Exploration of Chess Puzzle Generation Using Variational Autoencoders

Aaron Gan

December 25, 2024

Abstract

Chess Puzzle Generation based on real puzzles is a novel topic that has rarely ever been discussed. This paper introduces a traditional approach to generating images and applies it to the game of chess. It uses a Variational Autoencoder (VAE) with a strong focus on the loss function and how changing its parameters can affect the model's behavior. To start, I used an open-source database with millions of puzzles, and I was able to extract and process about five thousand puzzles to train my model. After that, I implemented a standard VAE model, its main components being an encoder, latent space, and decoder. Through principal component analysis, I was able to visualize the latent space by reducing its dimensions and display some reconstructed images from input data. I performed some hyperparameter tuning to handle a large amount of variability in weights, rates, and losses for all components of the model. I utilized various techniques such as depth-first search to create functions that could interpret the generated positions which were used in the loss function as metrics for the model's prediction. In the end, I was able to showcase the difficulty in generating chess puzzles and reveal the large limitations that occur from such a complex game. I conclude with a few remarks about future prospects and how to continue with the knowledge I gained. Overall, this project was a fun way for me to challenge myself in learning more complex models and apply them to something I am passionate about.

Contents

Introduction

Chess is a timeless game, yet even in modern times, it continues to advance further than ever expected. In fact, its popularity might currently be at its highest point. The combination of the COVID-19 lockdown, the success of the booming chess movie "The Queen's Gambit," and online platforms featuring chess streamers has breathed new life into the game [1]. With a wave of novice chess players joining the community, an efficient way to learn chess becomes crucial. This is where chess puzzles come into play; not only are they effective for training tactical ability, but they are also enjoyable and mind-expanding [2].

Although billions of puzzles have been developed over centuries, acquiring fresh puzzles can be challenging. Computer algorithms, such as those on *lichess.org*, must sift through millions of games played by individuals of varying skill levels. It's obvious that puzzle quality diminishes as these puzzles occur in anyone's game. Conversely, human-crafted puzzles, though much higher quality, demand substantial time and are often inspired by pre-existing puzzles. The conventional approach to puzzle creation is becoming obsolete because, with the progress of machine learning, puzzle generation should no longer be a repetitive task, but rather a creative and dynamic process reserved for Artificial Intelligence to learn off of. One type of model that suits this type of creativity is a generative model. Generative models have been implemented before, but most have centered on generating real-world objects like clothing, cars, or the well-known *MNIST* handwriting digits. There have been very few if any, instances of generative models being applied to chess puzzles. However, the potential for these models to generate new and challenging positions should not be underestimated.

To clarify my objective in generating chess puzzles, I'll start by defining what a chess puzzle is. Typically, chess puzzles present a position to a human solver, with a specific sequence of moves leading to a winning outcome (such as checkmate or material gain). They often involve innovative solutions, like sacrificing a piece, although simpler puzzles might involve more straightforward solutions. Puzzles encompass various themes such as zugzwang, forks, skewers, and more.

Through the 8-week summer program, SPARK, at Northwestern University, I've been fortunate to explore approaches to this challenge and comprehend the underlying concepts of machine learning and neural network models. Specifically, I focused on a particular model, Variational Autoencoders, studying their architecture and customization for chess puzzle generation. My priority is using historical puzzle data to reconstruct new puzzles with similar themes.

1 Part I: Data Exploration

1.1 Introduction

Numerous varieties of chess puzzles exist, consisting of a wide array of themes, difficulties, and compositions. While searching for an appropriate dataset, I prioritized the significance of labels over just the puzzles themselves to make the process easier. Given that neural networks, particularly Variational Autoencoders, demand a substantial amount of data, my aim was to acquire an extensive dataset to facilitate the process.

1.2 The Dataset

Fortunately, it didn't take much time to find the popular online chess website **lichess.org**, which is open-source and provides access to its puzzle training database. This dataset [3], consists of 3,412,058 chess puzzles, each labeled with the following features.

1.3 Labels

Column Name	**Description**
PuzzleId	The lichess puzzleId for reference
FEN	The initial position represented in Forsyth–Edwards Notation
Moves	The solution or correct sequence of moves to the puzzle
Rating	The difficulty of the puzzle represented by a four-digit number
Popularity	A popularity metric represented by the following formula: 100 * (upvotes - downvotes)/(upvotes + downvotes)
NbPlays	The number of times the puzzle has been played
Themes	The type of puzzle labeled with multiple tags such as Endgame, Middlegame, Fork, Skewer etc.

Table 1: Database Columns from Lichess Database [3]

1.4 Data Processing

Given the substantial size of the database, I opted to choose the top 5000 white puzzles. This selection was determined by a metric that combines both popularity and the count of plays. To save time, puzzles exceeding 13 moves in length were omitted. The metric utilized for puzzle scoring is outlined below:

$$P_S = 100 \cdot P + 0.005 \cdot N_B$$

The variable P_S denotes the puzzle score, where P stands for the puzzle's popularity, and N_B signifies the number of plays it has received. The puzzle's solving perspective (white or black) was determined with a Stockfish 16 evaluation of its initial position. Only the white puzzles were selected, allowing the model to focus solely on one side. Furthermore, puzzles were categorized into **Endgame** and **Middlegame** labels. This categorization was implemented to prevent any confusion between these two distinct types of puzzles. Endgame puzzles typically involve fewer pieces, resulting in fewer potential positions, whereas middlegame puzzles have more complex lines of play. In the end, both types offer the potential for innovative generation.

Next, I converted the images into a one-hot array format, wherein each index represents a piece or an empty square. Thus, accounting for both black and white pieces as well as the vacant square, there are a total of 13 indices in the array. This structure, combined with an 8x8 array representing the chessboard, results in an input data shape of 8x8x13. Before moving on, I stored these positions as NumPy arrays in two distinct folders, named "middlegame" and "endgame," to be used later. With this groundwork laid, we can now proceed to the next phase: model development.

2 Part II: Model Development

2.1 Generative Models

Generative models have emerged and gained a lot of popularity in the past few years. They offer a range of uses in generating music and images and can be helpful for distinguishing fake or anomalous data. More specifically, Variational Autoencoders (VAE) are popular for applications in both unsupervised and supervised learning, being useful for understanding patterns and the data's structure. VAEs use a neural network with or without convolutional layers, and can be represented by the following image:

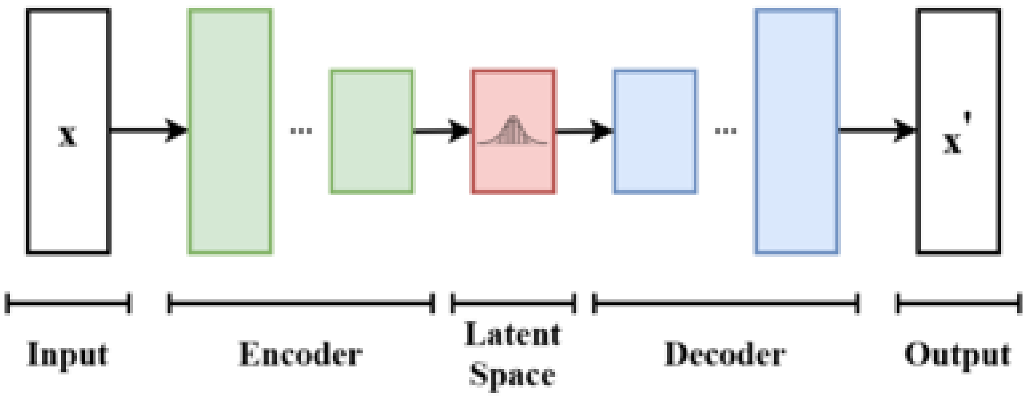


Figure 1: Variational Autoencoder Architecture [4]

2.2 General Variational Autoencoder Architecture

Variational Autoencoders are distinct in several ways. They are probabilistic models, which means they don't yield deterministic outputs like traditional feedforward neural networks. Instead, they generate probability distributions of outputs. For instance, if the input samples are images, the output would generate multiple potential outcomes along with their likelihoods. Variational Autoencoders utilize a neural network with multiple layers, each representing the encoder, decoder, or latent space of the model. The encoder typically takes input data, such as an image, resulting in a 3-dimensional array input. It works to reduce the input data's dimensionality to a lower-dimensional latent space. This is situated in the middle section of the model, as seen in Figure 1. This latent space serves as a reduced representation of the input data, saving resources and helping diversify the generation process. The decoder aims to reconstruct an image based on a point within the latent space. The latent space often follows a distribution pattern, frequently a Gaussian distribution resembling the bell curve shown in Figure 1, which we will explore further in section 2.3. Latent spaces can also be visually interpreted in cases of classification problems. For instance, in the well-known MNIST handwriting dataset, where each digit is classified and labeled, clusters can differentiate digits, as shown below.

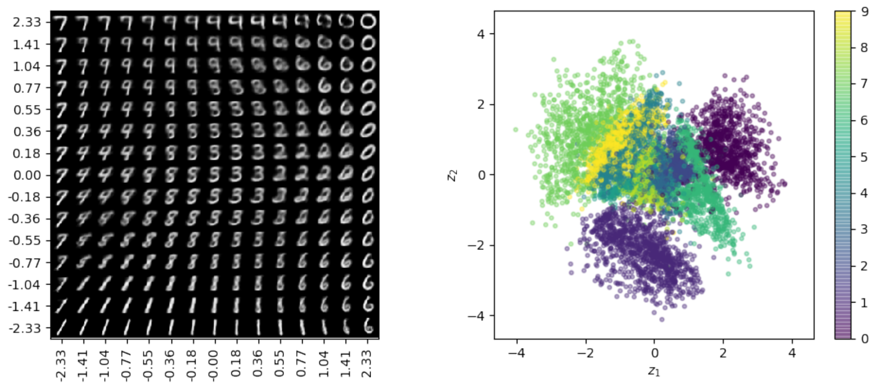


Figure 2: Variational Autoencoder Latent Space for MNIST handwriting dataset [5]

2.3 The Reparameterization Trick

The reparameterization trick proves valuable in guaranteeing a differentiable function. VAEs utilize stochastic sampling to introduce randomness within the latent space. This means that they randomly select latent space points. However, optimizing a model using stochastic sampling results in a non-differentiable equation, making it impossible to optimize. The reparameterization trick overcomes this obstacle by isolating the random sampling from the deterministic values. The equation for a latent space point without employing the trick can be expressed as follows:

$$z = f(\mu, \sigma) \tag{1}$$

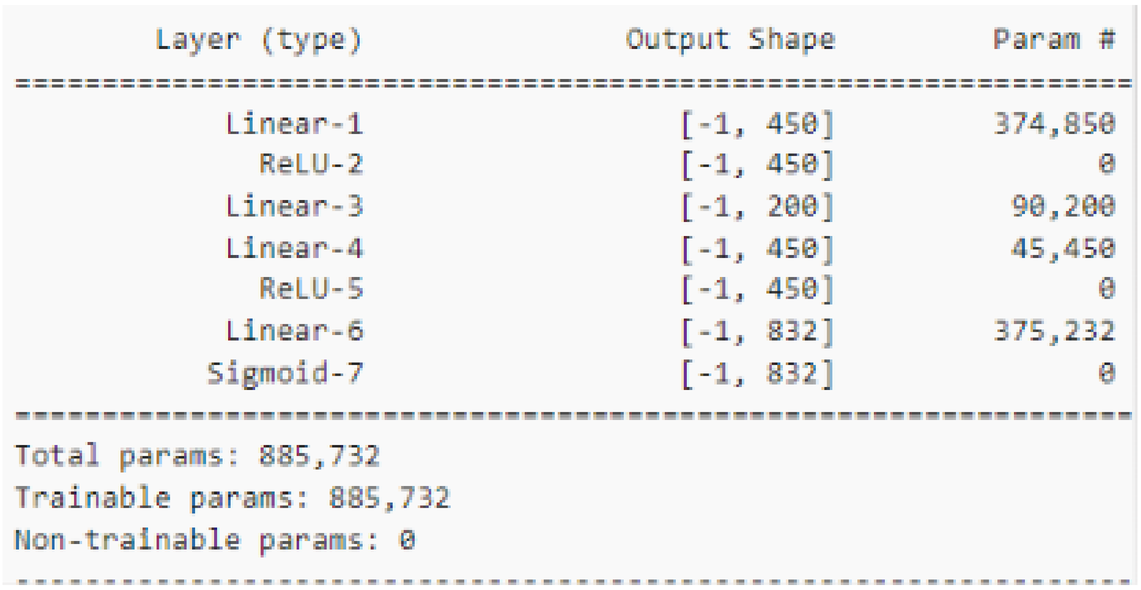
In this scenario, μ and σ represent the mean and standard deviation of the latent space distribution, and random sampling takes place external to this function. Since the latent space might not strictly adhere to the Gaussian Distribution, we can introduce a new function that leverages random sampling and utilizes a Gaussian Distribution as the source of randomness. This ensures the presence of a differentiable function. Consequently, the updated function becomes:

$$z = g(\mu, \sigma, \varepsilon) \tag{2}$$

In this instance, ε represents the randomly sampled value drawn from the Gaussian Distribution ($\mu = 0$, $\sigma = 1$). In essence, we utilize the Gaussian Distribution as a foundational source for sampling within our designated latent space.

2.4 ChessVAE Architecture

For the VAE model in my project, I used a standard structure for VAEs. With two layers for encoding and two layers for decoding, I balanced the number of parameters and trainability to ensure a complex but fast model.



```
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1                  [-1, 450]         374,850
              ReLU-2                  [-1, 450]               0
            Linear-3                  [-1, 200]          90,200
            Linear-4                  [-1, 450]          45,450
              ReLU-5                  [-1, 450]               0
            Linear-6                  [-1, 832]         375,232
           Sigmoid-7                  [-1, 832]               0
================================================================
Total params: 885,732
Trainable params: 885,732
Non-trainable params: 0
----------------------------------------------------------------
```

Figure 3: ChessVAE summary

As shown in Figure 3, the model comprises a total of $885,732$ parameters. Please note the input and output shapes of each layer. The initial layer takes an 8x8x13 array as input, resulting in a shape of $[-1, 832]$. The data undergoes compression to a latent space of dimension $[-1, 450]$, then $[-1, 200]$, then $[-1, 450]$, and finally $[-1, 832]$, which aligns with the input. The latent space dimension is customizable, but due to the complex nature of recognizing complex patterns in chess puzzles, I opted for a very high latent dimension.

2.5 Loss Function

The most crucial aspect of my model is, undoubtedly, the loss function. Loss functions play a pivotal role in all machine learning models, as they are the ultimate objective the model aims to minimize. While simpler models might utilize a single term for the loss function, such as Mean-Squared Error, I needed a more intricate approach to enable the model to generate more compelling chess puzzle positions. I constructed a loss function to calculate the loss of a single input image as we will see below.

For every VAE, two crucial terms need to be balanced: the *Reconstruction Loss* (RL) and the *Kullback-Leibler Divergence Loss* (KL). These two terms evaluate the loss of a single data point or image. In simpler terms, RL measures how closely the output image resembles the input image, often utilizing a formula known as *Binary Crossentropy.* This can be expressed as follows:

$$\mathrm{RL} = \mathrm{BCE} = -\frac{1}{N}\sum_{i=1}^{N}\left(x_i \log(x_{\mathrm{recon},i}) + (1 - x_i)\log(1 - x_{\mathrm{recon},i})\right) \tag{3}$$

In Equation 3, where i represents the i-th pixel of the image, $x_{\mathrm{recon,i}}$ stands for the reconstructed output, and x_i corresponds to the input image or one data point. As evident, this provides a good gauge of how closely the input data reflects the output data. On the other hand, KL loss can be considered as the converse. It evaluates the difference between the probability distributions of the VAE's distribution and a target distribution, which, in most cases, is the Gaussian Distribution. For my specific VAE, I employed the Gaussian Distribution as the target distribution. The KL loss for the Chess VAE can be depicted using the subsequent formula:

$$\mathrm{KL} = \mathcal{D}_{\mathrm{KL}}(q(z|x)||p(z)) = \frac{1}{2}\sum_{j=1}^{J}\left(1 + \log(\sigma_j^2) - \mu_j^2 - \sigma_j^2\right) \tag{4}$$

In Equation 4, $q(z|x)$ signifies the distribution produced by the encoder, representing an input image. On the other hand, $p(z)$ designates the predefined distribution, which, in this case, is the Gaussian Distribution. Here, μ denotes the mean of the generated distribution, while σ represents the standard deviation. This formula essentially computes the disparity between the two distributions and ensures that the distribution converges towards a Gaussian Distribution. Gaussian Distributions are useful because of their simplification of computations, their smooth and differentiable nature, and their greater manageability and familiarity compared to other distributions. Lastly, I incorporated a KL annealing factor. This implies that the model initially prioritizes RL over KL, but as the model's progression unfolds, it begins to give greater importance to KL. The annealed KL weight is calculated as follows:

$$w_{akl} = \text{annealed\_kl\_weight} = \min(1.0, \text{anneal\_rate} \times \text{global\_step})$$

The anneal_rate is adjustable. The global_step refers to the step that the model is on during training throughout all steps in each batch in each epoch. This equation is useful for letting the model warmup and get to know the input data better. While only RL and KL are required for a functional VAE, to ensure diverse and interesting positions, I proposed including five other loss function terms.

$$ES = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_{ES} - 40)^2 \tag{5}$$

$$WL = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_{WL} - 1)^2 \tag{6}$$

$$SD = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_{SD} - 8)^2 \tag{7}$$

$$DP = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_{DP} - 13)^2 \tag{8}$$

$$VP = \hat{y}_{VP} \tag{9}$$

$ES =$ Empty Squares Loss.	$\hat{y}_{ES}$ is the number of empty squares in that predicted chess position.
$WL =$ Winning Lines Loss.	$\hat{y}_{WL}$ is the number of winning lines or solutions in that predicted chess position.
$SD =$ Solution Depth Loss.	$\hat{y}_{SD}$ is the depth of the deepest winning solution in that predicted chess position.
$DP =$ Distinct Pieces Loss.	$\hat{y}_{DP}$ is the number of distinct pieces in that predicted chess position.
$VP =$ Valid Position Loss.	$\hat{y}_{VP}$ is either 0 or 1, where 1 represents an invalid predicted chess position.

As shown above, the loss function for each input data batch involves computing the mean squared error between the predicted image's predicted value and a constant. This constant was chosen to guide the predicted value toward convergence. For instance, in pursuit of a vibrant position, the model penalizes significant numbers of empty squares. Similarly, the model aims for a single solution, a complex solution (with a large solution depth), and the utilization of different pieces. Lastly, a crucial validation check ensures the generated position is a valid chess position. With these elements, we can simply assign a weight to each and then sum them. Our final loss function equation is as follows:

$$\begin{aligned}\text{Total Loss} =& w_{akl} \cdot w_{RL} \cdot RL + w_{KL} \cdot KL + w_{ES} \cdot ES \\ &+ w_{WL} \cdot WL + w_{SD} \cdot SD + w_{DP} \cdot DP + w_{VP} \cdot VP\end{aligned} \tag{10}$$

Please note that all these weights are adjustable. I tuned these hyperparameters a couple of times and concluded with this combination of values for the final model:

1. $w_{RL} = 0.01$

2. $w_{KL} = 0.3$

3. $w_{ES} = 0.1$

4. $w_{WL} = 10$

5. $w_{SD} = 0.04$
6. $w_{DP} = 0.1$
7. $w_{VP} = 100$

Each of these components is valuable in assisting the model to generate diverse positions while keeping the original image as a stable foundation. However, while the use of these components might have seemed promising initially they could have posed a challenge to the model's capacity, which we will explore in more detail in section 3.

2.6 Model Summary

Complex challenges need equally intricate solutions, and the ChessVAE model certainly falls under this category. To achieve optimal results, even with a high-performance computer, it will take several days to complete the necessary number of epochs for the loss function to converge toward zero. Furthermore, due to the extensive range of adjustable parameters, finding the optimal combination of weights, rates, sizes, and so on—where the model aligns seamlessly—is a challenging task. I have provided a list of potential parameters that could be modified during the model development phase.

1 Winning lines weight
2 Depth weight
3 Valid position constant weight
4 Empty squares weight
5 Distinct pieces weight
6 KL divergence weight
7 RL loss weight
8 # of layers and nodes encoder
9 # of layers and nodes decoder
10 Latent space dimension
11 Batch size
12 # of epochs
13 Line depth
14 Learning rate
15 KL Anneal Rate

As indicated, all these values are important for the model's success. Altering these parameters will certainly yield different results. As we'll explore in the next section, the model is sensitive to changes and is likely to produce some faulty results.

3 Part III: Results

3.1 Visualization Techniques

Visualizing generative models is a straightforward process. In a VAE model, the generated images can be compared with the original images, allowing the human eye to perceive their similarity. For chess puzzles, we can assess a puzzle's realism based on its solvability, presence of solutions and their quantity, and feasibility in a real game scenario. Moreover, visualizing the latent space is possible even without specific classifications. Although not immediately interpretable, clusters of similar-colored data points generally indicate shared characteristics, prompting a desire for their grouping. Utilizing the loss function as a metric offers a means to monitor the model's performance.

3.2 Generated Images

Here are a few sample inputs (on the left) and generated outputs (on the right)

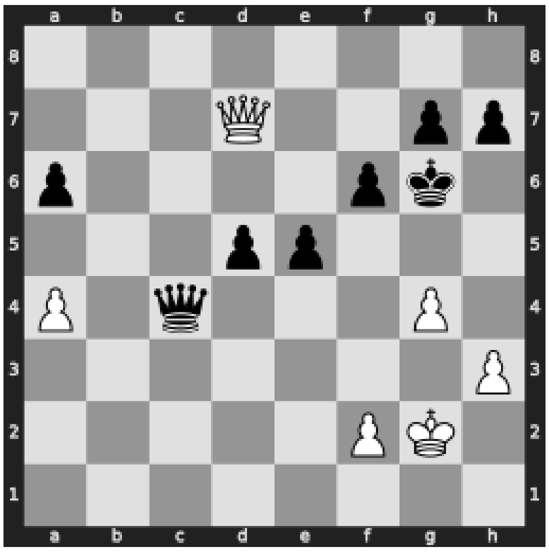


Figure 4: Input Image Epoch 1 Batch 1 Sample 125

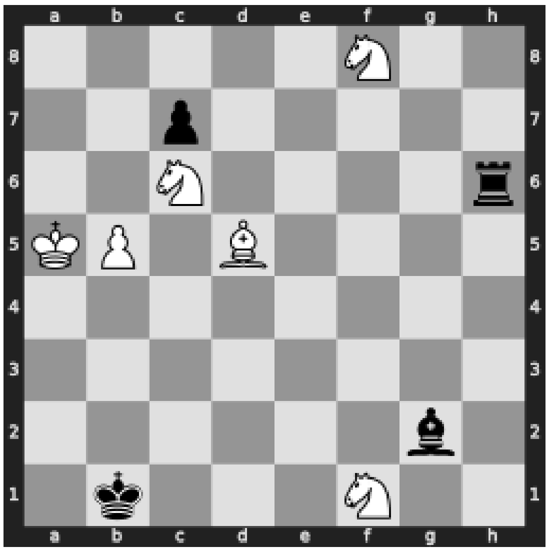


Figure 5: Generated Image Epoch 1 Batch 1 Sample 125

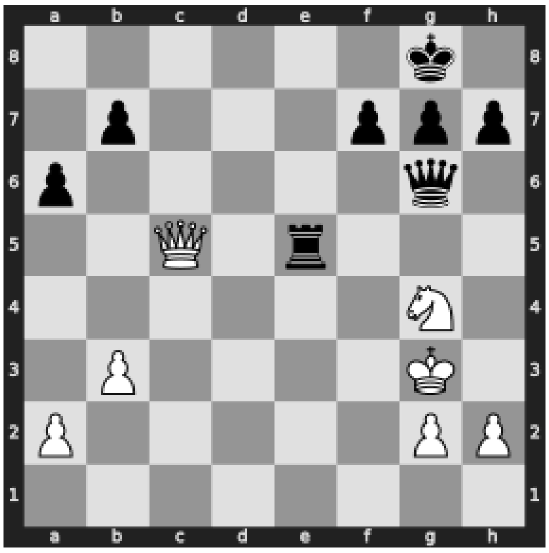


Figure 6: Input Image Epoch 1 Batch 1 Sample 172

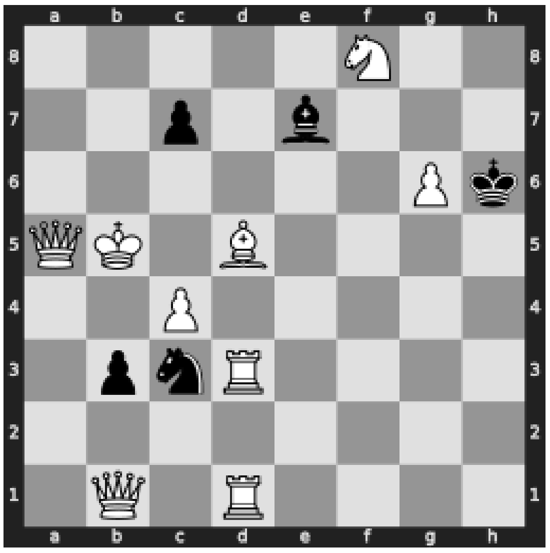


Figure 7: Generated Image Epoch 1 Batch 1 Sample 172

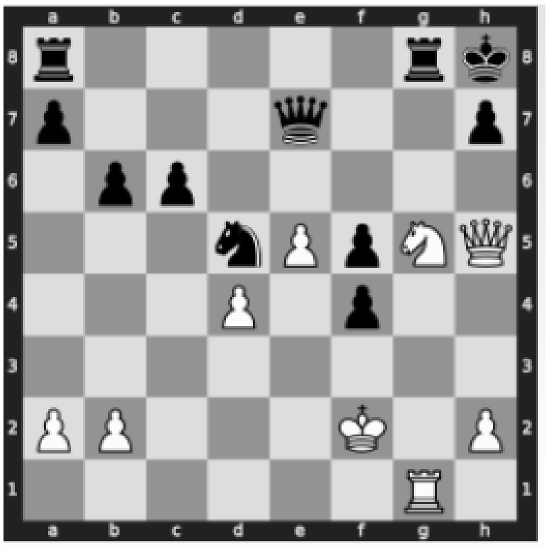


Figure 8: Input Image Epoch 1 Batch 1 Sample 204

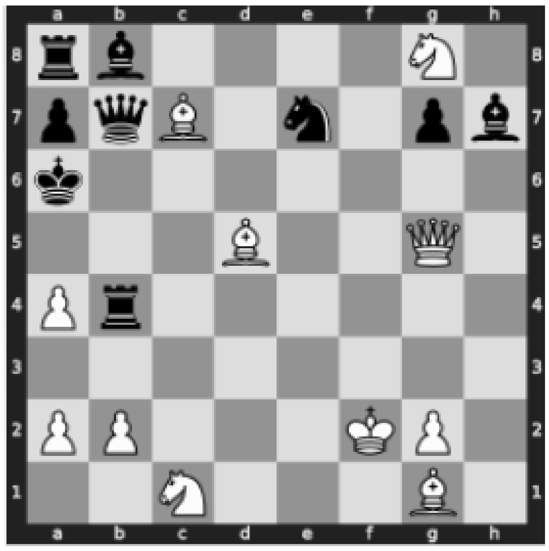


Figure 9: Generated Image Epoch 1 Batch 1 Sample 204

3.3 Latent Space

Below is the generated latent space for two epochs. Please note that Principal Component Analysis was utilized to condense the latent space dimensions into 2 dimensions, for easier visualization.

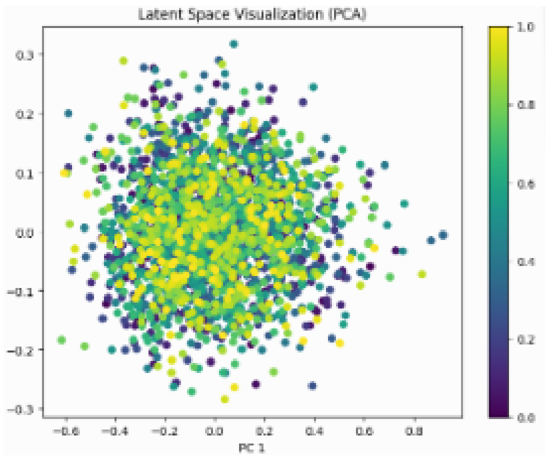


Figure 10: Epoch 1 Latent Space

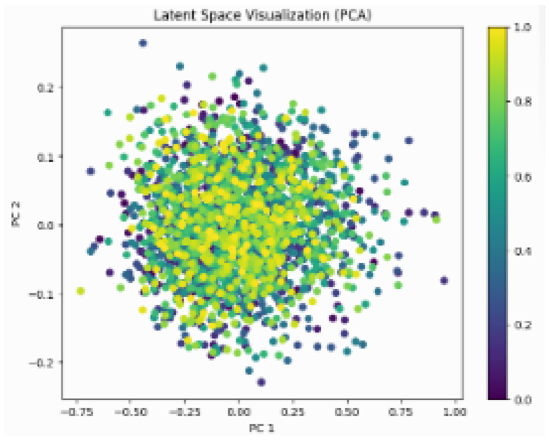


Figure 11: Epoch 2 Latent Space

3.4 Loss Function Graph

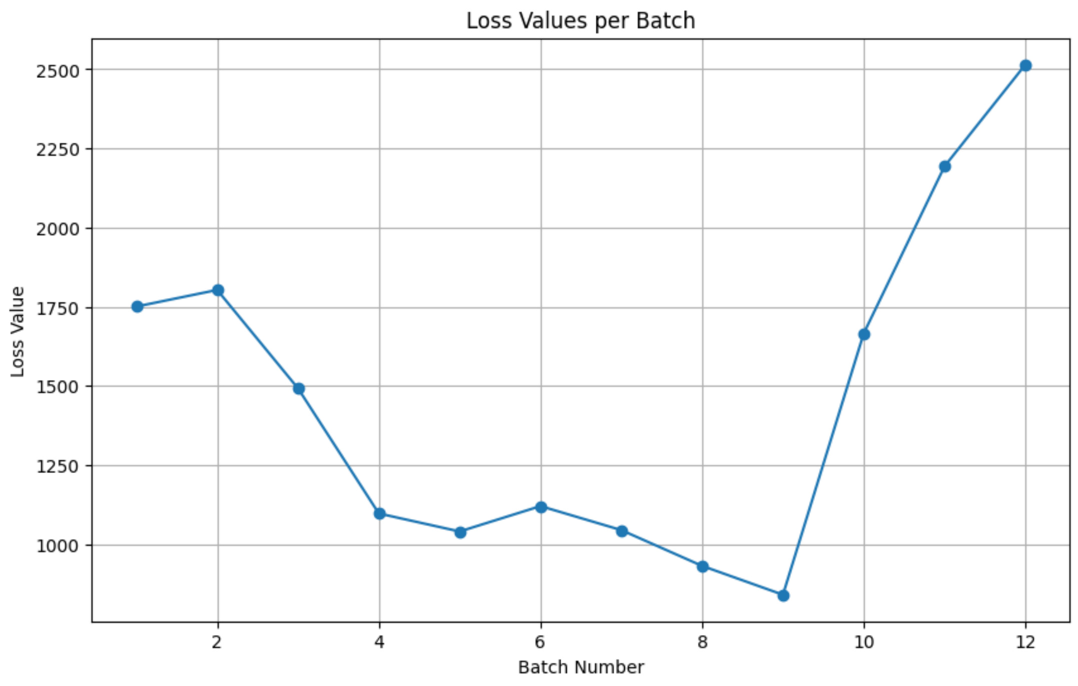


Figure 12: Loss Function Graph Over Time

3.5 Analysis

As observed, the generated images gradually begin to resemble the original image. In Figure 8 and Figure 9, some of the pieces are in the same spots, and solutions begin to form. These images were taken from the first batch, but as the number of batches that were processed increased, the images started to yield repetitive positions. For instance, the images resembled the following:

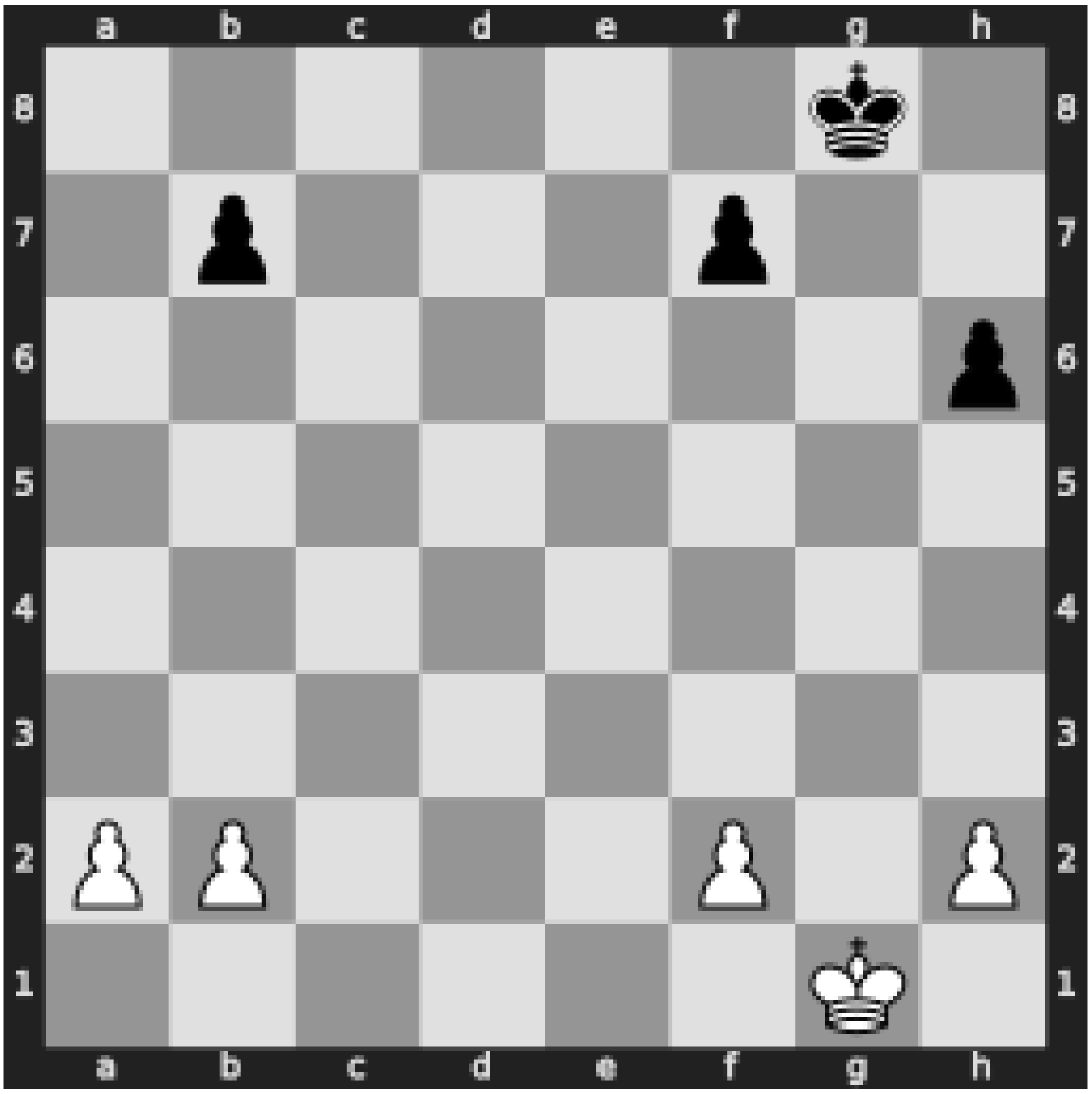


Figure 13: Generated Image Epoch 1 Batch 10 Sample 44

This may be due to several reasons that could include

1. The latent space was not complex enough for the model to understand patterns in positions.
2. The reconstruction loss weight was too high or diminished too slowly which resulted in the model favoring specific squares for specific pieces.
3. The model was not run on enough puzzles or not run enough times.

4. The loss function itself had too many large weights and distinct values that the model didn't know how to progress in lowering the loss value.

Note that there are many possibilities and these are just a few. Additionally, when examining the latent space in Figure 10 and Figure 11 it becomes apparent that it forms a cluster of points with various colors and values, which is a sign of a flawed latent space. Finally, the loss function graph as seen in Figure 12, displays an initial decrease followed by a significant increase after 9 batches. This is a clear indication that the model was confused about how to reduce the loss function at that point and the problem was too complex.

3.6 Improvements

There are many ways I could enhance my model. One solution is to spend more time fine-tuning the hyperparameters. As I mentioned earlier, there are a lot of settings that can be adjusted, and tweaking them could really change how my model performs. Another thing I could do is get a bigger dataset or a better GPU. I might also think about coming up with a better way to measure how well my model is doing, or trying out different methods for calculating losses.

I could also think about using a different type of model, like reinforcement learning, to help my model learn chess better. Adding temperature, which is a trick often used in VAEs, could help my model start off in a better place. There's also something called a β-VAE that I could try, which lets me have more control over the latent space. These ideas could help my model generate clearer and more interesting results.

If I aimed to expand on my project, there are several directions I could have explored. I might have considered incorporating labels to classify various puzzle types, delving into puzzles from the black perspective, and assigning ratings or levels of difficulty to the puzzles. There's a lot more potential for improvement and advancement in this project, beyond what I could achieve in my 8-week period at Northwestern University.

Conclusion

Chess puzzles are, quite frankly, far from simple. Constructing elegant and innovative chess positions demands a highly intricate model that surpasses the capabilities of a basic VAE. Variational Autoencoders fall short when it comes to encapsulating the complexity inherent in chess positions. Nonetheless, the discoveries made in this project carry significance. They shed light on how the model reconstructs images and adapts its predictions through the loss function. By delving into VAEs, I've gained a comprehensive understanding of their functioning and can apply this knowledge to other domains like image or even sound generation. Most importantly, this journey has challenged me to explore complex models and apply them to a subject I'm deeply passionate about. Even if the model's outcomes didn't fully meet my aspirations, I believe it serves as a valuable stepping stone for my future endeavors in further exploring this project.

During my time at Northwestern University, the bi-weekly lectures on neural networks provided me with a great background on how they work. I dived into topics such as backpropagation, gradients, and optimization, and learned mathematical applications of these terms. Moreover, I gained substantial exposure to the varied applications of projects and had the privilege of witnessing the diverse projects my peers pursued. It consistently amazes me how machine learning can find its application in such diverse ways.

As I conclude, I look forward to continued exploration within the realm of chess. Despite its seeming simplicity, chess is an ageless game renowned for its complexity. Even as Artificial Intelligence approaches near mastery, chess endures as a captivating game of boundless possibilities. I anticipate witnessing more instances of machine learning's integration into chess, and I'm hopeful humanity will keep pushing the boundaries even further.

Acknowledgements

The completion of my project has been made possible because of the invaluable contributions of everyone involved in the SPARK program at Northwestern. I am deeply thankful for my peers, whose projects inspired me to explore beyond the scope of the program. I am thankful for all the mentors who taught lectures bi-weekly and put in extensive work in preparing presentations. I appreciate the diverse topics they covered, accommodating varying levels of experience. Their balance of lectures and hands-on activities provided a real-world perspective on the applications of deep learning.

I would also like to thank my mentor Sourav Saha, who met with me in inspiring my project idea and providing any assistance I needed throughout my process throughout my experience. I am thankful for the lectures, code snippets, and examples posted weekly. I appreciate how quickly he responded when I came across any difficulties.

Lastly, I would like to thank Professor Liu for all his hard work in putting this program together and giving me the opportunity to learn about deep learning this summer. I value the program's structure, including the option for both remote and in-person participation, which allowed for a flexible experience.

Thank you to all those involved in the SPARK program at Northwestern University through the summer of 2023. I thoroughly enjoyed learning and collaborating with everyone on this journey!

References

[1] Chess is Booming. https://www.nytimes.com/2022/06/17/crosswords/chess/chess-is-booming.html

[2] Chess Puzzles: 10 Reasons to Solve Them Daily. https://thechessworld.com/articles/general-information/10-reasons-to-solve-chess-puzzles-daily/

[3] Chess Puzzle Database. https://database.lichess.org/#puzzles

[4] Variational autoencoder. https://en.wikipedia.org/wiki/Variational_autoencoder

[5] Implementing Variational Autoencoders in Keras: Beyond the Quickstart Tutorial. http://louistiao.me/posts/implementing-variational-autoencoders-in-keras-beyond-the-quickstart-tutorial/

Code

The entire code snippet for this project can be found in my GitHub repository: https://github.com/teachingtome/Sample-Code/blob/main/ChessPuzzleVAE.py