

CS462: Final Project - Minesweeper

Yash Chennawar | yc1376@scarletmail.rutgers.edu

December 16, 2025

Code: <https://github.com/yashc73080/CS462-Deep-Learning/tree/main/Minesweeper>

This project analyzes the performance of various neural network approaches to solve a custom Minesweeper task.

1 Board Creation

The game environment was set up as a 22×22 board with many randomly placed mines, depending on difficulty chosen. Easy mode had 50 mines, medium had 80, and hard had 100. A mine was denoted by the number -1 and any hidden/unrevealed cell was denoted by the number -2. Mines were placed after the first move, which is randomly selected, and this avoids losing on the first turn.

The clue values were efficiently computed with a PyTorch convolution operation with padding of 1. It convolved the following filter over the game board.

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

This essentially summed all the adjacent mines for every cell, creating a board with clues from 1-8 depending on how many mines are neighbors.

For every next cell clicked (by either the logic bot or neural network), the game follows specific rules based on the official game. If the cell is a mine (-1), the game ends. If the cell is a clue from 1-8, just that cell is revealed. If the cell contains a 0, a flood fill algorithm is triggered. This utilizes a breadth-first search (BFS) on all connected 0 cells that can be reached from the clicked cell, ending at the last clue cells greater than 0 at the frontier.

The following Figure 1 shows a sample game in progress and the game board with all clues and mines revealed. Note that this full detailed board was not shown to the logic bot or any neural network as it contained information about mines. The agents only saw a masked board, resembling Figure 1a, with many hidden cells as they have not been explored.

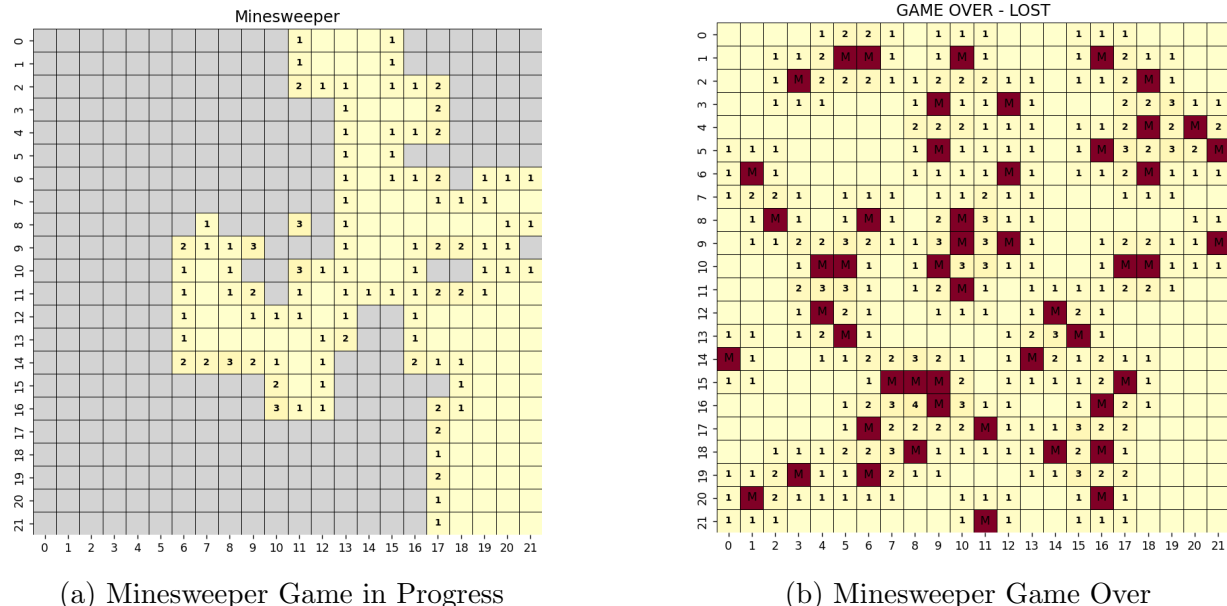


Figure 1: Sample Minesweeper Gameplay

2 Logic Bot

A simple logic bot was created using the algorithm in the assignment. It was then tested for 100 trials for all three difficulties, and the results are summarized in Table 1. 'Avg Safe' refers to the average number of safe cells opened per game. 'Avg Safe Wins' refers to the average number of safe cells opened in games that the bot won. And 'Avg Safe Losses' is the average number of safe cells opened in games that the bot lost.

Table 1: Minesweeper Performance by Difficulty

Difficulty	Trials	Wins	Win Rate	Avg Safe	Avg Safe Wins	Avg Safe Losses
Easy	100	67	0.670	84.98	104.04	46.27
Medium	100	25	0.250	105.67	199.56	74.37
Hard	100	2	0.020	51.14	221.50	47.66

3 Task 1: Playing by Mine Prediction

In this task, we have to train a network to output a prediction of which cells are safe to open, given an input of the game board in some state.

3.1 Data Generation

The data was generated by creating many game boards to pass to the network. Depending on difficulty, a random number of moves was played to advance the game. Initially, a

uniform random selection of non-mine cells was attempted. However, this resulted in poor performance as it failed to capture the local structural dependencies required for inference. So instead, I simulated a player by using a frontier boundary: where the clue cells meet the hidden cells. This approach checks all hidden and safe neighbors on the frontier and a random cell from this list is chosen. Each selection advances the game state by revealing more cells, using the logic explained earlier for the game board creation.

After the determined random number of moves, a mask board is generated from the real board. This is used for the input data, which is a tensor of dimension $12 \times 22 \times 22$. These 12 channels all encode a different meaning for the same game board, using one-hot encoding. Channel 0 will show a 1 where there are hidden cells on the board and 0 everywhere else. Channels 1 through 9 show 1 where the clue values 0 through 8 are shown respectively. Channels 10 and 11 were initialized as zero-matrices to maintain consistent dimensionality for subsequent tasks, though they remain unused in this specific architecture, these are just matrices of 0s.

The label is a matrix of dimension 22×22 , and the goal was to focus the model on the frontier. We want the model to make inferences only on cells adjacent to clue numbers, which is the frontier. As explained earlier, I initially randomly selected cells to open which meant that the network had to predict mine or not mine for every single hidden cell, even the ones it has absolutely no information about. This approach avoids forcing the network to guess, which removes random noise and adds real logic. Therefore, safe cells on the frontier have label 0 and mines have label 1. Every other cell has a label of -1 to teach the network to ignore those cells and only focus on the frontier. We do not want to penalize the model for failing to guess some random selection as this simply adds noise.

For this task, 15,000 train samples and 1,000 validation samples were used in the dataset, with a batch size of 128.

3.2 Model and Training

I selected a residual convolutional neural network for this task. This was essentially a network made up of convolutional residual blocks. The following shows the architecture used.

ResidualBlock:

- **Conv Block 1:** Conv2d(kernel=3, dilation), BatchNorm2d, ReLU
- **Conv Block 2:** Conv2d(kernel=3, dilation), BatchNorm2d
- **Optional Downsample:** Conv2d(kernel=3, dilation), BatchNorm2d
- **Skip Connection:** ReLU with identity

MinePredictionNet:

- **Initial Conv Block:** Conv2d($12 \rightarrow 64$, kernel=3, padding=1), BatchNorm2d, ReLU
- **Layer 1:**
 1. ResidualBlock($64 \rightarrow 64$, dilation=1)

- 2. ResidualBlock($64 \rightarrow 64$, dilation=1)
- **Layer 2:**
 - 1. ResidualBlock($64 \rightarrow 128$, dilation=2)
 - 2. ResidualBlock($128 \rightarrow 128$, dilation=2)
- **Layer 3:**
 - 1. ResidualBlock($64 \rightarrow 128$, dilation=4)
 - 2. ResidualBlock($128 \rightarrow 128$, dilation=4)
- **Layer 4:**
 - 1. ResidualBlock($64 \rightarrow 128$, dilation=8)
 - 2. ResidualBlock($128 \rightarrow 128$, dilation=8)
- **Last Conv Block:** Conv2d($128 \rightarrow 1$, kernel=1)

Each residual block, meaning each convolutional layer, had a dilation factor. This was to expand the receptive field as in Minesweeper, information about the safety of a cell can come from clues that are far away. A simple 3×3 convolution only sees immediate neighbors. Increasing dilation to 2, 4, and 8 allowed the the network to see larger amounts of the field at once, its receptive field. We also avoided pooling to preserve exact spatial positions. Pooling would blur the receptive field, which would mean losing the exact position of a mine, a potentially fatal error. Dilated convolutions expand board size without reducing board size.

Training was set up with a Binary Cross Entropy with Logits Loss criterion, Adam optimizer, and Cosine Annealing learning rate scheduler. Weight decay was also used. To penalize false negatives (predicting a mine as safe), a weighted loss was applied to the loss map derived from the model output and labels. This higher weight was based on the proportion of mines to not-mines in the board, based on difficulty. Easy difficulty had a 10x weight while medium had a 5x weight.

This network still outputs probabilities of safest next choice for the whole board. To actually evaluate the trained network, a neural policy was implemented to choose the hidden cell with the highest predicted safety anywhere on the board. The model has been trained to prioritize the frontier but it still gives probabilities for the whole board. Using this policy, and a random choice fallback if no move is made, the network can also play Minesweeper, similar to the logic bot. It uses a sigmoid function to compute probabilities across the board and choose the highest hidden cell.

A learning rate of 0.001 and weight decay of 0.0001 was used for 25 epochs.

3.3 Results and Analysis

Two networks were trained, one for easy difficulty and one for medium difficulty, with the configuration explained above. Table 2 and Figure 2 show the training behavior of the model. The loss shows the weighted Binary Cross Entropy (BCE) loss. The low values of

0.227 and 0.184 demonstrate that the network converged to a strategy that minimized most fatal errors. The validation precision shows that the model correctly identified the state (Mine or Safe) for 91.7% of the unrevealed cells.

Metric	Value
Final Train Loss	0.227
Final Validation Loss	0.184
Validation Precision	0.917

Table 2: MinePredictionNet Training Results, Easy Difficulty

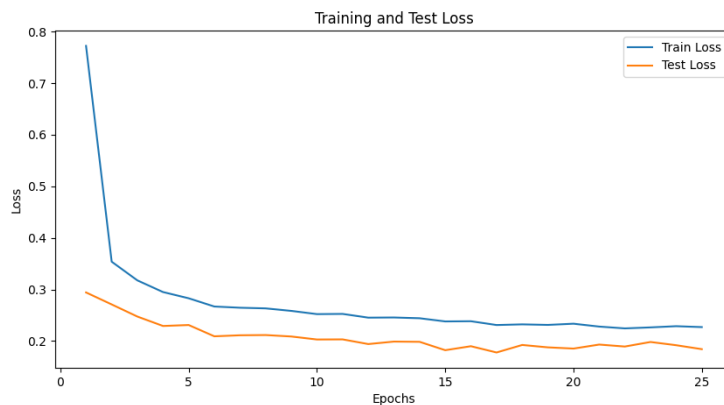


Figure 2: MinePredictionNet Training Dynamics, Easy Difficulty

This network trained on the easy dataset was compared to the Logic Bot, and the results are summarized in Table 3. 'Avg Safe Moves' shows the number of moves the bot survived on average, with a standard deviation shown in the last column. 'Avg Mines Triggered' shows the average number of mines a bot triggered in its gameplay.

Method	Win Rate	Avg Safe Moves	Avg Mines Triggered	Safe Moves St. Dev.
LogicBot	0.67	84.98	0.33	44.38
Neural Net	0.75	64.76	0.25	28.68

Table 3: Comparison for Easy Difficulty (100 Games)

Similarly, a different network with same architecture was trained on a medium difficulty dataset. Training dynamics are in Table 4 and Figure 3.

Metric	Value
Final Train Loss	0.305
Final Validation Loss	0.302
Validation Precision	0.877

Table 4: MinePredictionNet Training Results, Medium Difficulty

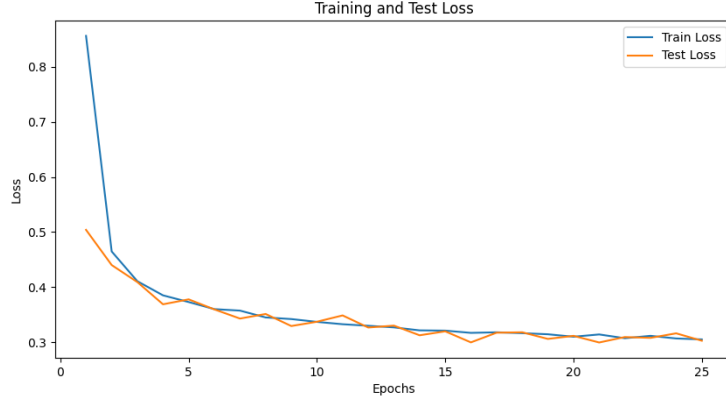


Figure 3: MinePredictionNet Training Dynamics, Medium Difficulty

Both the medium and hard datasets were tested with the network trained on medium difficulty. This is because the custom weight for the medium network would still perform well for the hard dataset. Tables 5 and 6 show the comparison between the logic bot and neural network for medium and hard difficulty games, respectively. Table 7 shows confidence intervals for key values. It also analyzes 'God Mode', which refers to when the bots are allowed to trigger mines and keep going with that information.

Method	Win Rate	Avg Safe Moves	Avg Mines Triggered	Safe Moves St. Dev.
LogicBot	0.25	105.67	0.75	92.27
Neural Net	0.52	118.02	0.48	70.33

Table 5: Comparison for Medium Difficulty (100 Games)

Method	Win Rate	Avg Safe Moves	Avg Mines Triggered	Safe Moves Std. Dev.
LogicBot	0.02	51.14	0.98	75.79
Neural Net	0.24	100.09	0.76	94.88

Table 6: Comparison for Hard Difficulty (100 Games)

Metric	LogicBot	NeuralNet	Difference (Neural - Logic)
Win Rate (%)	19.00 ± 7.73	47.00 ± 9.83	+28.00
Avg Steps Survived	97.92 ± 17.61	106.74 ± 14.72	+8.82
Avg Mines Triggered (God Mode)	5.35 ± 1.06	7.13 ± 3.01	+1.78

Table 7: Performance Comparison: LogicBot vs NeuralNet

The quantitative results demonstrate a clear ranking in performance, with the Neural Network consistently outperforming the Logic Bot as difficulty increases.

As shown in Tables 3, 5, and 6, the gap between the two agents widens with difficulty. On Easy mode, the Logic Bot is already competent (67% win rate) because the board density allows for simple logical moves with very little guessing. However, on Hard mode, the Logic Bot’s performance collapses to 2%, while the Neural Network maintains a 24% win rate. This suggests that the Logic Bot is limited by deterministic inference. When the Logic Bot encounters an ambiguous state, it guesses uniformly at random. But instead, the Neural Network has learned probabilistic patterns where one unrevealed cell is statistically safer than another, even if neither is guaranteed to be safe.

Table 7 highlights that the Neural Network survives longer on average (+8.82 steps) and achieves a significantly higher win rate (+28%). The God Mode analysis shows that the Neural Network triggers slightly more mines on average (+1.78) than the Logic Bot. This counter-intuitive result can be attributed to the Neural Network’s ability to progress further into the game. In games where the board is unsolvable because of forced guesses, the Logic Bot often fails early or leaves large sections of the board untouched. The Neural Network is confident in its learned probabilities so it attempts to solve these sections. While this leads to more board clearing (higher win rate), it also results in more mine triggers.

To understand the decision-making differences, I analyzed specific board states where the agents disagreed. The Logic Bot strictly adheres to definite knowledge; if `inferred_safe` is empty, it picks a random cell from `cells_remaining`. The Neural Network, however, outputs a continuous probability distribution. While testing, specific divergences were observed in "corner" cases and "wall" structures. For example, when facing a wall of unrevealed cells near to clues, the Logic Bot sees no distinction between them. The Neural Network instead preferred to reveal corner cells or cells breaking a pattern, which means that it learned that mines are less likely to group together in high density. This is because many mines together would violate the constraints on possible clue numbers.

In conclusion, the `MinePredictionNet` successfully mimics the Logic Bot on easy tasks but exceeds it on complex tasks by converting a logic puzzle into a probabilistic optimization problem. This allowed it to navigate uncertainty that causes the hard-coded Logic Bot to fail.

4 Task 2: Playing by Move Prediction

In this task, instead of predicting which cell is a mine or safe, we predict the overall quality of a given move. We have to predict how many steps a bot will survive given a current board state and a specific move. This allows us to make a policy that selects the move with highest predicted survival duration.

This problem uses an Actor-Critic approach. The Actor is the agent playing the game and making decisions while the Critic evaluates the Actor’s moves and learns from them.

1. **Generation 0:** The Critic (neural network) learns to predict the survival steps of the *Logic Bot*.
2. **Generation 1:** The Critic is used as the new Actor (the *Neural Bot*), and the Critic is re-trained on data generated by this new, potentially superior agent.

4.1 Data Generation

Since there is no pre-existing dataset for ”survival steps,” a custom simulation pipeline was implemented. The data generation process for a single sample works as follows:

1. **Initialization:** A game board is initialized, and an Actor Bot (Logic Bot for Gen 0, Neural Bot for Gen 1) plays the game for a random number of steps (5 to 15) to reach an intermediate state.
2. **Intervention:** At this state, a target move is selected to be evaluated. To ensure the model focuses on relevant moves, 80% of the time a ”frontier” cell (a hidden cell adjacent to a revealed number) is chosen. 20% of the time, a random hidden cell is chosen to ensure coverage of the entire board.
3. **Simulation:** The chosen move is made. If it is a mine, the survival count is 0. If it is safe, the Actor Bot continues to play the game until it wins or loses. The number of additional steps survived is recorded.
4. **Labeling:** The raw survival count is normalized to create a training label. A specific offset of 0.5 was added to distinguish between ”hitting a mine immediately” and ”surviving at least one step.”

$$\text{Label} = \begin{cases} 0.0 & \text{if move is a mine} \\ 0.5 + \frac{\text{steps survived}}{100.0} & \text{if move is safe} \end{cases}$$

This labeling scheme ensures that the model learns a distinction between fatal moves (value 0.0) and safe moves (value > 0.5), while also differentiating between ”safe but short-term” and ”safe and strategic” moves.

Therefore, the inputs were the current board state, a mask of the selected move, and the label explained above. The current board state was encoded in a way identical to that of Task 1. Due to the computational complexity of this data generation task, it was not feasible

to generate data on the fly, like in Task 1. So, 10,000 train samples and 5,000 test samples were generated and cached at medium difficulty. Data augmentation (rotations and flips) was applied to improve generalization as there is a risk of overfitting in this task.

4.2 Model and Training

The **CriticNet** is a Fully Convolutional Network that outputs a scalar value for every cell on the board, representing the predicted survival score for clicking that cell. The architecture has three convolutional blocks:

- **Block 1:** Conv2d ($12 \rightarrow 32$, 3×3), BatchNorm, ReLU.
- **Block 2:** Conv2d ($32 \rightarrow 64$, 3×3), BatchNorm, ReLU, Dropout (0.5).
- **Block 3:** Conv2d ($64 \rightarrow 32$, 3×3), BatchNorm, ReLU, Dropout (0.5).
- **Head:** Conv2d ($32 \rightarrow 1$, 1×1) to produce the final score map.

Dropout was included to prevent overfitting, as the model needs to generalize to unseen board patterns rather than memorizing specific game states.

The model was trained using Mean Squared Error (MSE) loss. Since we only know the true survival steps for the single specific move we simulated, the loss is masked. It is calculated only at the coordinate of the taken move:

$$\mathcal{L} = \text{MSE}(\text{PredictedMap} \odot \text{Mask}, \text{TrueLabel})$$

where Mask is 1 at the target move indices and 0 elsewhere.

Once trained, the **CriticNet** defines the policy for the **NeuralBot**. For any given board state, the bot:

1. Passes the board through the network to get a 22×22 grid of scores.
2. Masks out already revealed cells.
3. Selects the hidden cell with the maximum predicted value (argmax).

Both networks were trained with Adam optimizer and Cosine Annealing scheduler 10 epochs. The first used a learning rate of 0.0005 and the second used 0.001. Both had a weight decay of 0.0001 for regularization and to avoid overfitting.

4.3 Results and Analysis

The **CriticNet** was trained for two generations on the Medium difficulty setting.

Generation 0 (Logic Bot Actor): The model learned to mimic the survivability of the Logic Bot. Since the Logic Bot plays perfectly deterministically, the Critic learned to identify "safe" logic moves versus "unsafe" guesses.

Generation 1 (Self-Play): The **NeuralBot** used the Gen 0 Critic to play games. This generated a new dataset of 10,000 samples. The same Critic was then retrained (fine-tuned)

on this new data. This step is crucial because the Neural Bot (unlike the Logic Bot) can make probabilistic decisions, potentially surviving longer in unclear situations.

Figure 4 and Table 8 show the training loss for the Gen 0 model. Figure 5 and Table 9 show the training loss for the Gen 1 model. The validation loss decreases steadily for generation 1, indicating the model is somewhat successfully learning to predict the outcome of its own policy. It is important to note that the "Test Accuracy" reported in Tables 8 and 9 represents the Mean Squared Error (MSE) of the normalized step predictions (steps / 100).

Metric	Value
Train Loss	0.0842
Validation Loss	0.2639
Test Accuracy	0.2621

Table 8: Training Results for Generation 0

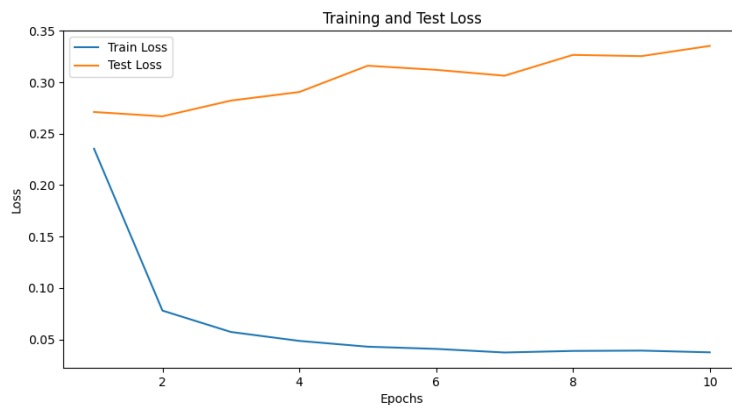


Figure 4: CriticNet Training Loss (Generation 0)

Metric	Value
Train Loss	0.0447
Validation Loss	0.0395
Test Accuracy	0.0396

Table 9: Training Results for Generation 1

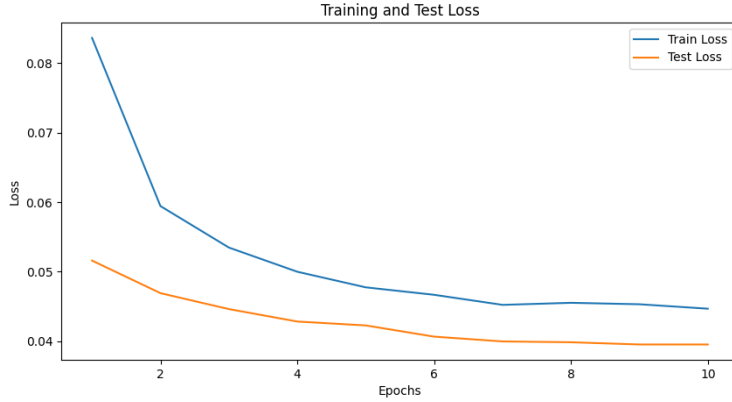


Figure 5: CriticNet Training Loss (Generation 1)

Tables 10 and 11 show the comparisons of the logic bot and critic networks.

Method	Win Rate	Avg Safe Moves	Avg Mines Triggered	Safe Moves Std. Dev.
LogicBot	0.25	105.67	0.75	92.27
CriticNet v0	0.00	4.20	1.00	4.22

Table 10: Comparison for Medium Difficulty (100 Games): LogicBot vs CriticNet v0

Method	Win Rate	Avg Safe Moves	Avg Mines Triggered	Safe Moves Std. Dev.
LogicBot	0.25	105.67	0.75	92.27
CriticNet v1	0.00	8.47	1.00	8.63

Table 11: Comparison for Medium Difficulty (100 Games): LogicBot vs CriticNet v1

In the initial generation, the Critic was trained on data generated by the LogicBot. While the training MSE of 0.084 suggests the model learned to predict the LogicBot’s survival reasonably well, the deployment performance was catastrophic (Win Rate: 0%, Avg Safe Moves: 4.20).

This collapse is an example of distribution shift. The LogicBot only selects safe or highly probable cells; therefore, the Critic was never exposed to ”bad” moves (clicking mines) during training. When the NeuralBot (Actor) queried the Critic for the value of all hidden cells, the Critic failed to penalize dangerous cells simply because it had never seen them clicked. The Actor exploited these ”blind spots,” selecting cells with incorrect high predicted values and triggering mines almost immediately.

Generation 1 utilized the ”bad” gameplay data from Generation 0 for retraining. As shown in Table 11, the average survival time doubled to 8.47 steps. This improvement validates the

Actor-Critic hypothesis: by adding the NeuralBot’s own failures to the dataset, the Critic learned to associate those specific bad moves with low survival scores. The drop in MSE ($0.26 \rightarrow 0.04$) also shows that the dataset shifted from high-variance LogicBot games to low-variance, short-duration NeuralBot games, which are easier to predict. While the bot is still far from solving the game, the positive slope in performance indicates that the model is successfully learning from its own mistakes.

5 Task 3: Thinking Deeper

This task explores ”thinking time” in neural networks. We are testing if a sequential model can improve its decision-making quality by ”thinking” about the board state for more iterations before making a final output. This allows the network to refine its predictions across the board over time.

5.1 Data Generation

For this task, the data generation strategy from Task 1 was reused. It was used to generate 15,000 static board snapshots for training and 1,000 for validation at medium difficulty. Each sample consists of a masked game board input (encoded with 12 channels) and a target label map indicating safe frontier cells and mine frontier cells.

It is important to note that while the model is sequential, the data itself is not a sequence of game frames (video). Instead, the ”sequence” is internal to the model: the model processes a single static input board repeatedly for a fixed number of timesteps ($T = 10$) to simulate a deepening thought process.

5.2 Model and Training

To implement the concept of thinking, I designed a Recurrent Convolutional Network, **ThinkingNet**. Unlike a standard Feed-Forward network that processes the input once, this model maintains a ”thought state” (hidden state) that is iteratively updated. It had the following architecture.

- **Input Encoding:** A convolutional layer projects the $12 \times 22 \times 22$ input into a 64-channel hidden state H_0 .
- **Thinking Block:** A reusable Residual Block (**ThinkingBlock**) with of two 3×3 convolutions with ReLU activations. The same block (shared weights) is applied iteratively: $H_{t+1} = \text{ThinkingBlock}(H_t)$.
- **Output Head:** At every time step t , a readout convolution maps the current state H_t to a probability map P_t .

The model was not trained by calculating loss only on the final output. Instead, the Binary

Cross Entropy loss was calculated for the prediction at every time step t from 1 to 10.

$$L_{total} = \sum_{t=1}^{10} \text{BCE}(P_t, \text{Target})$$

This forces the model to produce a reasonable guess immediately (at $t = 1$) but encourages it to refine and correct that guess as t increases. The training used the same class weighting as Task 1 to penalize mine triggers.

5.3 Results and Analysis

First, the model was trained on the dataset, with dynamics summarized in Table 12 and Figure 6. A reasonable loss was achieved.

Metric	Value
Train Loss	0.4103
Validation Loss	0.3539
Final Validation Precision	0.8793

Table 12: Thinking Network Training Results

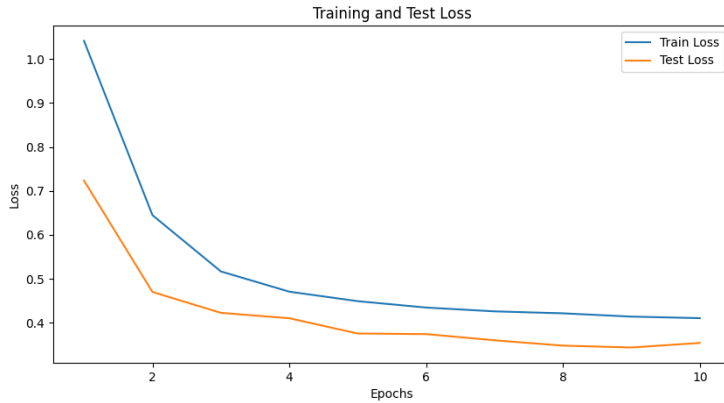


Figure 6: Thinking Model Training Dynamics

This thinking network was also compared to the logic bot on medium difficulty, as shown in Table 13. The thinking network beat the logic bot, indicating that seeing the game board for longer has a positive effect.

Method	Win Rate	Avg Safe Moves	Avg Mines Triggered	Safe Moves Std. Dev.
LogicBot	0.25	105.67	0.75	92.27
ThinkingNet	0.30	91.14	0.70	71.89

Table 13: Comparison for Medium Difficulty (100 Games): LogicBot vs ThinkingNet

The model was then evaluated to determine if thinking longer correlates with better performance. The validation loss was measured at each internal step of the model. As illustrated in Figure 7, the loss decreases and then increases as thinking steps increase, resembling a parabolic shape. The loss decreases until step 5 (the training depth as the model was only allowed to train for 5 steps) and then rises. This 'parabolic' shape indicates that while the model generalizes well within its training horizon, extending the recurrence beyond the trained depth ($T > 5$) introduces noise, suggesting the need for training on longer sequences to utilize the full potential of deeper thinking. If the model were allowed to train for 10 steps, we could expect a monotonic decrease.

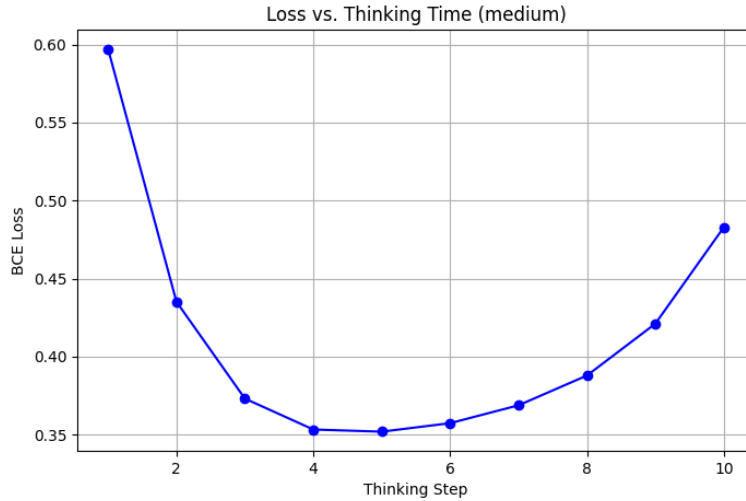


Figure 7: BCE Loss vs. Thinking Step (Medium Difficulty)

To verify if lower loss translates to better gameplay, the model was evaluated by playing games with varying thinking depths (steps = 1, 3, 5, 8, 10). The results are summarized below:

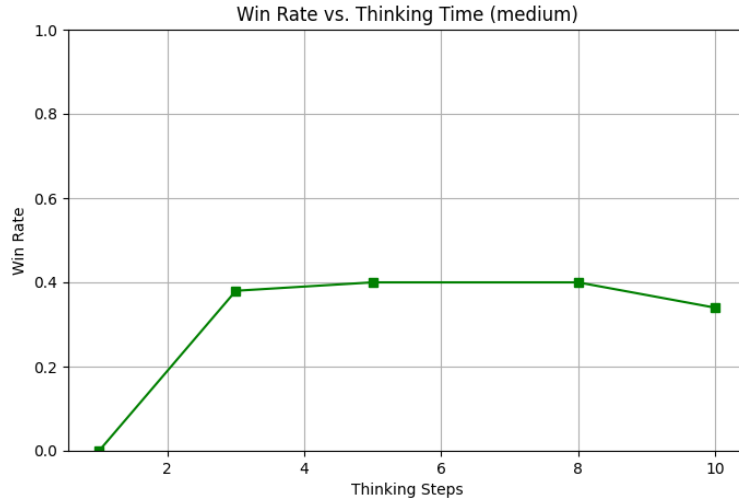


Figure 8: Win Rate vs. Thinking Steps

The upward trend in win rate demonstrates that the additional computation time is functionally useful. A model allowed to "think" for 8 steps outperforms the same model forced to react after only 1 step. This aligns with the intuition that Minesweeper needs information to go through the grid and inform predictions for far away sections.

Qualitative analysis of the probability heatmaps (Figure 9) reveals how the model's confidence evolves. At $T = 1$, the heatmap is often blurry or uncertain around complex boundaries. By $T = 10$, the model sharpens its predictions, driving probabilities of safe cells closer to 1.0 and mines to 0.0, effectively reducing noise in its decision-making.

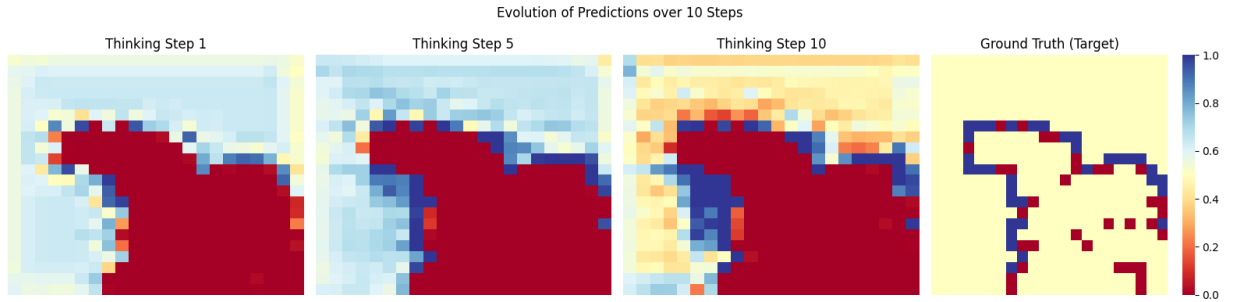


Figure 9: Evolution of Mine Predictions over 10 Steps