Mate in 2 Blog Post

Written by Amita Satish, Erkam Isbilir, Jazhiel Segura-Monroy, Luca Macesanu, and Tarun Kholay
Presentation Video: https://youtu.be/vE_ab8h__oo

Problem Background

A mate-in-2 chess puzzle is a type of chess problem where White (usually) is tasked to force checkmate in two moves – that is, White makes a move, Black replies with any legal defense, and then White delivers checkmate on the second move. These puzzles are a staple of chess compositions, often designed with unique “key” moves and clever themes that ensure checkmate no matter how the opponent defends. Solving a mate-in-2 requires looking ahead one move to envision a position that is checkmate-in-1 for the next turn. In essence, White’s first move creates a threat of mate and limits Black’s responses such that any reply still allows White to mate on the following move.

Mate-in-2 puzzles are significant for testing neural networks because they encapsulate a mini search problem and pattern recognition challenge. Unlike typical tactical puzzles that might be one-move checkmates or short tactics, a mate-in-2 demands multi-step reasoning: the solver must account for an opponent move and still guarantee a forced mate. Traditional chess engines solve these by brute-force search up to depth 3 (which is trivial for modern engines). However, for a neural network without explicit search, mate-in-2 puzzles test whether the network can learn patterns of coordination and forced mate setups. They serve as a controlled environment to assess if a network is merely pattern-matching or implicitly learning a form of lookahead. Indeed, chess composition puzzles (like mate-in-n) are often out-of-distribution for standard game positions and require “reasoning” rather than just pattern recall. A neural network that performs well on mate-in-2s might be demonstrating the ability to internalize tactical motifs and plan short sequences, which is a non-trivial capability in AI. This is one reason the very first chess program ever written (1951) was only capable of solving mate-in-two problems – full game play was too complex for the early computer, but mate-in-2 provided a proof-of-concept for chess reasoning

Project Goals

The goal of this project is to develop a neural-network-based solver for mate-in-2 puzzles. Given a chess position (provided in FEN notation) where a mate in 2 exists, the system should predict the solution moves (White’s first move, and the subsequent mating move). We approach this by training two types of models on a dataset of composed mate-in-2 problems: (1) a simple feed-forward neural network (baseline), (2) a more advanced ConvLSTM-based model that can capture spatial and temporal features, and (3) a transformer based model which can capture spatial features through positional embeddings. The project aims to evaluate whether these

networks can learn to solve the puzzles without brute-force search, and compare their performance.

We assume each puzzle has a *unique solution* or a canonical mainline (one key move and a forced mate), which allows us to use supervised learning (each puzzle in the dataset comes with the correct two-move sequence as the label). The network operates under the constraint of limited lookahead – it must output the mate sequence essentially in one go (unlike an engine that could search). Additionally, the input representation and network size are constrained to keep the problem tractable: we use a fixed 8×8×12 tensor input (described below) and reasonably small model architectures that can be trained on typical hardware. This project does not incorporate an actual move generator or tree search; instead, it's a pure neural prediction approach. As such, one challenge (and implicit constraint) is that the network must generalize from patterns seen in training puzzles and may struggle if a puzzle has a very novel theme or requires precise move-order reasoning outside its experience.

History of Computer Chess

The development of chess-playing AI spans from early symbolic systems to today's deep learning models. In 1950, Claude Shannon outlined how a computer might play chess using minimax search, while Alan Turing manually tested his own chess algorithm. In 1951, Dietrich Prinz's program could solve mate-in-2 problems, proving even limited hardware could tackle tactical puzzles.

Between the 1960s and 1990s, engines improved with better algorithms like alpha-beta pruning. In 1997, IBM's Deep Blue defeated world champion Garry Kasparov using brute-force search and expert-designed evaluation, a milestone in classical AI.

The 2000s saw engines dominate humans, still using human-tuned heuristics. A turning point came in 2017 with DeepMind's AlphaZero, which learned chess entirely via self-play and deep neural networks, defeating Stockfish after just 24 hours of training. This marked a shift from handcrafted algorithms to learning-based AI.

Today, neural networks are integral to top engines. Leela Chess Zero, an open-source AlphaZero-style project, uses distributed training, while traditional engines like Stockfish now incorporate neural evaluation (e.g., NNUE). The evolution from Deep Blue to AlphaZero reflects AI's shift from brute-force to learning-driven methods.

Implementation

Data Loading & Encoding

Chess positions are given in *Forsyth-Edwards Notation (FEN)*, a standard string format that describes piece placement, active player, castling rights, etc. For our purposes, we focus on the piece placement portion of the FEN (since other details like castling or en-passant are less relevant in mate puzzles). We parse each FEN string and encode the board as a tensor of shape 12×8×8. This is a common encoding in chess deep learning literature, albeit simplified: we use 12 channels, one for each piece type (6 for White and 6 for Black). The mapping from piece to channel is:

```
piece_to_plane = {
    'P': 0, 'N': 1, 'B': 2, 'R': 3, 'Q': 4, 'K': 5,   # White pawn, knight, bishop, rook, queen, king
    'p': 6, 'n': 7, 'b': 8, 'r': 9, 'q': 10, 'k': 11  # Black pawn, knight, bishop, rook, queen, king
}
```

Each channel is an 8×8 matrix corresponding to the chessboard: a value of 1 on `[channel, row, col]` indicates the presence of that piece type on that square, and 0 means no such piece on that square.

To fill this tensor, the data loader uses the `python-chess` library to parse the FEN into a `chess.Board` object, then iterates over all 64 squares. For each square that contains a piece, we compute the matrix indices and set the appropriate channel to 1. We take care to flip the board vertically when mapping to the tensor: in our code, `row = 7 - (square // 8)` is used. This means we index the tensor with row 0 as the top rank (rank 8) and row 7 as the bottom rank (rank 1). Flipping the vertical orientation ensures that the tensor's first row corresponds to White's 8th rank (from White's perspective). This is a conventional approach so that the neural network sees the board in a consistent orientation (white pawns moving "up" in the tensor, etc.). Column indexing is straightforward (`col = square % 8` yields files A through H as col 0–7). After encoding all pieces, we obtain a 12×8×8 float tensor (with 1.0 for occupied squares in the respective piece channel, 0.0 elsewhere), which is then converted to a PyTorch tensor of dtype float

Naive Feedforward Network

As a baseline, we implemented a simple feedforward neural network (multi-layer perceptron) to predict the mate-in-2 solution from a single board position. This model treats the problem as a

multi-class classification across the 64 board squares for each of four outputs (from and to squares of two moves). The architecture is relatively straightforward:
Input layer: 12×8×8 board tensor (768 inputs). We first flatten this 3D tensor into a 768-dimensional vector
Hidden layer 1: Fully-connected (dense) layer with 1024 neurons, followed by a ReLU activation. This expansion allows the network to learn a large set of features from the board.
Hidden layer 2: Fully-connected layer with 512 neurons, with ReLU activation. This further refines the feature representation. We chose two hidden layers somewhat arbitrarily as a balance between model capacity and overfitting risk – the idea was to give the network enough neurons to potentially recognize important patterns (like specific mating nets or piece constellations) without making it too deep.
Output layer: A fully-connected layer with 4 * 64 = 256 outputs. We reshape this output into 4 vectors of length 64, corresponding to:

1. White's first move from-square (64 classes)
2. White's first move to-square (64 classes)
3. White's second move from-square (64 classes)
4. White's second move to-square (64 classes)

Essentially, the network has four "heads," each producing a probability distribution over the 64 board squares. We apply a softmax on each head during training (via the loss function) to interpret them as predicted probabilities. The highest probability square in head 1 is the network's predicted from-square for the first move, etc.

This design was chosen for simplicity: the network outputs everything in one forward pass. It does *not* explicitly model the turn-by-turn sequence; instead, it must learn the relationship between the first and second move implicitly. For example, if the first move was a queen move, the network's third output (second move from-square) should often correspond to a different piece (maybe a rook or bishop delivering mate). The network has to pick a consistent combination of four squares that represent a logical mate-in-2 sequence. We hoped that by training on many puzzles, the hidden layers would internalize patterns like "if the key move goes to square X, then the mating move often comes from square Y" etc.

However, this "flat" approach is quite challenging for the network. It ignores the temporal aspect of the problem – the fact that the second move is made after the first move and in response to Black's move. The network might struggle to learn the dependency between the two moves, since it must output both moves from the initial position alone. Despite this, it serves as a useful baseline to see how far a plain neural net can go in solving these puzzles without any recurrence or search.

```
class TwoMoveMateSolver(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()              # 12x8x8 -> 768
```

```
        self.fc1     = nn.Linear(768, 1024)
        self.fc2     = nn.Linear(1024, 512)
        self.output  = nn.Linear(512, 4 * 64)    # outputs 256 logits
    def forward(self, x):
        x = self.flatten(x)          # [B, 768]
        x = F.relu(self.fc1(x))      # [B, 1024]
        x = F.relu(self.fc2(x))      # [B, 512]
        x = self.output(x)           # [B, 256]
        return x.view(-1, 4, 64)     # [B, 4, 64] reshape into 4 heads
```

As shown, after the final linear layer, we reshape the output to a `[Batch, 4, 64]` tensor for convenience, so that we can index the four heads easily in the loss calculation. (During training we applied softmax + CrossEntropyLoss on each of the 4 output vectors, as described in the next section.)

This naive network provides a baseline to compare against more sophisticated models. It has about 7681024 + 1024512 + 512*256 ≈ 1.2 million weights, which is quite modest. The expectation was that it might learn to solve some common patterns but could struggle on more complex compositions due to the lack of an explicit mechanism to handle sequential move dependencies.

ConvLSTM Model for Sequential Move Prediction

The second model we implemented is based on a Convolutional Long Short-Term Memory (ConvLSTM) network. The idea behind using a ConvLSTM is to explicitly model the sequence of moves in the mate-in-2 and capture spatiotemporal features – how the board configuration evolves from the initial position to after the first move (and potentially after the second move). ConvLSTM was originally proposed by Shi et al. (2015) as an extension of LSTMs for handling sequences of images. In a ConvLSTM, the usual LSTM equations are modified so that the matrix multiplications are replaced by convolutions, allowing the cell to preserve the spatial structure of the data (in our case, an 8×8 board). This is well-suited for chess boards, as it can learn patterns like piece movements or threats in a localized manner, and maintain a memory across time steps (moves).

We implemented a ConvLSTM cell module that takes an input tensor called X at time t (which has the shape [Channels, 8, 8]) and also takes the previous hidden state H at time t−1 and the previous cell state C at time t−1 (each of shape [HiddenDim, 8, 8]). It then computes the next hidden and cell states using convolution operations.

The cell uses gates like a standard LSTM: it computes the input gate (i), forget gate (f), output gate (o), and a candidate activation (g). In the code, we first concatenate the input X_t and the previous hidden state H_{t−1} along the channel dimension. Then we apply a 2D convolution to this combined tensor, which produces an output with 4 times the number of channels as the hidden dimension (so, for example, if the hidden dimension is 64, we get 256 channels).

Next, we split that output into four equal parts: one for each of the gates — input [i], forget [f], output [o], and candidate [g]. Each of these parts has size [HiddenDim, 8, 8].

Then, we apply activation functions: we use the sigmoid function for the input gate, forget gate, and output gate; and we use the tanh (hyperbolic tangent) function for the candidate activation.

The updated cell state is computed by combining the previous cell state and the new candidate activation, weighted by the forget and input gates. Finally, the new hidden state is computed by applying the output gate to the tanh of the updated cell state.

The equations for ConvLSTM are:

$$i_t = \sigma(W_{xi} * X_t + W_{hi} * H_{t-1} + b_i)$$

$$f_t = \sigma(W_{xf} * X_t + W_{hf} * H_{t-1} + b_f)$$

$$o_t = \sigma(W_{xo} * X_t + W_{ho} * H_{t-1} + b_o)$$

$$g_t = \tanh(W_{xg} * X_t + W_{hg} * H_{t-1} + b_g)$$

$$C_t = f_t \odot C_{t-1} + i_t \odot g_t$$

$$H_t = o_t \odot \tanh(C_t)$$

We stacked two ConvLSTM layers (two ConvLSTMCells in sequence) to increase the capacity. The first layer takes the input boards, the second layer takes the first layer's hidden state as input. Each ConvLSTM layer in our model had `hidden_dim = 64` feature maps and used a 3×3 kernel for convolutions (with padding=1 to preserve board size). Thus, the ConvLSTM's hidden state at each time is a 64×8×8 tensor.

For each puzzle, we feed a sequence of length T = 2 (in general, we designed the model to handle variable T). At time t = 0, the input X_0 is the initial position (a 12×8×8 tensor), and at time t = 1, the input X_1 is the position after White's first move. (In principle, we could extend the sequence further to include Black's response or the final mate position, but we found that using just two steps was sufficient for the model to get the necessary information.) The ConvLSTM processes these in order. We initialize the hidden state H_0 and the cell state C_0 as zero tensors for each ConvLSTM layer. Then, for t = 0 and t = 1, we update the states. After

the sequence is processed, we obtain the outputs H_0 and H_1 from the final ConvLSTM layer (where H_1 is the hidden state after processing the second board).

Our model, ChessConvLSTMNet, uses the ConvLSTM layers as described and then applies a prediction head to the sequence of hidden states. The prediction head is a small feedforward module: it flattens the 64×8×8 hidden state into a 4096-dimensional vector, applies a linear layer that reduces it to 512 units followed by a ReLU activation, and then applies a final linear layer to produce the output. We set the output dimension to 64, because, as discussed, we only predict the destination square for each move. The head is applied at each time step's hidden state, so we get a 64-dimensional output for t = 0 (predicting the first move's destination) and a 64-dimensional output for t = 1 (predicting the second move's destination). The model returns a tensor of shape [Batch, T, 64], representing the logits for the 64 board squares at each time step.

Here's an example of the forward pass:

```
# Within ConvLSTM.forward(x) where x is [B, T, C, 8, 8]
batch_size, seq_len, _, H, W = x.size()
# initialize hidden and cell states for each layer
h = [torch.zeros(batch_size, hidden_dim, H, W) for _ in layers]
c = [torch.zeros(batch_size, hidden_dim, H, W) for _ in layers]
outputs = []
for t in range(seq_len):
    input_t = x[:, t]            # input at time t
    for i, layer in enumerate(self.layers):
        h[i], c[i] = layer(input_t, (h[i], c[i]))  # ConvLSTMCell update
        input_t = h[i]           # output of this layer becomes input to
next
    outputs.append(h[-1])        # collect output of final layer
outputs = torch.stack(outputs, dim=1)  # shape [B, T, hidden_dim, 8, 8]
return outputs
```

Transformer Model for Move Prediction

The third model that we chose to implement was that of a transformer based approach. This is because we had already developed a baseline approach. Additionally, we had expanded upon this approach by introducing additional complexity which was able to capture spatial and temporal features by the use of a ConvLSTM model. As such, we were interested in observing the performance of a model that utilizes a different approach, that of a Transformer model.

Recently there have been some breakthroughs in the computer chess world, some of which was spearheaded by Google's DeepMind 2024 paper titled Grandmaster-Level Chess Without Search. Using Stockfish 16 evaluations on 10 millions games played on lichess in 2023, they trained a 9 million, 136 million, and 270 million parameter based transformers, the best of which

was able to achieve an outstanding nearly 2900 glicko 2 rating when playing against humans in 3 minute blitz games, all without explicitly running any sort of search algorithms such as the alpha-beta min maxing, or monte carlo tree search which is found in many of the current top chess engines. This highlights the potential of transformer based models and their applications in computer chess.

At its core, a Transformer model is a neural network architecture built to process the sequential data in parallel through a method known as self-attention. This allows the model to be highly versatile in capturing relationships among the entirety of the input data, empowering it to be especially suited to data structured such as that of a chess board state. A usual Transformer model is composed of multiple encoder layers which consists of multi-headed attention, input/output embedding, and masked attention. Additionally, Transformers can also utilize positional embeddings to retain information about the spatial structure of the input data, which is critical when considering the positional relationship between pieces of our problem. Our specific approach applies a Transformer architecture defined in the following manner to predict the solution moves where a mate in 2 exists.

Input: This model utilizes the tensor of shape [Batch, 12, 8, 8] as described previously. This input tensor is reshaped into [Batch, 64, 12] which effectively treats each square as a 12 dimensional token. Afterwards, these tokens are projected into a 128-dimensional embedding.

CLS Token: Next, a learnable CLS token of shape [1, 1,128] is attached to the beginning of the embedding, resulting in an embedding of shape [Batch, 65, 128]. This token represents the entire input sequence and is used to capture global information about the board.

Positional Embedding: Next, we incorporate a learnable positional embedding of shape [1, 65, 128] into the input sequence. This is done to implement spatial information which is critical in assisting the model in better understanding the spatial relationship between pieces on the board.

Transformer Encoder: This step processes the input sequence and creates contextualized representations of the tokens. This is where self-attention occurs, and overall, transforms the input tokens into contextualized representations. By capturing the context, highly quality predictions are able to occur.

Classification Head: Lastly, we extract the information given by the CLS token and pass it through a linear layer to produce a shape of size [Batch, 128]. Afterwards, we reshape this to [Batch, 4, 64] for the same reason as discussed before (for convenience, so that we can index the four heads easily in the loss calculation).

Here's an view of the forward pass in code:

```
    B = x.size(0)
      x = x.view(B, 12, 64).permute(0, 2, 1) # [B, 64, 12]
```

```
        x = self.input_proj(x)                  # [B, 64, 128]
        cls = self.cls_token.expand(B, -1, -1) # [B, 1, 128]
        x = torch.cat([cls, x], dim=1)          # [B, 65, 128]
        x = x + self.pos_embed                  # [B, 65, 128]
        x = self.encoder(x)                     # [B, 65, 128]
        cls_out = x[:, 0]                       # [B, 128]
        logits = self.head(cls_out)             # [B, 256]
        return logits.view(B, 4, 64)            # [B, 4, 64]
```

Training process & Evaluation

As you can see from the graphs at the bottom of this document, the naive model's approach quickly began to overfit to the training data, with the testing accuracy plateauing after 3 epochs. The training accuracy however continued to steadily increase to around 85% by epoch 10. The testing accuracy hovered at roughly 48%.

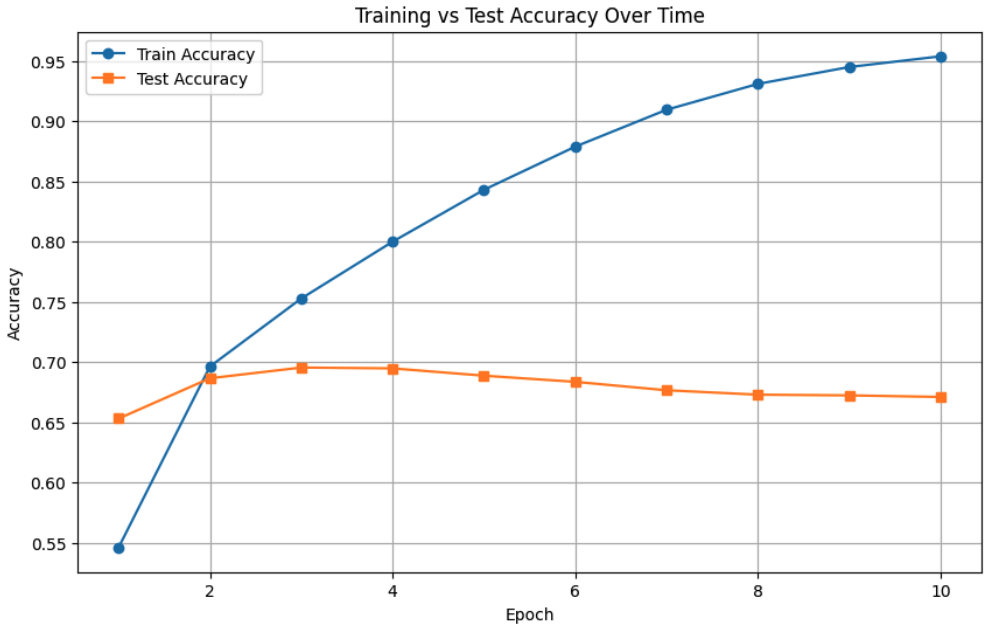
The ConvLSTM model's approach also overfit to the training data, with the testing accuracy once again plateauing after just 3 epochs. The training accuracy sharply increased around 95% by epoch 10. The testing accuracy hovered at roughly 67%, which was the best result out of the three models we tried. This is attributed to the spatial and temporal considerations within the ConvLSTM's architecture.

The Transformer model on the other hand did not overfit the training data. Rather both the training and testing accuracies continued to steadily increase before tapering off around 61% for the testing and roughly 64% for the testing accuracy by epoch 50. This model's testing performance places it between the naive and ConvLSTM approach. This is largely attributed to the transformer model's spatial considerations and lack of temporal considerations.

ConvLSTM's stronger performance shows that incorporating the sequential state (i.e., seeing the board after the first move) greatly helps in predicting the mate.This aligns with the notion that a network benefits from understanding *how the position can evolve*. In fact, our ConvLSTM essentially performs a learned "minimax": it considers the position after the key move (assuming a likely black response, since we fed the actual resulting position), and then finds the mate. This is somewhat analogous to a human solving a mate-in-2 by considering candidate key moves and envisioning the follow-up.

Naive NN Graph:

ConvLSTM Graph:



Transformer Graph: