

Board Autoencoders: Foundations for a Strong Chess Engine

Jordan Zedeck

December 2024

1 Introduction

Progress in the field of computer Chess playing has historically served as a barometer of the advancements in the larger fields of artificial intelligence and computer hardware. Since Garry Kasparov’s famous defeat by Deep Blue in 1997, the gap between the best human and computer players has only grown, and the best engines in the world now outplay all humans by far.

These engines typically [4] utilize 2 primary components in order to select moves:

1. An evaluation function, which returns a heuristic scoring of the utility of a chess position for either player.
2. A highly optimized search tree which traverses future possible boards, and compares the outputs of the evaluation function on leaf nodes to gain the projected utility of selecting a move.

These components placed together allow chess engines to select the best projected move at any given Chess position. In this project, we endeavored to optimize the evaluation function component of this process. In classical chess engines, the evaluation function often consists of the combination of simple, human understandable heuristics based on discrete representations of a position. This enables a very fast static evaluation of board position. This evaluation is not very strong on its own, but becomes advantageous when used in tandem with a deep search algorithm.

Despite the unrivaled playing strength of state of the art chess engines, we question whether an extremely deep search algorithm is the best approach to high performance. Instead, we considered the idea of utilizing a deep neural network as an evaluation function, combined with a shallower search depth. Intuitively, this is much closer to how the best human chess players think, as its impossible to run 30-ply of mini-max in ones head.

As stepping stones to creating a strong engine with this architecture, we contribute the following:

1. A perfectly reservable method of constructing discrete, sparse, vector representations of chess positions specified in Forsyth-Edwards Notation. These are referred to as **TensorBoards** in our project.
2. Multiple autoencoder models capable of encoding **TensorBoards** into continuous, dense, representations of a smaller dimensionality. Networks with target dimensions of 256, and 128, were trained. Both models functioned with +90% precision and recall in encoding and decoding **TensorBoards**. We name these models **autoencoder256** and **autoencoder128**

2 Board Representation

2.1 Discrete Tensors

We first sought to create a discrete, vector representation of a board which could be used as the target for our autoencoder. We wanted this discrete representation to be directly translatable to the standard, universal Forsyth-Edwards Notation (FEN) [2] for describing positions. Thus, our **TensorBoards** directly encode the six components which comprise a FEN description of a position.

These consist of:

1. Piece configuration: Indicates what type of piece is at each square in the board. As a chess board is composed of 64 squares, with each square either empty, or having a piece that is either White or Black, and either a Pawn, a Knight, a Bishop, a Rook, a Queen, or a King. Therefore, to discretely represent the pieces on a board, we chose to encode the position using $64 * 13$ bits.
2. Active color: We encode whether it is White’s turn, or Black’s turn, for a given position, with 1 bit.
3. Castling rights: At the start of the game, as long as various criteria hold, both players have the right to perform the special ”castling” move when they are able to. However, players can lose the right to castle, either from either moving their King or Rooks, or from already having castled. We encode castling rights for each player, using 4 bits, which indicate White and Blacks’ King-side and Queen-side right to castle.

4. En-Passant target square: When a pawn performs an initial two square advance, it is allowed to be specially captured the next turn by pawns in the same row and adjacent file. Our encoding allows for 65 bits to specify any square, as well as no square being a distinct class and bit, as being the En-Passant square, although legally it may only be one of 16.
5. Half-move timer: The number of half-moves, or ply, since a capturing move or pawn advance is tracked, as a draw can be declared if this reaches 50. We encode this as a single integer value.
6. Full-move clock: A full-move timer is included, which simply tracks how many moves have passed since the start of the game. We encode the full-move clock as an integer as well.

Since our autoencoder network would require floating point inputs, these values are all converted to floats, and concatenated to give us a vector of length:

$$64 \cdot 13 + 1 + 4 + 65 + 1 + 1 = 904$$

Intuitively, this is a very verbose, and largely sparse, representation of a board, illustrating the need for an autoencoder should we seek to use chess positions as inputs to an evaluation network.

2.2 Choosing a Target Dimensionality for the Encoder

Multiple variables were considered in selecting a target dimensionality for our encoding. We decided a useful heuristic may be that if we need to represent 2^N positions, then a dimensionality of at least N must be used. This is a large theoretical underestimation, as our encoded tensor representation would not consist of binary values, but rather floating point numbers. However, in practice, we anticipated our autoencoder network could not achieve such a high degree of efficiency and performance.

First, we calculate the possible number of positions our **TensorBoard** can represent, which consist of both legal and illegal configurations.

Given the constraints that each piece-square may only belong to one of 13 classes, that there is a single en-passant square, we calculate there exists approximately (as we do not include the unbound full move timer in this calculation):

$$13^{64} \cdot 2^5 \cdot 65 \cdot 50 \approx 2.0 \cdot 10^{76}$$

Here, 13^{64} represents all piece configurations, 2^5 represents the turn and castling bits, 65 represents the possible squares, or no square, which may be specified (legally or not) as the en-passant square. The term 50 comes from the 50-turn-rule.

Given that $2^{254} < 2.0 \cdot 10^{76} < 2^{255}$, rounding up to 2^{256} , we would anticipate an encoding size of 256 to be the bare minimum required. However, since a **TensorBoard** is capable of representing much more positions than are legal or possible in a chess game, and greater even than positions which are extremely unlikely in real games, this number is much higher than the number of positions which need to be targeted.

We turn to the research of Claude Shannon [1], who is often credited as introducing the field of computer chess. Shannon roughly calculated a total number of positions as $64!/32!(8!)^2(2!)^6 \approx 10^{43}$. Using this value, we find that $2^{198} < 10^{43} < 2^{199}$.

Additionally, we found work from a similar project [4] using an autoencoder for positions with a dimensionality of 128, which functioned with high accuracy and a relatively small number of parameters. However, it should be noted that these researchers neglected to implement the half-move and full-move clocks, as well as the en-passant square. These omissions may make a smaller representation more feasible.

We decided on training **autoencoder** with both dimensionalities of 256, as well as 128, and would compare the performance of each.

3 The Autoencoders

3.1 Training Data

For our training data, we utilized lichess.org’s open database of games [3]. From this source, we downloaded approximately 5.8 million games, processed each position in each game into our **TensorBoard** format, and saved approximately 130 million positions for training. We utilized a small percentage of about 1 million positions as our testing set, and the rest as our training data. It should be noted that we split our testing and training data by games, and not by individual positions. We do not ensure that our training data has no repeating positions. We thought it desirable to have a real distribution of board configurations throughout games, with certain positions being often repeated throughout the training data. We believed this measure would help the model to learn real configurations faster. Additionally, we took no measure to ensure that our training data and testing data were disjoint on the individual board level, as we sought to evaluate the performance of our model on the most common positions, as well as rare ones. We believe not training, or not testing, on these common positions would be a mistake.

3.2 Autoencoders Architecture

We went through many iterations of training and testing different hyper-parameters for autoencoders, before training two final feed-forward networks with the following numbers neurons per layer:

autoencoder256: 904- > 2048- > *Droupout*($p = 0.1$)- > 1024- > 512- > 256- > 512- > 1024- > 2048- > 904

autoencoder128: 904- > 4096- > *Droupout*($p = 0.1$)- > 1024- > 128- > 1024- > 2048- > 4096- > 904

The Droupout layers were included after the first hidden layer, and were only utilized in training, not testing.

3.3 Constructing the Loss Function

Due to the nature of our target data, the **TensorBoard** format described above, we required the usage of a composite loss function to cover each element of a position. Below is a table of the different loss functions we created and summed to create the final loss for an output and target.

Function Name	Algorithm	Purpose
<code>piece_loss_fn</code>	Cross Entropy	Used to create a loss for incorrectly predicting piece labels for each of the 64 squares.
<code>piece_count_loss_fn</code>	Custom	Used to further punish the model for mis-predicting non-empty pieces where there are none. This loss term becomes more aggressive the fewer pieces of that type are on the board.
<code>turn_loss_fn</code>	Binary Cross Entropy	Used to generate a loss for incorrectly classifying whose turn it is.
<code>castling_loss_fn</code>	Binary Cross Entropy	Used to generate a loss for incorrectly predicting castling rights.
<code>en_passant_loss_fn</code>	Cross Entropy	Used to generate a loss for incorrectly classifying the en-passant square.
<code>clock_loss_fn</code>	Mean Absolute Error	Used to generate a loss from the linear difference between the model's predicted half-move and full-move clocks and the target values.
<code>illegal_pawn_loss_fn</code>	Custom	Used to create a loss for any probability generated for a pawn being in a row in which it would be illegal.

Table 1: Description of Loss Functions Used

Because all class labels, such as piece types, castling rights, and the En-Passant square, are heavily biased towards configurations and frequencies, we create class weights for the Cross Entropy and Binary Cross Entropy components of the loss function before starting training. To do this, we read in the training data before hand to generate weights for each class position, at each index.

The function `piece_count_loss_fn` was only added midway through training `autoencoder256`, as it was seen through renderings of inputs and outputs, that many pieces were being hallucinated, often very confidently. Below shows an example of this phenomenon, which was largely eliminated from the final version of the model (at the expense of a decrease in recall)

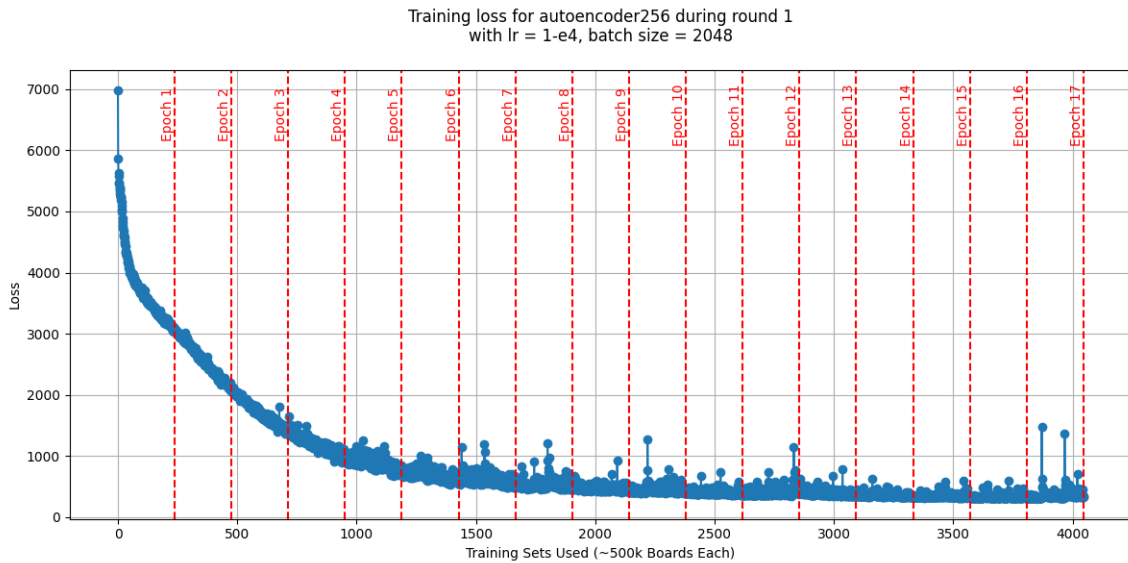
Figure 1: Example of partially trained model hallucinating pieces. Notice that recall for non empty pieces is very high, but many pieces are hallucinated. Additionally, non-piece terms have been retrieved with perfect accuracy.



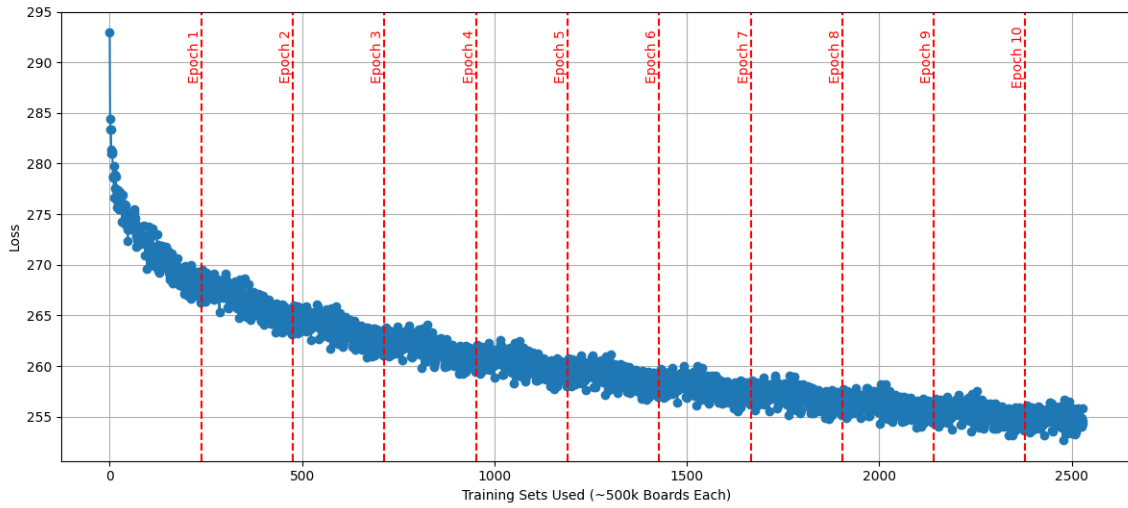
Multiple drafts of this function were written, all with the intent of increasing precision. A previous iteration of this function, `piece_count_limit_loss` was tried as well, but this did not have much of an effect on the training process. Older drafts were made as well, but were completely scrapped as they largely only hindered the model's performance. Ultimately, we believe our final version of this function helped to increase the models performance significantly, as can be seen from the graphs of our losses before and after this function is added. Loss had leveled off almost entirely, but performance had not yet reached a satisfactory level. The addition of `piece_count_loss_fn` allowed the model to continue to learn during training for 3 additional epochs.

3.4 Training

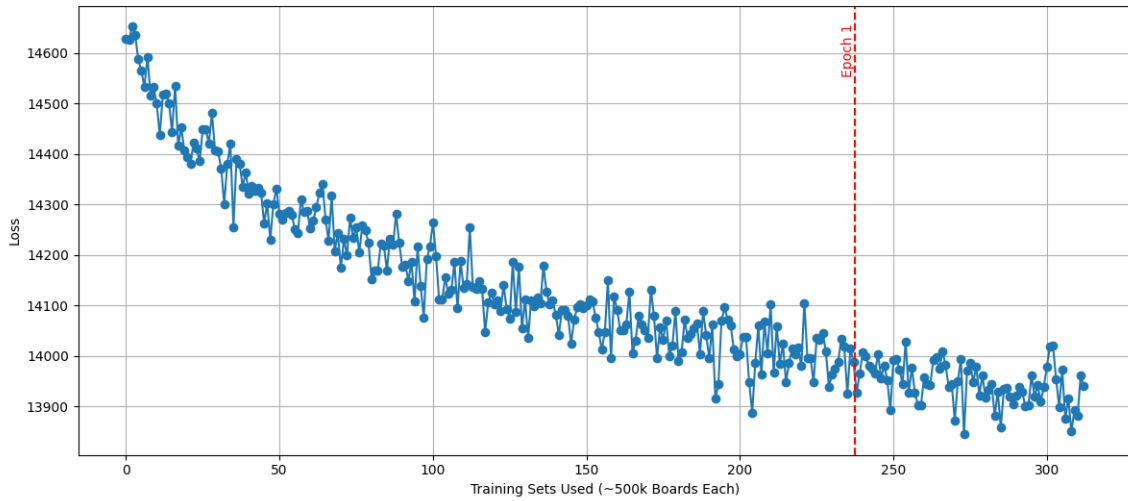
`autoencoder256` was trained for 32 epochs with interruptions to experiment with hyper-parameters such as the learning rate, as well as to implement the `piece_count_loss_fn` term. Below are graphs of the loss during 5 rounds of training. Training was concluded when this component was implemented and the model's loss over the test set no longer improved with its addition.



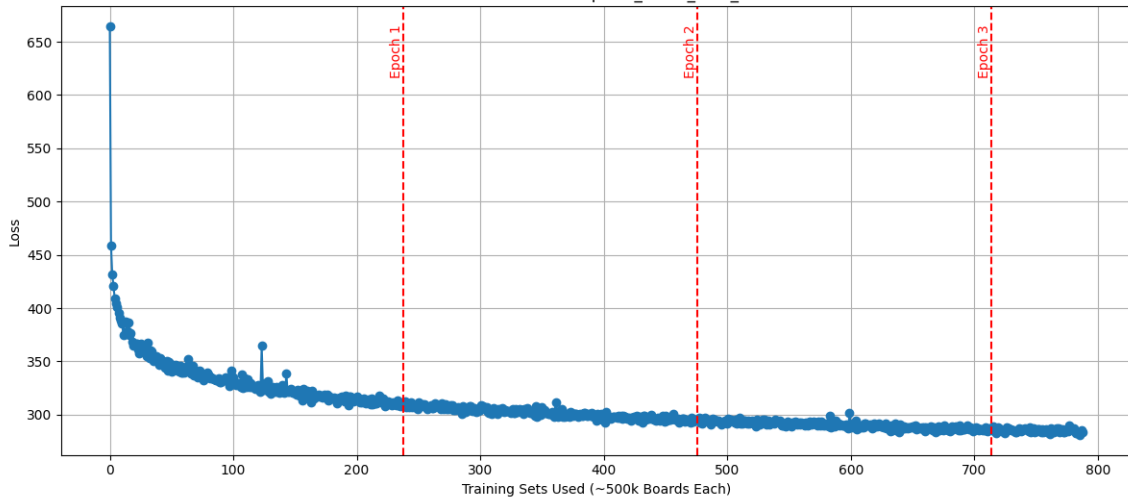
Training loss for autoencoder256 during round 2
with lr = 1-e5, batch size = 2048

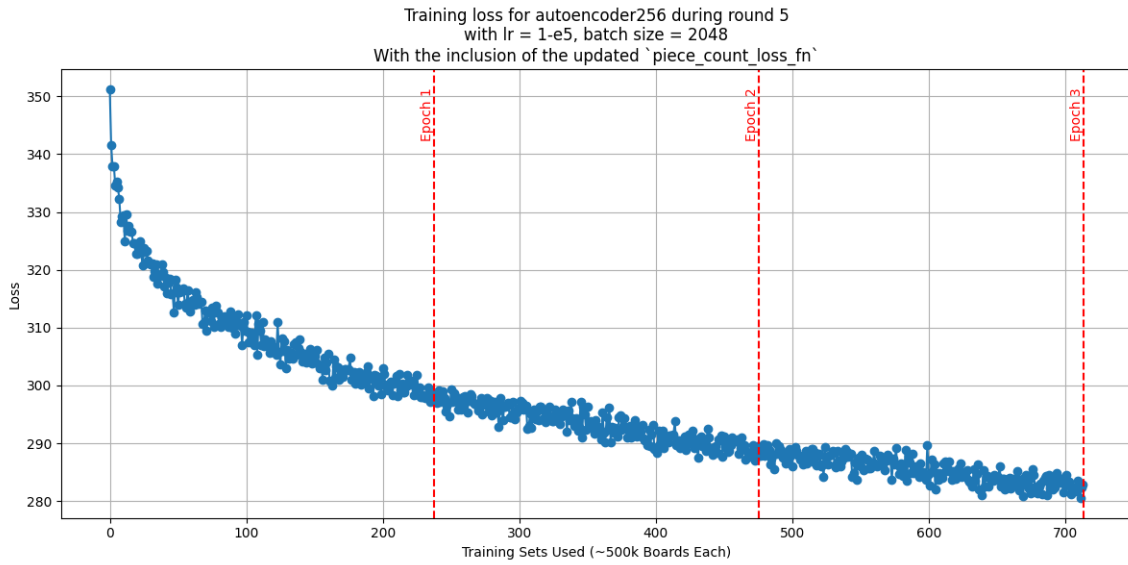


Training loss for autoencoder256 during round 3
with lr = 1-e3, batch size = 32

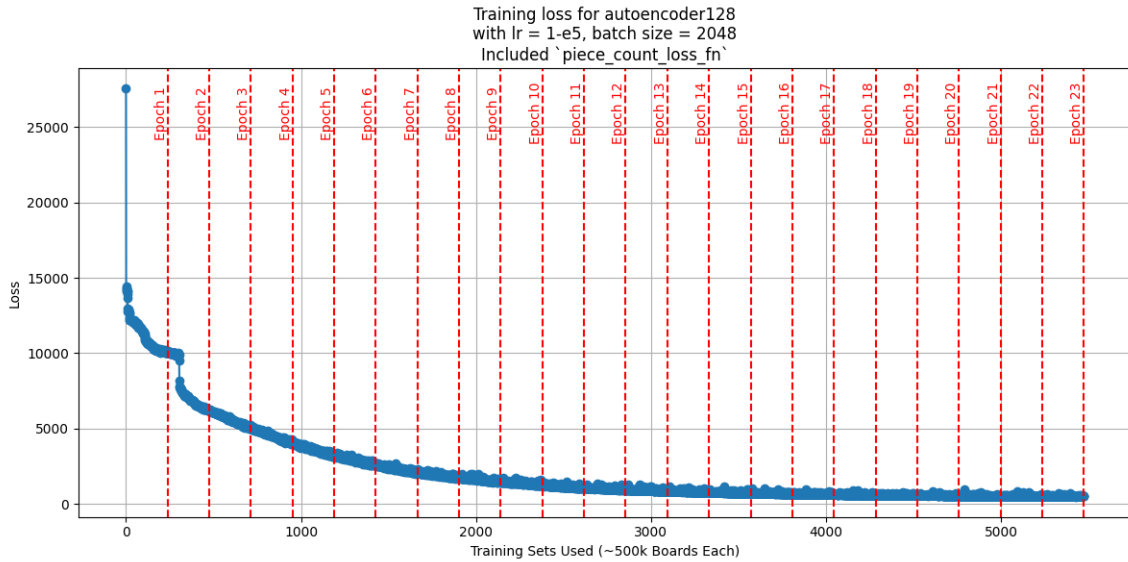


Training loss for autoencoder256 during round 4
with lr = 1-e5, batch size = 2048
With the inclusion of 'piece_count_limit_loss'



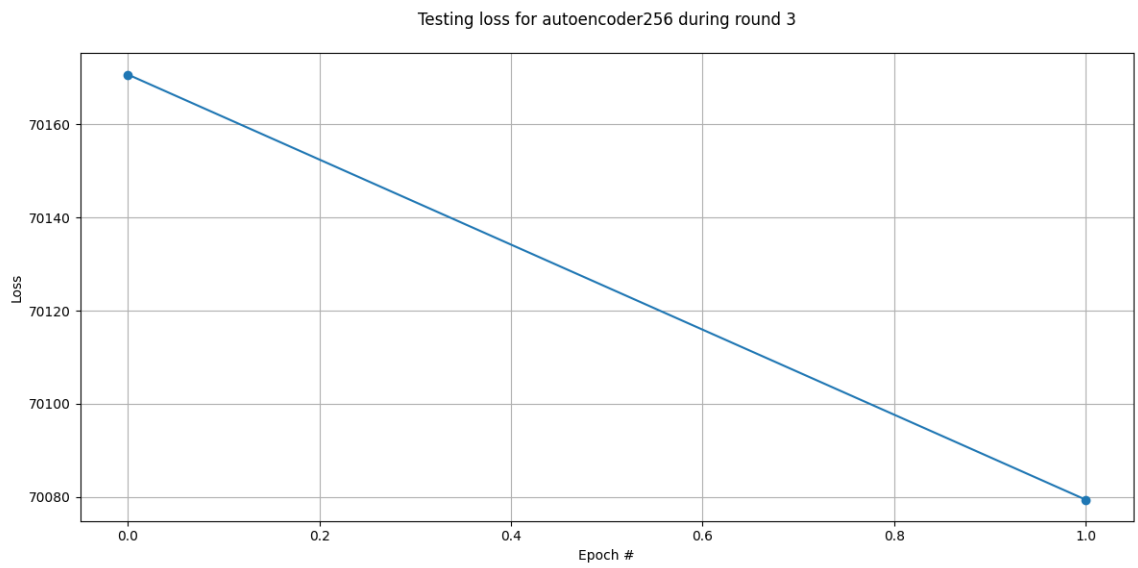
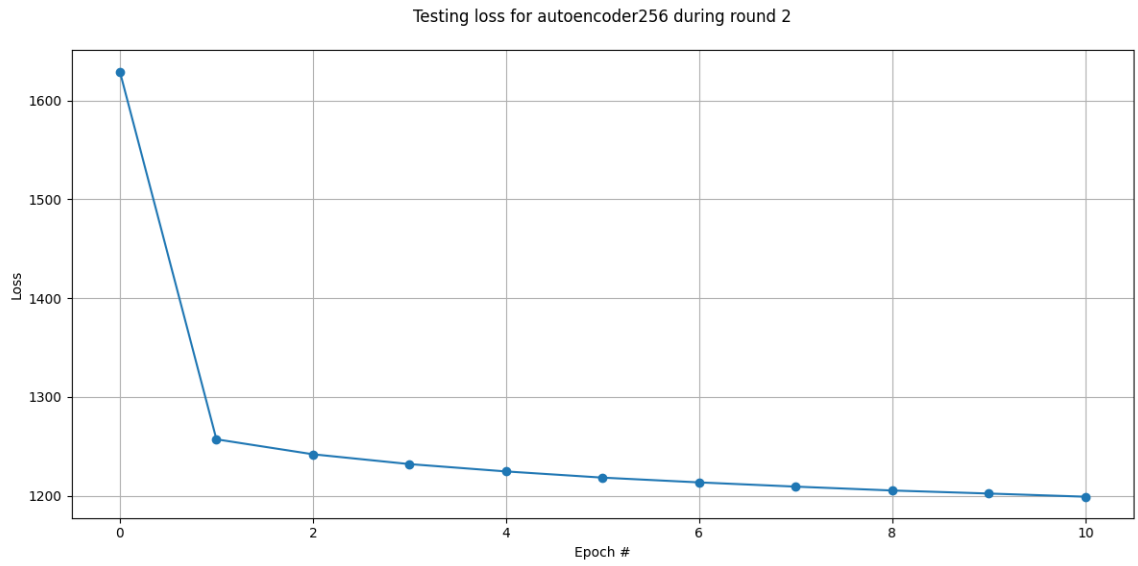
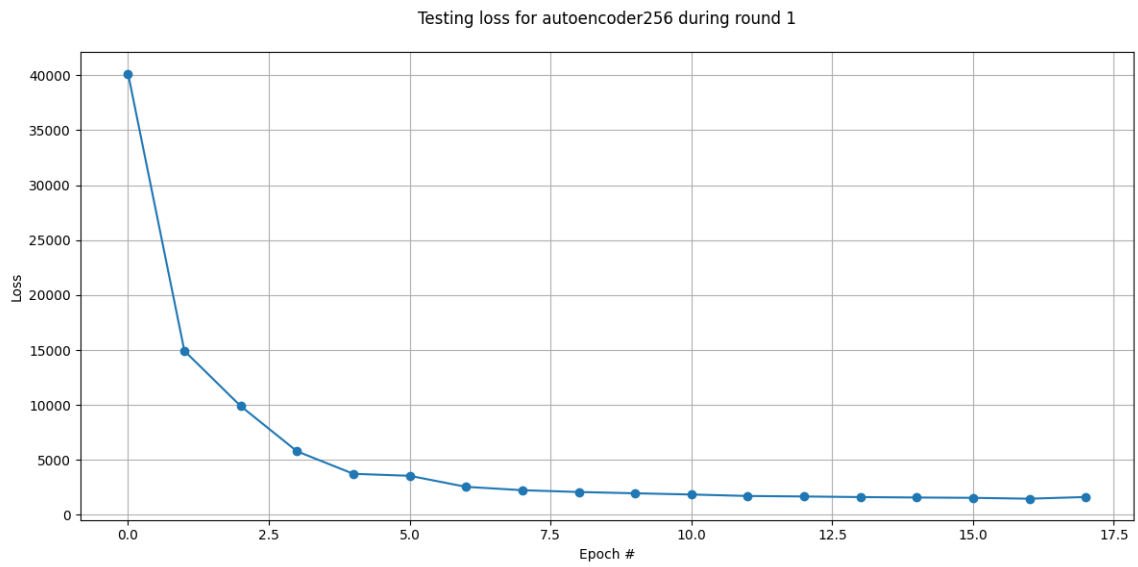


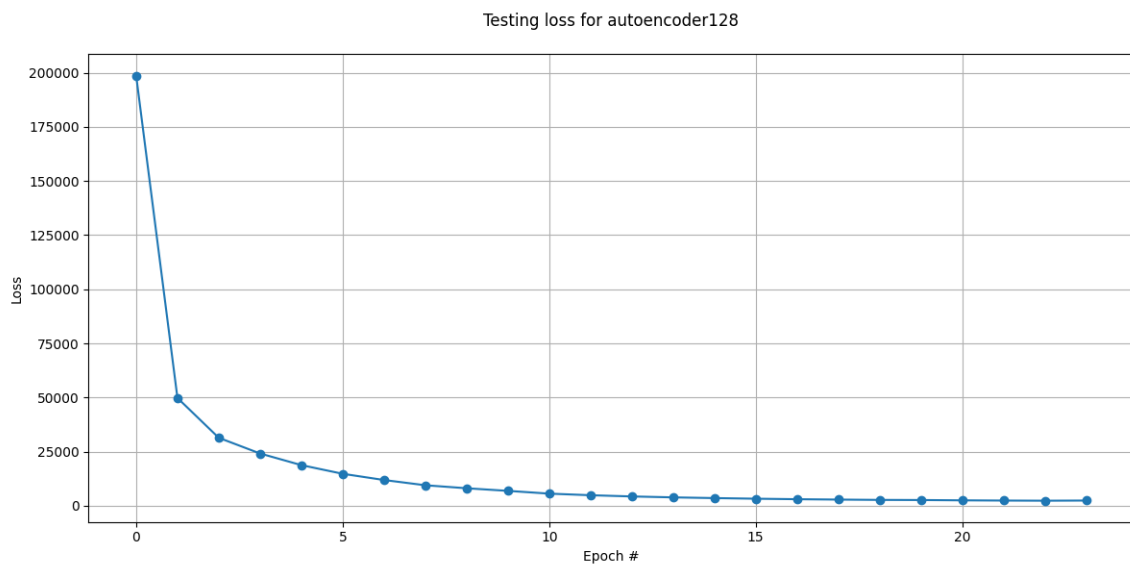
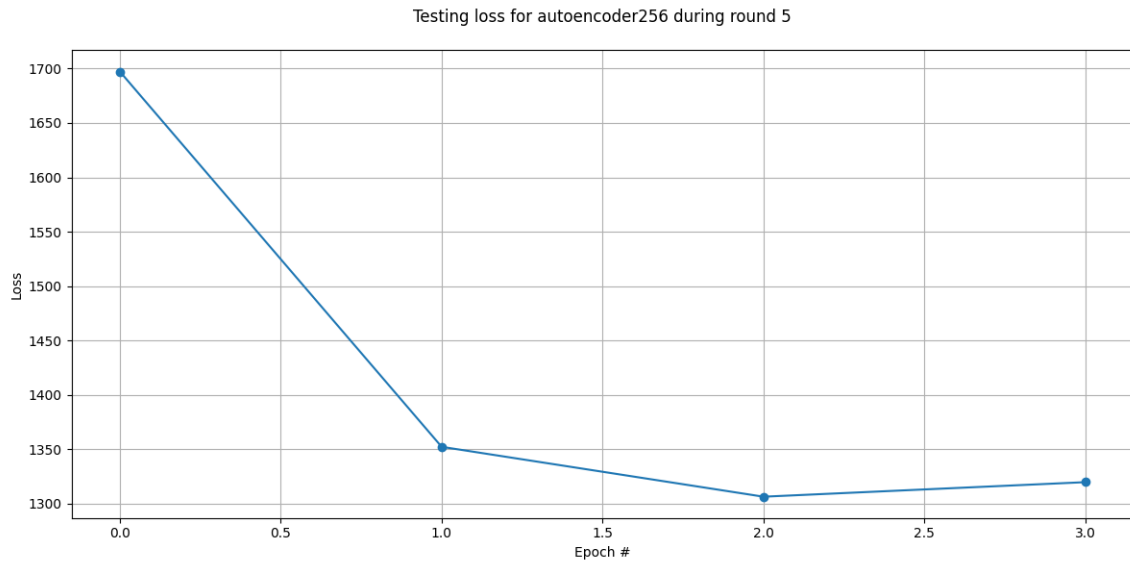
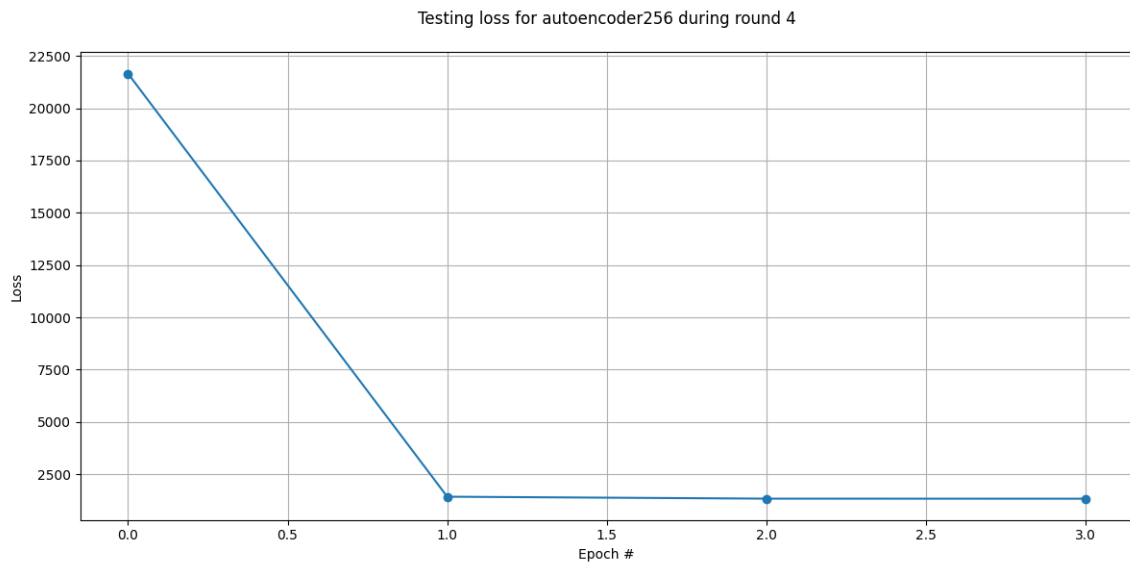
With what seemed to be optimal parameters set, we ran the training for **autoencoder128** uninterrupted for 28 epochs, at which point testing loss no longer improved.



3.5 Testing and Evaluation

Testing was performed between epochs with the loss values in the following plots. As stated, each test set comprised 1 million positions, from games, but not specific boards, distinct from the testing set.





In order to produce a more understandable metric of `autoencoder256` and `autoencoder128`'s performance, we calculate

the precision and recall for each type of piece and for castling rights. Additionally, we calculate the accuracy of retrieving the en-passant square, the current turn, and the half-move and full-move clocks. We use approximately 522k positions for these calculations.

A tolerance for the clock being off a single turn was allowed in these calculations. Additionally, it should be clarified that outputs from the model must be rounded before being able to be evaluated in this manner against targets.

Table 2: Metrics for `autoencoder256`

Piece	Color	Precision (%)	Recall (%)
Pawn	White	99.625432	98.853648
Knight	White	98.431897	97.990316
Bishop	White	98.650312	97.814000
Rook	White	97.980213	95.248318
Queen	White	98.394507	99.873316
King	White	99.235207	99.345225
<hr/>			
Pawn	Black	99.646759	99.061155
Knight	Black	98.426497	97.908771
Bishop	Black	98.649073	97.730684
Rook	Black	98.042899	94.811946
Queen	Black	98.375863	99.848622
King	Black	99.165624	99.558747
<hr/>			
Empty Square	—	99.126077	99.573267
<hr/>			
Castling	Color	Precision (%)	Recall (%)
Kingside	White	99.983490	99.997360
Queenside	White	99.981308	99.989319
Kingside	Black	99.979693	100.000000
Queenside	Black	99.981225	99.996871

Metric	Accuracy (%)
En-Passant Accuracy	100.000000
Turn Accuracy	100.000000
Clock Accuracy	99.994594

Table 3: Metrics for `autoencoder128`

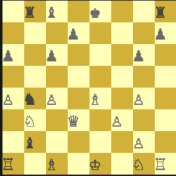


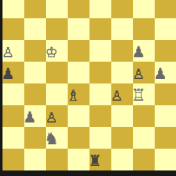
Piece	Color	Precision (%)	Recall (%)
Pawn	White	98.576081	97.783393
Knight	White	96.971959	96.884882
Bishop	White	96.867883	95.028615
Rook	White	96.590596	92.833626
Queen	White	95.531058	96.481651
King	White	96.365219	94.734168
<hr/>			
Pawn	Black	99.089336	97.885132
Knight	Black	97.131294	97.314483
Bishop	Black	97.036546	95.088112
Rook	Black	96.557212	93.169045
Queen	Black	95.746601	95.904487
King	Black	96.204841	95.601869
<hr/>			
Empty Square	—	98.650336	99.372572
<hr/>			
Castling	Color	Precision (%)	Recall (%)
Kingside	White	99.753606	99.759537
Queenside	White	99.625045	99.703538
Kingside	Black	99.752235	99.862134
Queenside	Black	99.755341	99.802178

Metric	Accuracy (%)
En-Passant Accuracy	99.999577
Turn Accuracy	50.302577
Clock Accuracy	95.720327

In almost all aspects, both models achieve extremely high performance, with `autoencoder128` only lagging a slight percentage or two in most measures. However very starkly, `autoencoder128` completely failed at the turn prediction task, and on inspection almost universally predicted boards were for Black’s turn. This is very odd, as the training data and loss functions were equivalent between the models. We have little to explain this discrepancy.

3.6 Visualizing Outputs

Finally, we present some visual examples of board inputs, encodings, and reconstruction by our models. Below are a couple examples of input FEN’s, a shortened look into the output of the encoder, and the retrieved board from processing the output of the decoder. Notice that board reconstruction is good, but far from perfect. This seems to be especially true of more sparse boards. Largely, precision seems to be much higher than recall, as errors are largely missing pieces.


```
autoencoder128 results
INPUT FEN: 1rb1k2r/3p3p/p1p3p1/8/PnP1B1P1/1N1q1P2/1b4P1/R1B1K1NR w KQk - 0 22

ENCODED TENSOR:
[-0.28800094127655032, 1.02781057357788, -0.963455080986023, -2.359076499938965, 2.557985305786133...]
DECODED FEN: 1r2k2r/3n3p/p1p3p1/8/PnP1B1P1/1N1qbP2/1b4P1/R3K2R b KQk - 0 21

autoencoder128 results
INPUT FEN: 1r1k4/8/p1K3p1/5pPp/P2B1PR1/1pP5/1Pn5/4r1N1 b - - 0 44

ENCODED TENSOR:
[-1.69686710834503170, 3.611454367637634, -2.486816167831421, -0.6049577593803406, 2.8648176193237305...]
DECODED FEN: 8/8/P1K3p1/p5Pp/3B1PR1/1pP5/2n5/4r3 b - - 0 42

```

4 Conclusion

As stated, we began this project with the end goal of constructing a full chess engine using an autoencoder and deep evaluation function. With this in mind, the autoencoder models we contribute serve as a launching off point for future work. We see large space for improvement in the autoencoders capabilities. Despite having numerical high performance metrics, these numbers mask the practical performance of our networks, as small error rates over many classes over many squares per board easily add up. In the future, we seek to revise our approach with alternative model architectures, loss functions, and the amount of training done. We firmly believe better performance can be reached through experimentation with these variables, which should be continued upon before attempting to implement any evaluation network.

References

- [1] Claude E. Shannon and Bell Telephone. “XXII. Programming a Computer for Playing Chess 1”. In: *Philosophical Magazine Series 1* 41 (1950), pp. 256–275. URL: <https://api.semanticscholar.org/CorpusID:12908589>.
- [2] Steven J. Edwards. *PGN Standard*. 1994. URL: <https://archive.org/details/pgn-standard-1994-03-12>.
- [3] *Lichess Open Database*. 2013. URL: <https://database.lichess.org/>.
- [4] Arman Maesumi. *Playing Chess with Limited Look Ahead*. 2020. arXiv: 2007.02130 [cs.AI]. URL: <https://arxiv.org/abs/2007.02130>.