokens, the repeating “PAD” tokens are hidden.

Figure 4: Example sequence, padding would repeat until the sequence contained 34 tokens.
The generation function had to be heavily adapted, where we had to implement the possibility for constraining its generation to certain requirements. Here we implemented temperature, an interval for acceptable score, and lastly blank tokens. The original model could already handle fixed tokens since the architecture in nature is recurrent, making it a natural capability. Temperature was implemented for the control of randomness of the generations, influencing its diversity, in other words forcing the generations to be more erratic. This is something that was easy to implement. The model uses a softmax function to select from the generated probabilities of the upcoming token, so if we lower the peaks, and raise the lows we increase the probability of it selecting a less probable token. This is performed by dividing the logits by the temperature, higher temperature equals further scaling the probability values equals more randomness.

The implementation of acceptable score intervals and blank tokens were more rudimentary, since a more elegant solution could not be thought of in an acceptable time frame. For the acceptable score interval model generates a full sequence and compares the generated sequence score to the acceptable interval. If the score is outside of said interval the model generates a new sequence until it either generates an acceptable candidate or eventually after x amount of tries returns an unacceptable sequence paired with a code saying it failed. The fixed blank spots worked in a similar manner, where the model generated a token, checked if the list of wanted blank tokens contained this token, if it did, it regenerated a new token until acceptable. This solution however had a backup if it failed x amount of times, it then started to work through the probabilities in order until it found a generation that was acceptable. However, if it worked through all this without finding an acceptable candidate it returned a blank board with fail-code.

### 3.5 Hyper parameter testing

Once the model started working as intended hyperparameter testing began. The model proved to work surprisingly well early on in the project. With vastly differing combinations it could produce a more than acceptable result according to Noctie, where they argued that the generated positions could arguably be considered realistic to some degree. Still a large section of this work was spent on hyper-parameter testing, however, it could be done in parallel to other work. The architecture has the following hyper-parameters: batchsize, blocksize, learning rate, number of embeddings, number of heads, number of layers, as well as dropout rate. During this phase we tried to find patterns in order to understand which parameters were the most impactful regarding the efficiency of the model, based on measurements of accuracy as well as time consumption. Early on the number of heads seemed to have very little impact, so that parameter was left as its original value of 4, as did dropout-rate, which was left as 20%. Overfitting did not seem to be an issue and was thereby not in much focus. Testing showed that the batch-size only had an impactful difference up till 32. Past this number, the model's capability of running code in parallel seems to be unable to take advantage of any larger size. The amount of layers had a small impact on training-loss, but a large negative impact regarding training-time. So the decision was made to have 1 layer as baseline, but to investigate the models performance on more layers. The number of embeddings immediately showed a large positive impact on
training-loss, as well as not drastically increasing training time. Lastly testing came down to evaluating layers’ impact when working with 512 embeddings in order to see if the extra cost in time could be worth it. The data showed that training-loss improved with the number of layers, where the step from 1-2 layers made substantial improvement. Further increasing the layers had progressively less impact on performance, while the increase in training-time kept rising. This imbalance led to Noctie deciding to go with the combination of 2 layers and 512 embeddings for training the final model, allowing more time to be spent on evaluating.

3.6 Approaches for accuracy measurement

When it came to hyper-parameter testing, a measurement for the models performance was needed in order to determine which combinations performed better than others. The base model for natural language generation, developed by Andrej, used cross entropy between the targets and logics, a commonly used approach. This became the measurement on which the hyper-parameter testing was based, however, initially, this was somewhat challenging to comprehend, so other metrics of accuracy were developed. One of these measurements was comparing each individual coordinate on the two boards, and comparing piece information on said coordinate. If the coordinate from the generated board were an exact match to its real counterpart, keeping track of the count. Once every coordinate had been parsed, the matches were compared to the total, giving a measure or accuracy. This however led to a consistently high accuracy, embellishing the perception of its potential, since a large amount of coordinates were on average empty on both boards. Based on this realization, comparisons were strictly made where the real board contained pieces. If the generated board had a piece outside of the coordinates the real board had, it counted as incorrect. Then it compared every coordinate where the real board had pieces, and compared for exact matches in the generated board. The count of exact matches compared to the total then showed the accuracy in a truer light. These measurements' purpose was solely for improved understanding of the progress, not for inclusion in the final results or any formal analysis. The final training of the model utilized the initial accuracy measurement of cross-entropy due to its superior time efficiency.

3.7 Generating data

When hyperparameters had led us to a desirable model providing generations with passable accuracy, data had to be generated which would represent the models capability of answering the research question two, “To what extent can this generative architecture generate realistic, game-like chess-positions as if they were to occur in a real match”. The planned method of evaluating the model was a type of turing-test, where the participants are presented a pair of chess-positions, where one were generated using the model, the other taken from a real game of chess. However the team foresaw an possible issue, if the model were to generate freely, and the real game were taken at complete random, there could possibly arise a pattern such as the number of pieces generally being fewer on the generated board, the score differing etcetera. The process began at selecting the real positions. A script selected at random positions from the database containing real positions, and applied a filter, sorting out the positions that were too close to the starting positions. The filter counted the anomalies in row 1,2,7,8, if the number of anomalies were greater than 8, the positions were considered acceptable. Now we had the real positions for our test. The
generated positions were then created using the real set of positions as foundation. For each real position, 20 percent of the pieces were taken at random, acting as the first constraint on the generation. The second constraint necessitated an acceptable score, requiring the generated position to exhibit a similarity in values on the boards by maintaining a score within a 5-point distance from the real board. Lastly the generated position had to follow the same filter of anomalies to assure the board not being too trivial. These constraints were designed to minimize the users ability to base its guesses on a pattern rather than gamelike appearance.

3.8 Creating the turing-test

In order to be able to argue regarding the models capability of answering research question two a turing-test were created. The aim here was to keep it as simple as possible, with minimalistic design to reduce bias in attention. A decision was made to make a web based test in order to streamline the process of data collection. This decision was great for us to reach an acceptable amount of data in a short time, but it may possibly have led to a less trustworthy result, taking in consideration we can keep track of the participants state, if its a unique tester, internet trolls and much more. Firebase and Firestore became the choice of hosting and data management for its simplicity and suitability for the test nature. The database were fashioned in the simplest thought of way. One collection for storing the positions together with that pairs correct and incorrect number of guesses. Another collection for storing survey data from the user, which contains the participants answers regarding game knowledge, confidence in guessing, thoughts of realism, lastly the participants degree of recognizing a pattern rather than degree of realism. The structure is presented in figure 5.

Figure 5: Firestore database overview, displaying document content.
The test was designed in which the user is presented two randomly selected chess-positions, one generated, the other real. The user is then asked to click on the board they think contains the position generated using AI. Clicking on either of the boards checks whether the user guess was correct or not, and updates that specific pairs values accordingly in the database. Next the user is presented a new pair of positions as well as if they guessed correctly on the previous pair. This specific design choice was made in order to keep the users interest, in order to get the user to provide us with more data before leaving the site. This was especially important to ensure that as many as possible performed the entirety of the test, allowing linking between the users survey, describing game knowledge etcetera with their resulting score. Making it possible to compare the results based on this information.

3.9 Evaluation through Turing test
The manner in which the model was evaluated was a statistical analysis of the data gathered from the Turing test, together with argumentation of various perceived patterns. For the model to be seen as capable of generating realistic game-like positions, the result of the Turing test should align with randomness, meaning participants could strictly be guessing. If the result strives too far from 50/50, on either side, there is evidence for the participants being able to see a pattern which they could use to distinguish between real and generated.

4 Experimental setup
In this chapter the various parts of the experimental setup will be presented such as the final version of the transformer model with its combination of hyper-parameters. In addition to this the various planned tests will be described combined with reasoning behind its design.

4.1 Model hyper-parameters
The generated positions used in the Turing test were created using a model trained on the following combination of hyper-parameters presented in table 1. This combination leads to a total of 7.11 million parameters.

<table>
<thead>
<tr>
<th>Hyper-parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Batchsize</td>
<td>32</td>
</tr>
<tr>
<td>Blocksize</td>
<td>34</td>
</tr>
<tr>
<td>Learning rate</td>
<td>3e-4</td>
</tr>
<tr>
<td>Embeddings</td>
<td>512</td>
</tr>
<tr>
<td>Heads</td>
<td>4</td>
</tr>
<tr>
<td>Layers</td>
<td>2</td>
</tr>
<tr>
<td>Dropout</td>
<td>20%</td>
</tr>
</tbody>
</table>

Table 1: Hyper-parameter combination used when training the model which generated data for Turing test.
These parameters were decided based on a combination of factors. The hyperparameter tests showed that we could reach a higher accuracy from the model if the number of embeddings and layers increased, however the improvement in accuracy was outweighed by its cost in time consumption. The final decision of these parameters were decided in collaboration with the company based on a presentation of hyperparameter testing data, to make sure the company was satisfied with the outcome.

4.2 Turing test

The Turing test was performed through a created website described in section 3.6. The nature of the test being web based brought some worries regarding its trustworthiness in testers. At first the option of sending out the page on various chess forums were discussed, allowing a larger amount of data to be gathered, however this would also include the risk of “online provocateurs” flooding the database with noise. The decision was taken to send it out to friend groups and groups the team deemed more trustworthy, in order to reduce that risk. The amount of data that could be collected would be drastically lower, however the argument was that an acceptable amount of data could most likely still be gathered.

4.3 Statistic analisis

The data gathered from the Turing test will provide binary data regarding the participants ability to distinguish between real and generated boards, in the form of correct or incorrect guesses. In order to evaluate this data, a binomial test can be employed in order to evaluate if their responses display a pattern, deviating what's expected from random choice.
5 Result

In this chapter the trained model’s capabilities will be presented through example generations, with various constraints. The developed Turing test will be displayed, as well as statistical analysis of the data gathered from users performing the test.

5.1 Model capabilities

The resulting model can generate complete chess-positions according to the rules of the game with three available constraints. The first constraint is the possibility to set fixed pieces in the position as an initial input for the model. The second is the option of a required score interval, however this constraint is not naturally generated by the probabilistic generation of the model, rather a built-in controlled re-generation until a satisfactory position. Same goes for the last constraint, to fix empty positions on the board, is applied through iterating over the models order of probabilities. Figures 6,7 and 8 present three generations using various combinations of constraints.

In figure 6, the generator were given two fixed piece-positions as a constraint. The generation took less than a second, produced a legal position which followed the given constraint.

![Chessboard](image)

Figure 6: Presenting example board generation given a set of fixed tokens.

Figure 7 illustrates a generation in which the generator was given free reins, though with a lower temperature setting. The generation was completed in less than a second, producing legally valid positions.
In Figure 8, the generator was subjected to three specific constraints: a predetermined piece-position, a defined score interval, and several forced empty positions. The generation executed in less than a second, adhering to each constraint.
5.2 Turing test website

When first entering the website, the user is presented with the option to select the number of pairs they are willing to review, presented in figure 9.

![Select number of pairs](image)

Figure 9: Users first interaction with the site, presenting a choice of number of pairs to review.

After selecting the wanted number of pairs and submitting, the user is directly presented with a pair of chess positions, one generated using the model, another from a real set. The user has minimal impressions outside of the positions, only text describing to click on the board the user thinks is generated using AI, in addition to feedback whether the last guess was correct or incorrect. Figure 10 is the compressed view of the browser.

![Chess positions](image)

Figure 10: Compressed view of the browser, when a user is performing the test.

Once the user has gone through the entirety of their wanted pairs, a feedback survey is presented to gather further information about the users experience during the test, see figure 11. Also enabling further understanding of the different experiences connected to game-knowledge.
5.3 Statistical evaluation of Turing test data

In this chapter statistical evaluation will be presented based on the data gathered from participants taking the web based Turing test. In Table 2, the number of correct, and incorrect guesses based on entered ranking interval. These numbers only include data that can be linked to a survey, meaning the participant completed their full test. Score is calculated by the number of correct guesses, subtracting the incorrect.

<table>
<thead>
<tr>
<th>Ranking interval</th>
<th>Correct guesses</th>
<th>Incorrect guesses</th>
<th>Total guesses</th>
<th>Based on X surveys</th>
<th>Correct %</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 800</td>
<td>95</td>
<td>105</td>
<td>200</td>
<td>9</td>
<td>47.5 %</td>
</tr>
<tr>
<td>800 - 1200</td>
<td>76</td>
<td>84</td>
<td>160</td>
<td>5</td>
<td>47.5 %</td>
</tr>
<tr>
<td>1200 - 1600</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0 %</td>
</tr>
<tr>
<td>1600 - 2000</td>
<td>85</td>
<td>75</td>
<td>160</td>
<td>6</td>
<td>53.1 %</td>
</tr>
<tr>
<td>2000+</td>
<td>40</td>
<td>20</td>
<td>60</td>
<td>3</td>
<td>66.7 %</td>
</tr>
</tbody>
</table>

Table 2: Rankings connected with their performance in the Turing test.
5.3.1 All data

The Turing test gathered in total 1212 guesses from participants, presented in figure 13. Out of these guesses, 633 were correct, resulting in a proportion of approximately 0.522 correct guesses. This data was evaluated using an exact Bernoulli trial in the programming language R, the approach can be seen in figure 14.

![Query results](image)

Figure 13: Query results, collecting sum of correct and incorrect selections from turing test.

Null hypothesis \( H_0 : p = 0.5 \) There is no significant difference between the participants guessing-accuracy and random chance.

Alternative Hypothesis \( H_1 : p \neq 0.5 \): The participants guessing-accuracy is significantly different from random chance.

\[
P (X = 633) \sim \text{binom}(1212, 0.5) = 0.1279 \\
0.1279 > 0.05 \\
95\% \text{ confidence interval} : [0.4937183, 0.5507279]
\]

Figure 14: Showing the result from the exact Bernoulli trial in R, based on all data.

Based on the calculations, with a significance level of 0.05, we cannot determine that there is a significant difference between the participants guessing accuracy and chance.

The breakpoint for significance, with a significance level of 0.05 were 641 correct guesses.

5.3.2 Rank specific evaluation

The measurable number of guesses performed by a player with ranking 1600 or above, were in total 220, with 125 correct. Out of these 220, 60 guesses came from players with a 2000+ ranking. In Table X, a noticeable pattern emerges, indicating that higher-ranking players tend to exhibit enhanced correct guess percentage. The result of performing an exact Bernoulli trial on this section of data is shown in figure 15.

\[
P (X = 125) \sim \text{binom}(220, 0.5) = 0.05031 \\
0.05031 > 0.05 \\
95\% \text{ confidence interval} : [0.4999095, 0.6345916]
\]

Figure 15: Showing the result from the exact Bernoulli trial in R, based on 1600+rankings.

Based on the calculations, with a significance level of 0.05, we cannot determine that there is a significant difference between the participants guessing accuracy and chance.

The breakpoint for significance, with a significance level of 0.05 were 126 correct guesses.
5.3.3 Feedback statistics

The feedback surveys gave the following result, presented in table 3,4,5. Table 3 contains the averages of each statistic based on all submissions. Score counts the number of correct guesses submitted incorrect guesses. The questions regarding Confidence, knowledge, pattern and realistic were stated as shown in section 5.2, figure 11.

<table>
<thead>
<tr>
<th>Knowledge</th>
<th>Confidence</th>
<th>pattern</th>
<th>Realistic</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.21</td>
<td>2.26</td>
<td>2.65</td>
<td>3.94</td>
<td>1.18</td>
</tr>
</tbody>
</table>

Table 3: Feedback statistics based on average, over all ranking surveys.

<table>
<thead>
<tr>
<th>Ranking</th>
<th>Score</th>
<th>Confidence</th>
<th>Pattern</th>
<th>Realistic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-0.4</td>
<td>1.73</td>
<td>1.93</td>
<td>3.8</td>
</tr>
<tr>
<td>2</td>
<td>-1.6</td>
<td>2.22</td>
<td>2.89</td>
<td>4.22</td>
</tr>
<tr>
<td>3</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>4</td>
<td>1.67</td>
<td>3.17</td>
<td>4</td>
<td>4.17</td>
</tr>
<tr>
<td>5</td>
<td>6.67</td>
<td>3.67</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 4: Feedback statistics, showing individual ranking averages.

<table>
<thead>
<tr>
<th>Realism /Rank</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 - 800</td>
<td>2st</td>
<td>0st</td>
<td>3st</td>
<td>4st</td>
<td>6st</td>
</tr>
<tr>
<td>800 - 1200</td>
<td>0st</td>
<td>0st</td>
<td>1st</td>
<td>5st</td>
<td>3st</td>
</tr>
<tr>
<td>1200 - 1600</td>
<td>0st</td>
<td>0st</td>
<td>0st</td>
<td>0st</td>
<td>0st</td>
</tr>
<tr>
<td>1600 - 2000</td>
<td>0st</td>
<td>0st</td>
<td>1st</td>
<td>3st</td>
<td>2st</td>
</tr>
<tr>
<td>2000+</td>
<td>0st</td>
<td>0st</td>
<td>3st</td>
<td>0st</td>
<td>0st</td>
</tr>
</tbody>
</table>

Table 5: Feedback statistics, showing number of submitted results regarding realism.
6 Discussion & Conclusion

In this section, I, as the writer, will delve into my personal reflections and realizations that unfolded both during, and after the project's completion. Furthermore final remarks and conclusions regarding the research questions will be discussed.

6.1 Result analysis & Fulfillment of research question 2

The data gathered from the Turing test, presented in section 5.3, provided a good insight in the models generative capabilities regarding realistic chess-positions. Based on this data, we could not statistically determine a significant difference between chance and the participants ability to distinguish the generated positions from the real. However, this was only the general case, including inexperienced players, which could arguably create an unfair representation. But for arguments sake, given this statistical analysis, the model can to some degree fulfill the second research question of generating realistic positions for the “average person”. On the other hand when looking deeper, and taking the participants' game knowledge in consideration, we started seeing a clear pattern. This seems to show us that the participants ability to identify the generated positions has a significant relation to their rank. This was not something unexpected, rather the opposite. Experienced chess players with an above average rank, often have really good pattern recognition from playing the game. In addition to this, the experienced players have most undoubtedly, a heightened comprehension of what types of occurring patterns are more probable in a game. The experienced player can base their guesses upon this knowledge, while the less experienced player possesses a more restricted foundation for assessing the positions. Based on these results I personally have hard time arguing to what extent the model can generate realistic, game-like chess-positions as if they were to occur in a real match. The model can with ease generate valid positions according to game rules, but the evidence shows that experienced players seem to notice a pattern.

If we instead try to base the argument on survey results presented in section 5.3.3, table 3, the average perceived realism of the positions from all rank-intervals greater than 3. The users were presented with the following description of how to score, “1:Not at all, 5:They look fully realistic”. Here we can see that all rankings except for 2000+, rated the level of realism close to fully realistic, while 2000+ rated it straight in the middle. Table 5 presenting the number of submitted degrees of realism shows that one third of the people within 1600-2000 rank submitted fully realistic, which supports its capabilities. This metric is however very subjective, and unless the values showed to be averaging close to 5, it could mean just about anything, once again making it a tough question.

In hindsight I would have implemented this question as a text response, allowing the users to personally describe their experience, instead of a number that's difficult to interpret. In clarification I would argue we lack sufficient data to say whether the model can generate realistic gamelike positions, but it most certainly shows promise.
6.2 Fulfillment of research question 3

Regarding the last research question, one could argue both ways. The planned constraints for the model were predefined tokens for the model, as initial input for generation. This was fulfilled partly, by the models capability in taking predefined piece-position tokens as input for generation, but at the same time, empty positions are not a natural constraint for the models generative pattern, rather a built-in re-generation explained in section 5.1. However, the question is phrased as “To what extent do the generated positions conform to the provided constraints”, more referring to how the model conforms to given constraints. Based on this, the model fully conforms to the constraint of fixed piece-position tokens, where the generated sequence always contains the predefined tokens. However, if the given set of fixed tokens leads to an illegal position, no matter what tokens the model generates, the model won’t return a sequence, being unable to reach a valid position.

If we include fixed empty spaces and score intervals, the answer is less clear. Since the model method of achieving these constraints is by re-generation, not by understanding, there is a risk the models output won’t conform to the constraint, only being allowed a certain number of tires.

6.3 Data reliability & missed opportunities

One of the arguably most unreliable parts of the project is the size and quality of the gathered data. Firstly the amount of data gathered is not as large as hoped with a number of 1212 guesses, where only 600 can be connected to user surveys. In addition to this, there were no limits to the amount of tests taken per person, leading to some people performing the test several times, trying to beat their previous score. This is purely a good thing from the perspective of improvement, providing an opportunity to comprehend the potential learnability in discerning the generated positions. This understanding could then act as groundwork for the further development, counteracting a possible flaw. This aspect was however not initially considered, leading to the absence of a direct method for tracking or enabling users to provide feedback, aside from the few users contacting and describing their experience.

Possible additions to the database were thought of during the testing-phase, one being the time spent performing the test. This would give us a better insight to which actually took their time evaluating each position, and which based their guesses on a quick glance. This time-measurement could also be implemented for each individual position-pairs, allowing us to analyze which positions were quickly assessed as generated, possibly containing errors, or obvious patterns.

6.4 Direct feedback from users

Some users contacted me after the tests to provide me with extra feedback. These comments were very mixed in character, some saying it was impossible to tell which was which, but some experiences suggested a possible pattern of recognition. This described pattern was that some generations contained a piece positions combination that were suspiciously poor, where for example two white pieces were left hanging in strange ways. This would be really interesting to evaluate, to get a selection of testers, informing of this possible strategy and see if their result improves in order to see if this pattern has any merit.
6.5 Vocabulary design and its effect

Early on in the project we had to decide on the models vocabulary, in which we decided to have the possible combination of piece-position values, a PAD token, as well as start and end tokens. This led to an issue down the road. When it comes to the constraint of forcing fixed positions in the generation it comes naturally, however the option to fix empty positions was overlooked at first. This constraint was far less natural for the model in how it ended up being designed. There was no actual way for the model to tell whether a position was empty other than checking if the so far generated sequence contained a piece on said position. This made the constraint far less effective compared to fixed positions, requiring regeneration until satisfactory instead of giving the model it as an input.

There are, however, pros and cons to this approach. Skipping the “empty” token reduces our sequence length/block size to half the size. It could also be the case the training required is reduced, the argument being the “empty” token would be so frequent, that it would take a larger amount of training, just getting past the barrier of mostly generating empty positions. It would however be very interesting to see the effects of including this token, since it would lead to a much more natural way of implementing the constraint.

6.6 Choice of database

The choice of using Firebase and Firestore came with one problematic drawback. The service is free, and was extremely easy to work with, BUT, it had a daily limit regarding reads and writes. This became an issue the first few days of collection data, reaching the quota totally halted the data collection, making the website unable to load positions from the database. On the other hand I did not know of any other alternatives, and one could possibly argue its easy implementation compensated for these troubles.

6.7 Future work

In this section, I will discuss the future work that I've thought of while working on the project. There are a lot of areas in which improvement can be made, and possible interesting directions for further investigation.

6.7.1 Further training & testing

This work has barely scratched the surface regarding its potential. A clear area for further work is to train the model on a larger amount of data, both regarding number of epochs and size of the dataset. The training was certainly one of the parts that required the most amount of time, creating a bottleneck for the projects potential. Firstly a substantial part of the project timeline got spent developing the model, in combination with searching for a satisfactory hyper-parameter combination. Following this, the window for training was tight, due to the necessity of securing enough time for creating a Turing test and data collection. This resulted in the model being trained on a dataset containing around 5 million positions, the model was then trained on the full dataset in 6 epochs. This amount of training led to acceptable generations, and the rate of improvement showed vague patterns of stagnation, prompting the project to proceed to the next stage. Allowing the model to train on a larger amount of data, and further epochs could very likely show improvement, as well as making deeper combinations of hyper-parameters viable.
6.7.2 Develop a model for GANS

At the start of this project a theoretical investigation of the two architectures began in order to select which candidate had the most promise to achieve the goal of the research questions. After reading up on these architectures and discussing with Noctie we came to the conclusion that the transformer decoder-only was the more logical approach for numerous reasons described in section 3.2. Nevertheless, it would still be interesting to compare the two and investigate GANs capabilities of generating chess positions.

6.7.3 Investigate “empty” in vocabulary

The design choice of not including the “empty position” in the vocabulary were discussed in section 6.5. Implementing the “empty” token would make constraining empty coordinates a natural task for the model, enabling it to directly generate positions with fixed empty coordinates instead of exhaustively re-generating until the constraint is fulfilled. It may however come with its own pros and cons, and would be very interesting to investigate.
List of references


Google Colaboratory URL, https://colab.google/

Google Firebase URL, https://firebase.google.com/

Lichess URL, https://lichess.org/

Pytorch URL, https://pytorch.org/