Transformer Chess Bot for Move Prediction and Strategy Evaluation

Kyle Tranfaglia, Dustin O’Brien

05/19/25

Project Overview

Goal: Develop a probabilistic chess AI that plays competitive, human-like chess

Key Idea: Combine a transformer-based neural network (with attention + position averaging) with Monte Carlo Tree Search (MCTS) for move selection

Result: Functional AI with an estimated Elo of ~700. Not yet competitive—but captured human-like style and decision patterns

Why This Approach?

- Chess is similar to language: pattern-based, contextual
- Transformers excel at such tasks
- Probabilistic decision-making = more human-like
- Customizable "personality" via MCTS parameters

System Components

- **Neural Network**: Transformer-based model for evaluating positions
  - Uses position averaging, self-attention, residual connections, feed-forward networks, and layer normalization
- **Search Engine**: Monte Carlo Tree Search for move selection
  - Balance of exploration and exploitation, selective depth, neural network guidance, asymmetric tree development, anytime algorithm
- **User Interface**: Real-time move display, evaluation probabilities, configurable options

Exploring the Solution Space

Model Evolution:

- Baseline → transformer → deeper networks with residuals and position averaging

Data Pipeline:

- Lichess → GM games → hybrid datasets with minimum player Elos and Elo ranges (landslide victories)
- Filtered short draws, illegal moves, and imbalanced games

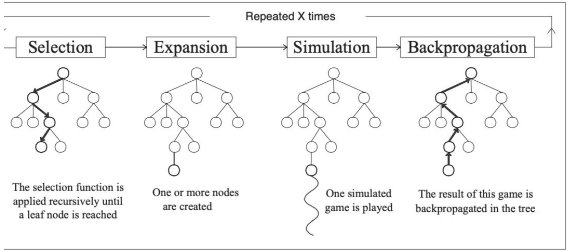
Search Strategy:

- Tuned MCTS with NN evaluations and exploration/exploitation balance

Evaluation Strategy

- **Move Prediction Accuracy**: Measuring the model's ability to predict expert moves using the test dataset
- **Human-Likeness**: Qualitative analysis of the model's playing style and strategic tendencies in all game phases
- **Live Testing**: Direct gameplay against human players, itself, and benchmarked opponents: Elo estimation = ~700

Monte Carlo Search Tree



- Uses random sampling and statistics to guide tree search
- Runs simulations by exploring possible future moves (depth-first)
- Balances exploring new moves vs. re-trying known good ones (exploration vs. exploitation)
- Great for games with huge move possibilities (like Chess or Go)
- Popularized by Alpha Go & Alpha Zero

Neural Network Pipeline

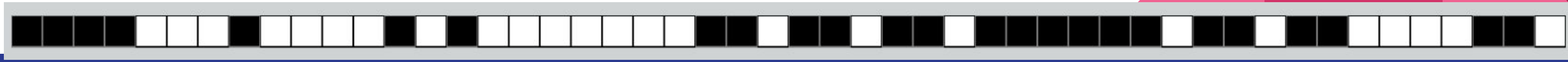
- Calculate vector encoding of Game board
- Run through model and back-propagate to improve NN
- Have model output probability of White Winning, Draw, Black Winning

Board Encoding

- Encoded 837 floats for input vector. 3 for output vector (Black, Draw, White)
- First index Player move (Black: 1, White: 0)
- Positions 2-4 (King & Queen Side Castling Rights)
- Positions 5-69 (Avaliabile En-passants) (64 Squares on Board)
- Positions 70-837 (Piece Locations) (64 Squares * 6 Piece Types * 2 Colors)
- Encoded Lichess & Grandmaster Game Positions for training

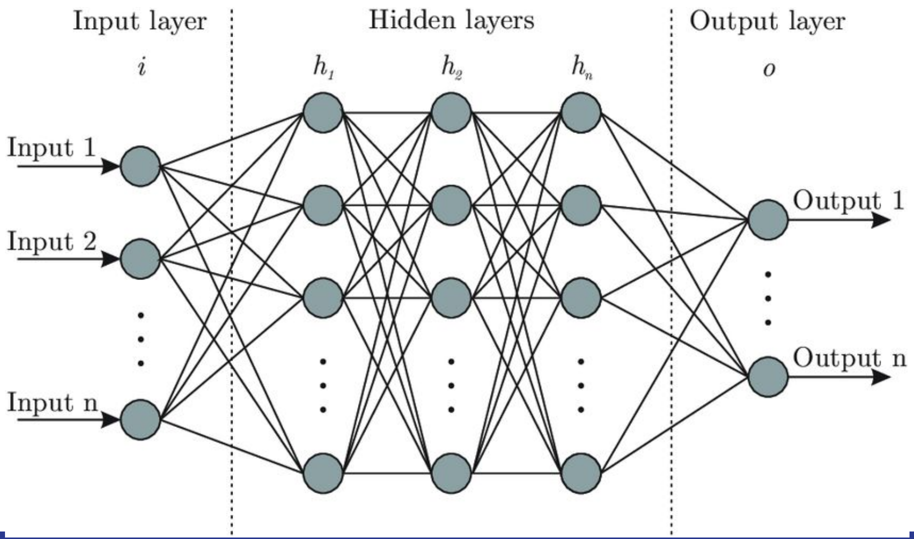
- Research validated encoding method
- Larger vector than normal due to En Passant encoding possible overfitting

Hypothetical Vector:



Initial Model

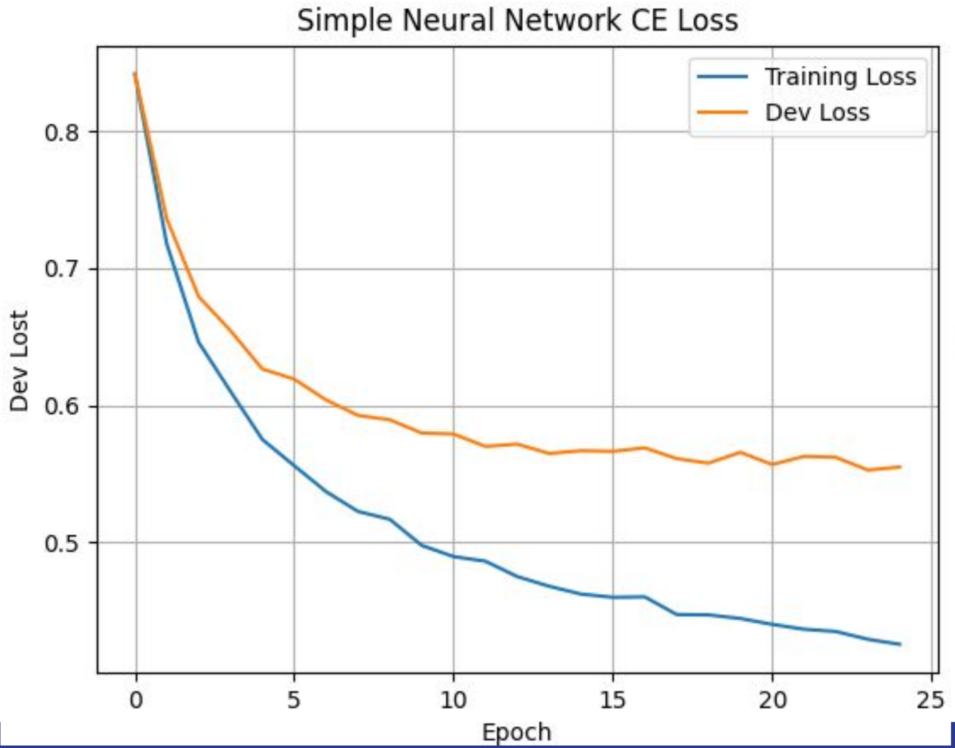
- No Attention Layer
- Simple large Neural Network (MLP)
- Has Res. Con., GeLU, AdamW, Layer Norm., Xavier Init., Batching, Step Sched.
- Cannot training seems to not work on large models (Deep or Wide)



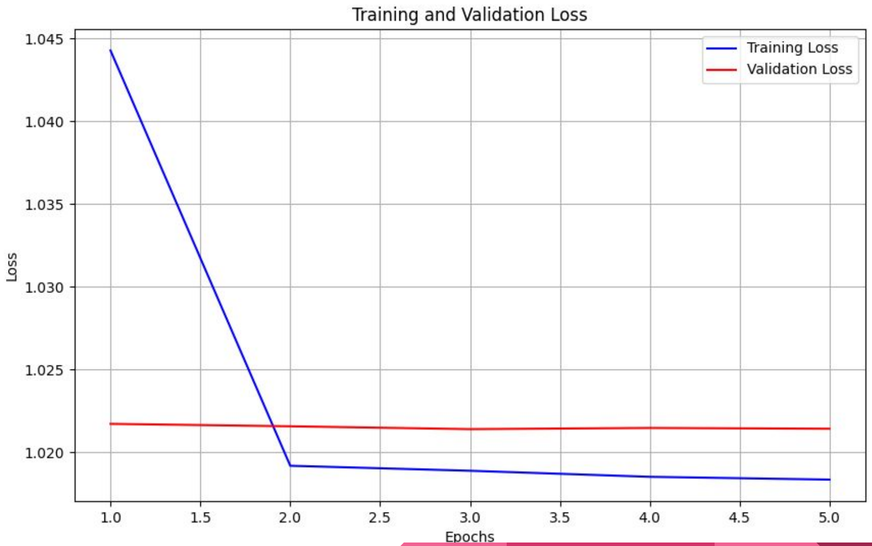

```python
class ChessArch(nn.Module):
    """
        Underlying Chess Eval Model Architecture
    """
    def __init__(self, model_width: int, model_depth:int, dropout_rate: float = .3):
        super(ChessArch, self).__init__()
        self.model_width = model_width
        self.model_depth = model_depth
        self.data_handler = DataHandler()
        self.init_layer = InitBlock(model_width, dropout_rate)
        self.hidden_layers = nn.ModuleList()
        self.final_layer = FinalBlock(model_width)


        for _ in range(model_depth):
            self.hidden_layers.append(HiddenBlock(model_width, dropout_rate))

    def forward(self, inputs):
        if not isinstance(inputs, torch.Tensor):
            raise ValueError(f"Invalid Type Final Layer {type(inputs)} expected {type(torch.Tensor)}")

        embedding = self.init_layer(inputs)

        for layer in self.hidden_layers:
            embedding = layer(embedding)

        embedding = self.final_layer(embedding)

        return embedding
```

Initial Model Performance

Small Model:



Large Model:



Problems With Model

- Roughly Small Model has 40% accuracy only mildly better than Random Guessing

- Doesn’t train large models neither wide nor deep (Unlikely Vanishing Gradient)

- Best Model: Width - 10 :: Depth - 5

Potential Solution

- Large amount of noise in dataset Ex. Opening Moves may have different winners and contradict
- Solution: Averaging winner for each position

- Potentially insufficient architecture and too complex for Data
- Solution: Introduce Attention

- Potential Problems with Vanishing Gradient
- Solution: Increase Residual Connections

Attention Model

- Transition from classification based Cross Entropy Loss to Distribution based KL Divergence Loss

- Added Attention Layers between each layer

- Increased Residual Connections only Moderate Improvements

- Outputs could only be Normalized

Major Fix (Pipeline Ordering)

```python
class FeedForwardBlock(nn.Module):
    def __init__(self, model_width: int, expansion_factor: int = 4, dr
        super(FeedForwardBlock, self).__init__()
        self.model_width = model_width
        self.hidden_dim = model_width * expansion_factor

        self.linear1 = nn.Linear(model_width, self.hidden_dim)
        self.linear2 = nn.Linear(self.hidden_dim, model_width)

        self.activ = nn.GELU()
        self.dropout = nn.Dropout(dropout_rate)
        self.layer_norm = nn.LayerNorm(model_width)

        # Initialize weights
        nn.init.kaiming_normal_(self.linear1.weight)
        nn.init.kaiming_normal_(self.linear2.weight)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        residual = x

        # Feed-forward network
        x = self.linear1(x)
        x = self.activ(x)
        #x = self.dropout(x)
        x = self.linear2(x)
        #x = self.dropout(x)
        # Residual connection and layer normalization
        x = self.layer_norm(x + residual)

        return x
```

```python
class FeedForwardBlock(nn.Module):
    def __init__(self, model_width: int, expansion_factor: int =
        super(FeedForwardBlock, self).__init__()
        self.model_width = model_width
        self.hidden_dim = model_width * expansion_factor

        self.linear1 = nn.Linear(model_width, self.hidden_dim)
        self.linear2 = nn.Linear(self.hidden_dim, model_width)

        self.activ = nn.GELU()
        self.dropout = nn.Dropout(dropout_rate)
        self.layer_norm = nn.LayerNorm(model_width)

        # Initialize weights
        nn.init.kaiming_normal_(self.linear1.weight)
        nn.init.kaiming_normal_(self.linear2.weight)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        residual = x
        x = self.layer_norm(x)
        # Feed-forward network
        x = self.linear1(x)
        x = self.activ(x)
        x = self.dropout(x)
        x = self.linear2(x)
        x = self.dropout(x)
        # Residual connection and layer normalization
        x = x + residual

        return x
```

New Results

- Significant Performance Increase
  - Accuracy ~40% -> ~90%
  - Loss ~.6 -> ~.3 (Different Metrics However)

Problems with Model

- Moderate Overfitting

Solution: Increased Dropout saw Slight Improvements

- Insufficient Data & RAM to hold Dataset

Solutions:

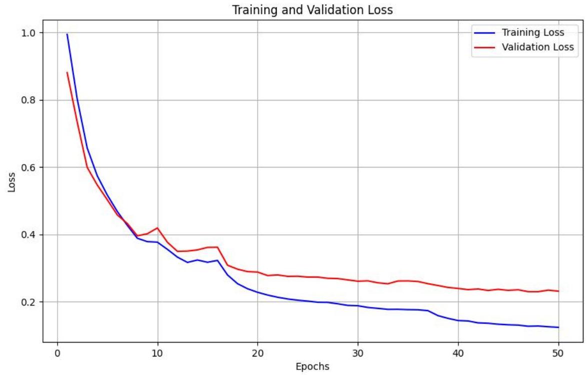
- Increased Dataset Quality
- Split dataset into Shards and train across multiple computers via MPI or Slurm

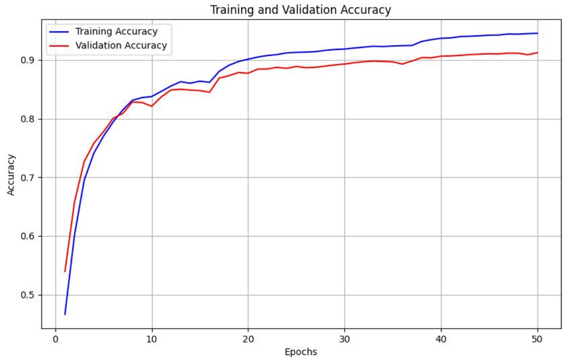
Successfully Created Shards and Dataset. However, didn’t get too distributing computation and data

- Improper MCST Integration

Solution: Change to Alphazero Loss function as well as change from accuracy based output to MCST heuristic output for more Alpha Zero like solution

Increased Dropout Results





What Worked

1. Transformer architecture successfully implemented
2. Effective integration with MCTS
3. Functional GUI with evaluation visualizations
4. Some pattern recognition & logical play emerged
5. Human-like behavior at lower levels

What Didn’t Work

1. Final Elo of 700, far below 2200 target
2. Model lacked tactical awareness (missed threats and free captures)
3. Limited generalization despite model scaling
4. Hardware bottlenecks → restricted training scale due to limited RAM
5. Still weak in opening principles and mid-game tactics, and becomes repetitive in end-game

Insights

- Data quality > Data size
  - GM games lack errors, low-Elo adds noise
  - Difficult to recognize good vs bad moves
- Position contradictions overshadow good moves
  - solved partially with averaging
- Hardware constraints limited training duration & model size
  - Limited to about 10000 Chess games → Ideally want 100000+
- Hybrid architecture validated core idea of probabilistic + strategic AI
  - Project served as proof of concept
  - More data plus better data = strong guiding model outputs

Future Work

1. **Reinforcement Learning** via self-play
2. **Hybrid Evaluations**: NN + traditional heuristics
3. **Skill-Focused Datasets** on openings, tactics, endgames, etc.
4. **Distributed Computing**: To allow for increased RAM usage and dataset storage
5. **Better Hardware** for deeper models & more data
6. **Fine-Tuning** pre-trained models instead of full training

Live Demo

a b c d e f g h
8 7 6 5 4 3 2 1
Draw by Threefold Repetition
Move History:
22. dxe5 d4
23. e6 Rc1
24. e5 Qe7
25. fxg5 f4
26. g6 Rb1
27. g7 Bc8
28. Bg2 hxg4
29. Ngf3 Rh5
30. Rc1 g3
31. Rd1 Rc1
32. Qb1 Rc2
33. Qc1 Rc3
34. Qc2 Rb3
35. Qc3 Rb4
36. Rda1 Rb3
37. Rb1 Rb4
38. Rba1 Rb3
39. Rb1 Rb4
Position Evaluation
White: 55.0%
Draw: 0.0%
Black: 45.0%
New Game
Undo Move
Flip Board
White: AI
Black: AI

Conclusion

- Built a working transformer-based chess AI
- Human-like play achieved, competitive level not met
- Foundations in place for stronger models with future work
- Valuable lessons learned in AI model training and evaluation
- Proof of concept with human-like model
    - More/better data with the suggested improvements shows promising signs of success