RL and CNN-Based Approaches for Maze Localization

Krishaan Chaudhary & Zuhayr Rashid

April 2025

Contents

1 Introduction & Problem Representation

This project involves designing different strategies for a bot to localize itself in a maze. Initially, it knows that it can be in any one of the open cells in the maze. However, moving around in the ship, it can narrow down its set of current possible locations, L down to 1.

We are working with 30×30 grids.

For the purposes of visualization:

- Black squares represent closed cells (walls)
- White squares represent open cells (cells the bot can move into)
- Blue square represents the cell the bot occupies

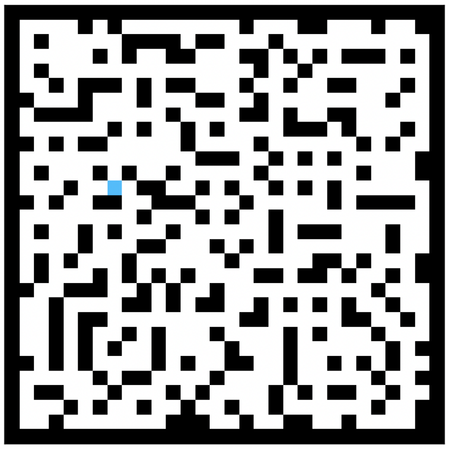


Figure 1: Initial Map

1. How many possible sets L are there?

   Let $|L_0|$ = the number of open cells in the maze

   A cell can be open or closed, leaving us with two choices for each cell.

   Therefore, there are a total of $\boxed{2^{|L_0|}}$ distinct sets L.

2. How can we update the set of possible sets L_{next} based on a current set L and an attempted move M?

Algorithm 1 Update Location Set After Move

```
Require: A set of possible locations $L$, a grid map ship, and a move direction
    move = $(dr, dc)$
Ensure: Returns the updated set of possible locations $L'$
 1: Initialize an empty set $L_{\text{next}}$
 2: for each cell $(r, c) \in L$ do
 3:     $nr \leftarrow r + dr$
 4:     $nc \leftarrow c + dc$
 5:     if ship$[nr][nc] = 1$ then
 6:         Add $(r, c)$ to $L_{\text{next}}$ {Move blocked by wall, remain in place}
 7:     else
 8:         Add $(nr, nc)$ to $L_{\text{next}}$ {Move successful}
 9:     end if
10: end for
11: return $L_{\text{next}}$
```

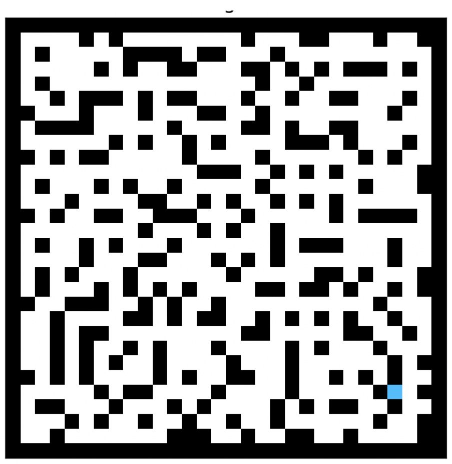
The overall idea is that we attempt moving in the specified move direction from every possible cell in L. If the move in that direction is successful, i.e. not blocked by a wall, we add the new location to L_{next}. Otherwise we add the original location to L_{next}.

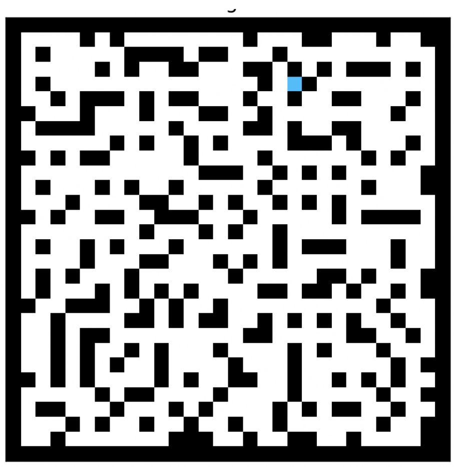
3. Let $C^*(\text{Map}, L)$ be the minimal number of steps needed to localize the problem (reduce it to one possible location for the bot) given Map and current locations L. For some sets L, $C^*(\text{Map}, L)$ is immediate to calculate:

   (a) L s.t. $|L| = 1$

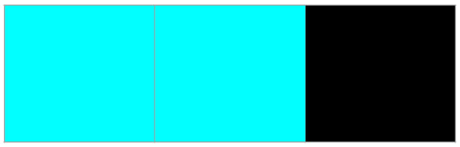
   If we have already narrowed down our set of locations to one location, we are done, so $C^*(\text{Map}, L)$ can be immediately deduced to be 0.

   Examples below:





(b) L s.t. $|L| = 2$ and L contains only 2 adjacent cells and one of the cells has a wall on the side it is not adjacent to the other cell.



Without loss of generality, we can think of the example above. We have 2 possible open cells, both adjacent to each other. In the same direction of the adjacency, one of the possible cells is adjacent to a wall on the side it is not adjacent to the other possible cell. In essence, if we see blue, blue, black type pattern in a 1x3 subsection of the grid, we know that $C^* = 1$. In this case specifically, we move right. This will narrow down the L to only be the right possible cell. This can be extrapolated in general for cases that have the property described.

4. Recursive formulation for C*:

$$C^*(\text{Map}, L) = 1 + \min \begin{Bmatrix} C^*(\text{Map}, L_{\text{up}}), \\ C^*(\text{Map}, L_{\text{down}}), \\ C^*(\text{Map}, L_{\text{left}}), \\ C^*(\text{Map}, L_{\text{right}}) \end{Bmatrix}$$

where $L_{\text{direction}}$ represents the set of possible locations after moving in direction.

5. Neither a finite nor infinite time horizon problem. Why?

   This is not a finite time horizon problem because there is not specific number of steps for which we can localize the bot to exactly 1 location for all the various Maps and L. It is heavily dependent on the shape of each map and the current L.

   This is not an infinite time horizon problem because the way we generate the map is a way that has deadends, corners and walls throughout. Because of this, and also considering that $|L_{\text{next}}| \leq |L|$, it is reasonable to conclude that eventually, $|L|$ will converge to 1 with decent planning and strategy.

6. Give a rough argument why, for maps of our type, it is exceedingly unlikely if not impossible to have a situation where $C^*(\text{Map}, L)$ is infinite.

   As described in the previous question, because of the presence of deadends, walls, and corners, if we move around the maze and use the update rule to transition from L to L_{next}, the size of L will always either decrease or stay the same. Also, the maze itself is finite and has walls around it. There are a finite number of possible locations to start with, and moving around will always reduce or have no effect on the updated L's size after moving. This means that it is very unlikely that we will end up in a situation where it is impossible to localize the bot and therefore it is very unlikely that $C^*(\text{Map}, L)$ is infinite.

7. Argue that if the function C^* were easy to calculate, then for any given set of locations L, it would be easy to determine what the next action you should take should be.

   Because of the recursive formulation from above, if C^* were easy to calculate, then it would be easy to determine the next action. For example, if my initial state is L_0, and I know $C^*(\text{Map}, L_0)$, I can easily determine the next move using the following approach:

$$C^*(\text{Map}, L_0) = 1 + \min \begin{Bmatrix} C^*(\text{Map}, L_{\text{up}}), \\ C^*(\text{Map}, L_{\text{down}}), \\ C^*(\text{Map}, L_{\text{left}}), \\ C^*(\text{Map}, L_{\text{right}}) \end{Bmatrix}$$

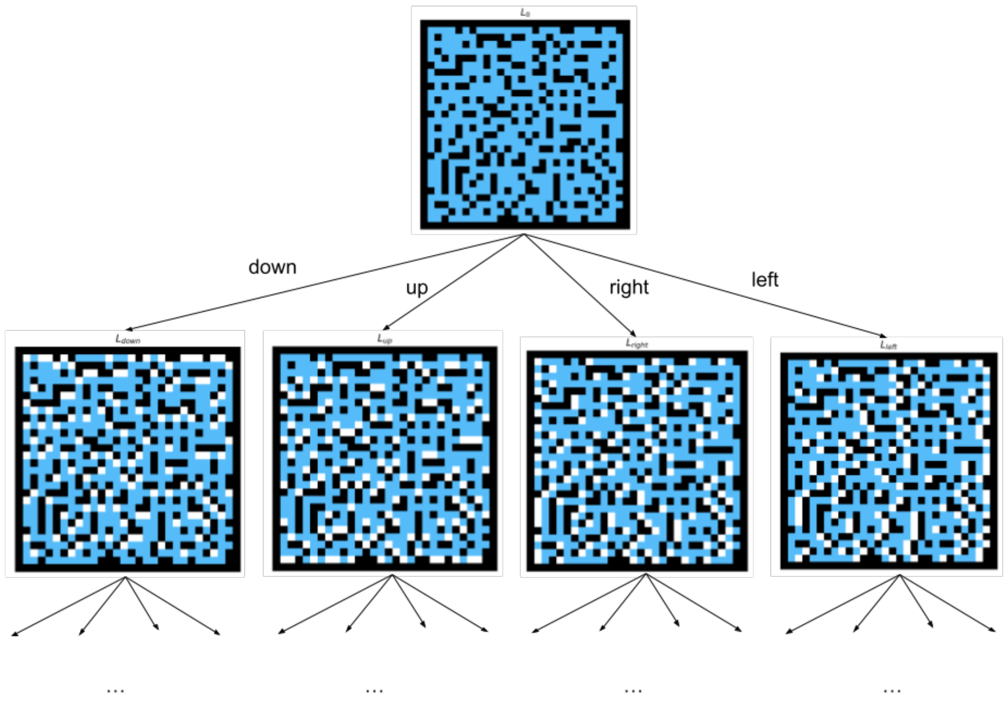
   Replace the LHS with the actual value for $C^*(\text{Map}, L_0)$ since it is easy to calculate.

   For each of the $C^*(\text{Map}, L_{\text{direction}})$ on the RHS, since we can easily calculate C^*, we also have those values. So, in order for the equality to tbe true the optimal move is one corresponding to the minimum of the 4 C^* values on the RHS. This leverages the recursive formulation of the problem.

   We make the optimal move, then replace L_0 on the LHS with $L_{\text{direction}}$ of the move we just made, and repeat the process.

8. How could we apply graph search techniques to this problem?

   We can represent states as nodes, where each state is a set L. We can represent transitions as follows. A directed edge exists from node A to node B if and only if we can reduce L_A to L_B by moving up, down left or right. We can also label the edge with the corresponding move.



   We can then apply graph search. Our initial position would be our starting L (a sample starting L is shown above. We can explore this graph and look for terminal nodes - nodes such that $|L| = 1$. Our goal is to find the shortest path from our starting L to any terminal node. So we can leverage techniques like Breadth-First Search (BFS), Depth-First Search (DFS), or A* to find the shortest path, which would give us our optimal sequence of moves to localize the bot. We could also consider pruning off certain branches if they do not seem to be helpful.

9. What might be a good heuristic for A* here?

   The number of moves to reach the goal is correlated with the size of L, making it a good heuristic for A*. For smaller L, we are closer to localizing, and for larger L, we have a lot more remaining steps.

$$\boxed{h(L) = |L|}$$

10. How should we understand something like A^*, relative to the problem of finding C^* above? Are they completely separate approaches to the problem, or is there a connection between them? Be clear and explicit.

    There is a connection between them.

    C^* = the length of the shortest path computed by A^* between our initial node and any target node.

    In our graph representation, we have an initial node, which corresponds to our initial set L. We have many terminal nodes, which correspond to L's of size 1. We can use A^* to compute the number of moves to to get from our initial node to any of our terminal nodes. C^* is the optimal number of moves to localize our bot to exactly 1 cell. So, we can use A^* to determine the shortest path to all of the terminal nodes, and choose to go to the terminal node that takes the fewest overall steps.

2 Machine Learning

2.1 Considerations & Architecture

1. **Input Space**: How do you want to represent L in a meaningful way?

   $L \in \{-1, 0, 1\}^{m \times n}$ is represented as a 2D grid of size $m \times n$, where:

   - 0 denotes an open cell,
   - 1 denotes a blocked cell (wall),
   - -1 denotes a possible location of the robot.

   This spatial representation allows the model to reason over possible locations and obstacles in the ship.

2. **Output Space**: What is the output of the model?

   The model outputs a scalar:
   $$\hat{y} \in \mathbb{R}$$
   which represents the expected number of moves required to localize the robot.

3. **Model Space**: How can you construct a model that is meaningful to the problem in some way? What parameters or decisions does your model depend on?

   Let $f_\theta : \{-1, 0, 1\}^{m \times n} \to \mathbb{R}$ be a convolutional neural network parameterized by θ:
   $$\hat{y} = f_\theta(L)$$
   where θ includes the learned convolutional filter weights and biases. We use regression, not classification since $\hat{y}$ is a real number.

4. **Loss Function**: How to measure error?

   Since we are solving a regression problem, the most natural and standard choice here is the Mean Squared Error (MSE) loss, which penalizes large deviations more heavily than small ones.

   Mean Squared Error (MSE) loss:
   $$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \left(f_\theta(L_i) - y_i\right)^2$$
   where y_i is the true number of moves for the i-th example, and N is the number of training examples.

5. **Training Algorithm**: What algorithm will produce the best model from the model space?

   We use stochastic gradient descent, using Adam optimizer to do so. Using a learning rate of lr=0.001 and running for 8 epochs, we ensure convergence to a good model from the model space. Through experimentation, we determined that 8 epochs was good enough so that we saw a good decrease in the loss function, and any further epochs did not provide much marginal improvements in loss.

Algorithm 2 Train CNN to Predict Remaining Moves

```
Require: Training data train_loader, test data test_loader, learning rate
    $\alpha$, epochs $N$, normalization bounds $y_{\min}, y_{\max}$
 1: Define CNN model:
 2:     Conv2D(1, 32, kernel=3, padding=1), ReLU
 3:     Conv2D(32, 64, kernel=3, padding=1), ReLU
 4:     MaxPool2D(kernel=2)
 5:     Conv2D(64, 128, kernel=3, padding=1), ReLU
 6:     Flatten
 7:     Dropout (0.5)
 8:     FC($128 \times 15 \times 15$, 256), ReLU
 9:     FC(256, 1)
10:
11: Set optimizer to Adam with learning rate $\alpha$
12: Set loss function to Mean Squared Error (MSE)
13:
14: for epoch = 1 to $N$ do
15:   Set model to training mode
16:   Initialize total loss to 0
17:   for each batch (inputs, targets) in train_loader do
18:     Zero gradients
19:     outputs ← model(inputs)
20:     Compute loss: loss ← MSE(outputs.squeeze(), targets)
21:     Backpropagate: loss.backward()
22:     Update weights: optimizer.step()
23:     Accumulate loss
24:   end for
25: end for
```

6. **Discussion of Network Layers**: Since the map has a spatial structure, a convolutional neural network (CNN) is well-suited for capturing spatial dependencies and patterns across the grid. The model begins with two convolutional layers, each followed by ReLU activations. With these layers, we hope to extract local spatial features such as edges, corners, and the density or spread of possible locations. A pooling operation follows to reduce spatial dimensions and promote translation invariance. The third convolutional layer, captures higher-level spatial features. The output is flattened and passed through a dropout layer to prevent overfitting. Finally, two fully connected layers convert spatial features into a single scalar output representing the estimated number of steps to localize.

3 Bot π_0

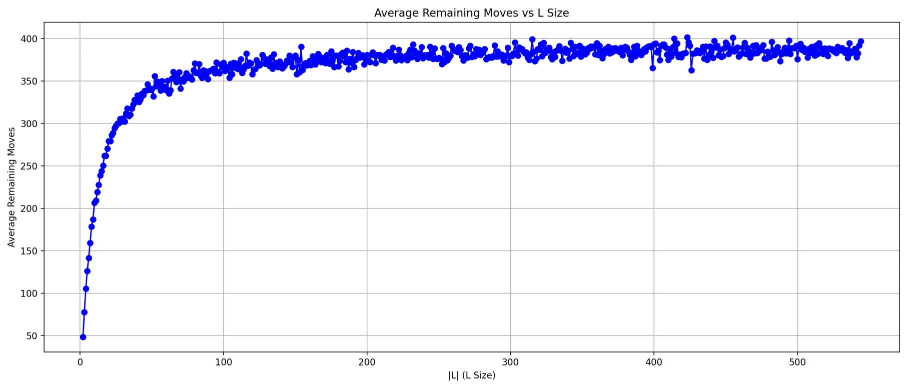
3.1 Data Generation

In order to generate data for the model, we used the following procedure:

For each value of n from 2 up to the total number of open cells in the map:

- Generate 150 random sample maps such that $|L| = n$
- Run the strategy on each map and record the number of moves required
- Store each result as a labeled data pair:
  $\text{data}_i = \text{map}_i, \quad \text{label}_i = \text{moves}_i$

This is what it looked like when we plotted a graph of Average Number of Moves Required vs. $|L|$ for strategy π_0:



3.2 Train and Test Data

Our goal was to generate a balanced training and test data that ensured reasonable representation of various $|L|$ and various labels (number of moves needed to localize).

Out of the 81450 total data points generated as described above (543 distinct $|L| \times 150$ samples per $L = 81450$), we noticed that most data points had a number of moves to localize between 275 and 550. What we noticed in our first few attempts of training a model is that it would almost always predict a value in this range because of how common it was in the data. In order to ensure that the sheer amount of data with this label did not skew the model, we crafted a more equal distribution of various n where n = # of moves to localize, using the following approach:

1. Split the data set into buckets of size 25:
   (a) Bucket 1 = 0-25
   (b) Bucket 2 = 25-50
   (c) ...
   (d) Bucket k = 25(k-1) - 25k
2. Cap overrepresented buckets
   (a) If the range was in between 275 and 350, cap the samples to be at most 700.
   (b) If the range was in between 350 and 550, cap the samples to be at most 500.

   These numbers were determined with a bit of playing around and testing after noticing that these samples contained thousands of data points in the original data set.
3. Train-Test Split:
   (a) Within each bucket, take 80% of the data for training and 20% for testing

   Overall, this came out to $\sim$ 8000 training data points and $\sim$ 2000 test data points.

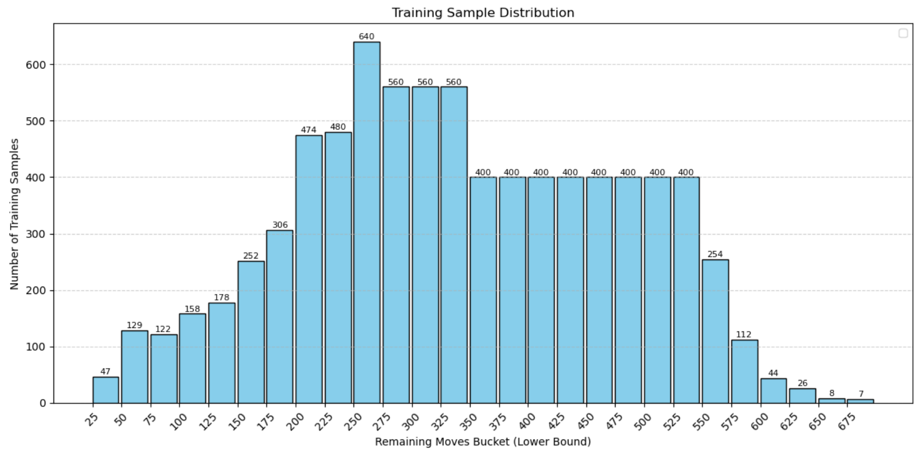


Figure 2: Data Distribution

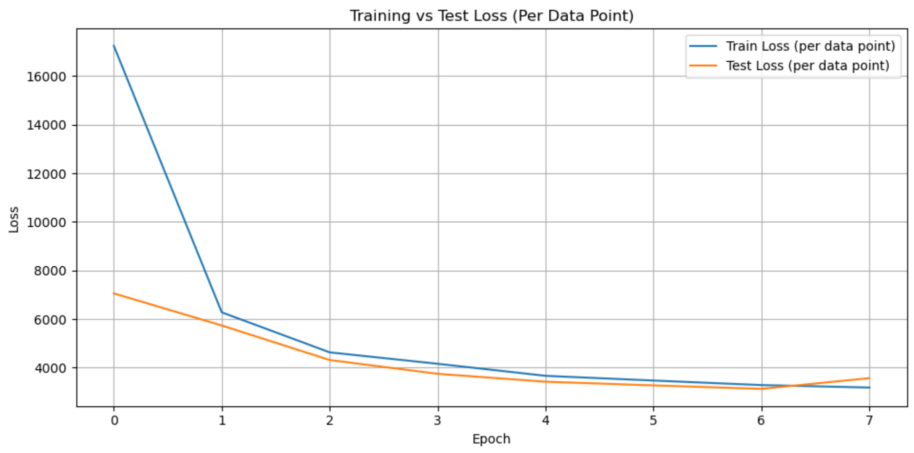
Discussion of Alternative Approaches:

We had tried alternative methods of data generation. For example, one was to always start with $L =$ set of all open cells in ship, and track the steps remaining for all intermediate maps across one run of localization. These intermediate maps were stored as data, and intermediate steps remaining as labels. We then repeated this for many runs of localization, and got a data set that had a lot of realistic L.

However, we noticed that many L, especially between the 300 and 500 range lacked any data because of how unrealistic it was to encounter them naturally if we were to start with an L of all open cells then move around. We considered augmentation to this data set by manually randomly sampling L within those range. This brought up an interesting consideration of the problem itself. Is the essence of the problem to be able to predict number of moves on any L or to localize assuming L starts out being the number of open cells. We decided to prioritize the first interpretation of the problem.

After experimenting with model training and these strategies, both of these approaches were not as effective as our eventual approach detailed above.

4. **Training and Testing Curves**



Overall, we see that both training and test loss is decreasing over time, and levels out by epochs 6-7.

In order to reduce / prevent overfitting, we did the following:

(a) **Limit number of epochs model runs**: After a certain number of epochs of training, the model no longer extrapolates high-level important patterns, but rather, it will start fitting to the noise in the data set. We cut off training once we noticed a drop off in the marginal reduction of loss from epoch to epoch.

(b) **Ensure variability + size in training data set**: We ensured that during the data generation process, we had a lot of different data in a distribution that does not heavily bias towards specific types of data or labels, so that the model does not only learn patterns from some niche cases.

(c) **Dropout Layer**: We introduced a dropout layer, which randomly deactivates certain neurons. This helps the model learn more reliable features without over reliance on a specific neuron.

(d) **Smaller Convolutional Layers**: Typically, Neural Networks are more prone to overfitting as the size of the layers increases, which is proportional to overall model complexity. More complex models are likely to fit to noise and intricate patterns in the data labels, rather than generalizing well. We chose smaller amount of parameters in the convolutional layers to prevent this.

5. **Graphs and Interpretations**

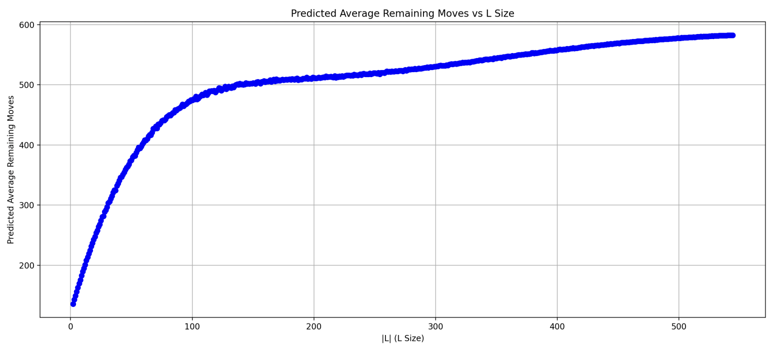


Figure 3: Graph of Average Predicted Number of Moves based on Model Trained vs. $|L|$

Comments: What we see in this graph is that typically the model tends to overestimate the number of moves relative to the size of L.

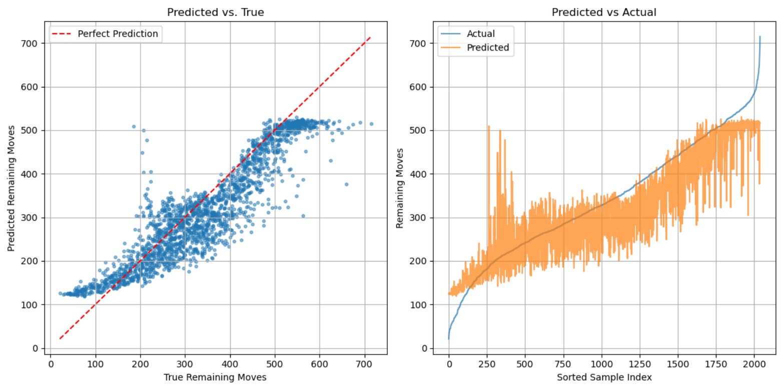


Figure 4: Scatterplot of Predicted Prediction vs Actual Number of Remaining Moves (Left), Actual vs. Predicted Plot Sorted by Actual Remaining Moves (Right)

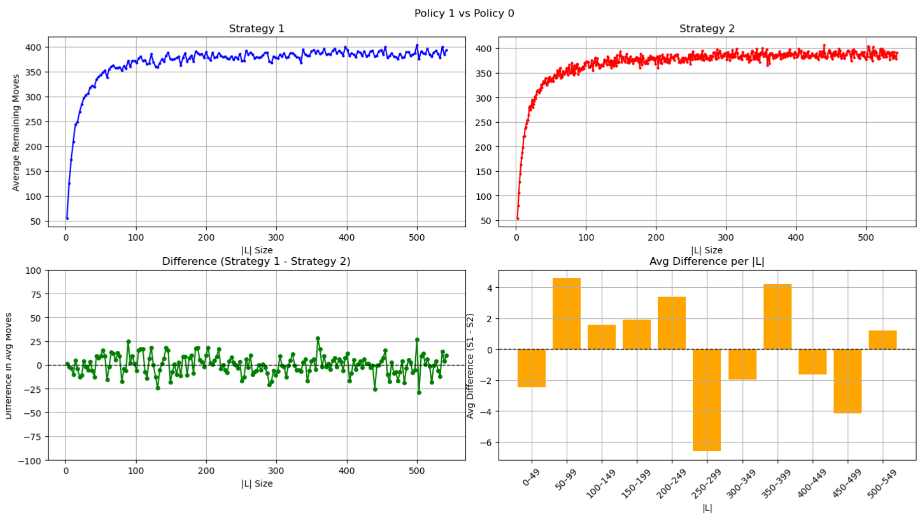
Comments: What we see in these graphs is that the model can extrapolate well when the true number of remaining moves is between 200 and 500, but it hesitates to predict remaining moves outside this range. One potential explanation is that it does not have much training data where the predicted number of remaining moves is that low or that high, so it tends to be somewhat biased towards patterns it sees in large portions of the data.

4 Bot π_1

Approach: Bot π_1 does the following:

1. It takes a starting L, and sees what is the best first move available to it, by evaluating $\hat{C}^*(L_{\text{up}}), \hat{C}^*(L_{\text{down}}), \hat{C}^*(L_{\text{left}}),$ and $\hat{C}^*(L_{\text{right}})$.
2. It evaluated each $\hat{C}^*(L_{\text{direction}})$ using the model trained above (described in Bot π_0.
3. Then, it defaults back to π_0.

Comparison of Bot π_0 and π_1

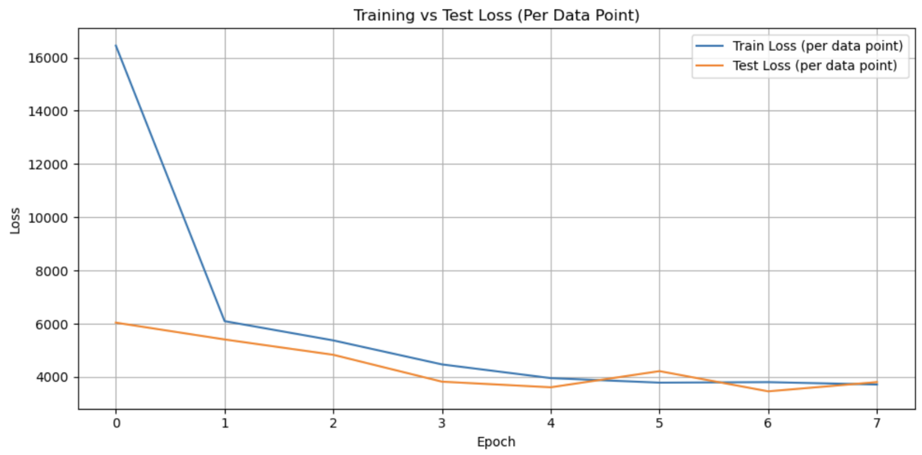


Strategy 1 = Bot π_0
Strategy 2 = Bot π_1

We notice that for some $|L|$, Bot π_1 outperforms Bot π_0. Specifically, looking at the fourth pot in the figure above, which plots the difference in strategy for various ranges of $|L|$, we see that |Moves taken by Bot π_0 - Moves taken by Bot π_1| tends to be positive for 6 of the 10 ranges of $|L|$ (indicating Bot π_1 has better performance) for L of those ranges. Particularly, we notice improvement for L of size 50-250.

Overall, there is not too much difference in performance. This can be explained by a few reasons. First, this strategy is only looking one move into the future, so picking the best first move based on predicted length, may not have a significant effect on the overall execution steps of the policy. Another consideration is that it is possible that the optimal first move calculated by this strategy is nullified by the switch back to the second phase of the strategy, which reverts to π_0, in which the bot repeatedly plans paths to a specific target cell. It is possible that the target cell the Bot is targeting is not a great target cell, undoing the potential benefit of making a good first move.

4.1 Training and Testing of Bot π_1 Data



Comments: Overall, we see that both training and test loss is decreasing over time, and levels out around epochs 6-7.

We then trained a model to predict how many moves Bot π_1 would take using the CNN architecture from before.

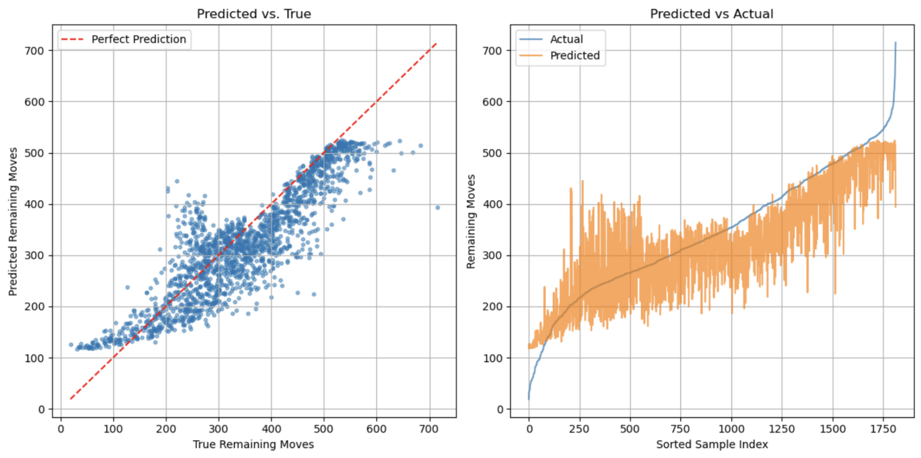


Figure 5: Scatterplot of Predicted Prediction vs Actual Number of Remaining Moves (Left), Actual vs. Predicted Plot Sorted by Actual Remaining Moves (Right)

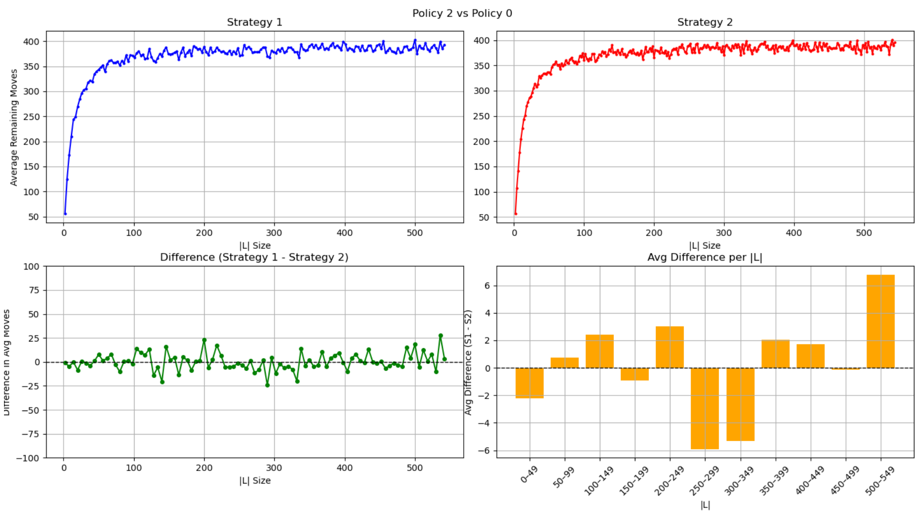
Comments: What we see in these graphs is that the model can extrapolate well when the true number of remaining moves is between 200 and 500, but it hesitates to predict remaining moves outside this range. One potential explanation is that it does not have much training data where the predicted number of remaining moves is that low or that high, so it tends to be somewhat biased towards patterns it sees in large portions of the data. The problem is compounded since strategy π_1 builds off of previous model predictions.

5 Bot π_2

Approach: Bot π_2 does the following:

1. It takes a starting L, and sees what is the best first move available to it, by evaluating $\hat{C_1}^*(L_{\text{up}}), \hat{C_1}^*(L_{\text{down}}), \hat{C_1}^*(L_{\text{left}}),$ and $\hat{C_1}^*(L_{\text{right}})$.
2. It evaluated each $\hat{C_1}^*(L_{\text{direction}})$ using a model trained on the data generated by Bot π_1.
3. Then, it makes an optimal first move.
4. After, this it defaults back to strategy π_1.

Comparison of Bot π_0 and Bot π_2



Strategy 1 = Bot π_0
Strategy 2 = Bot π_2

We notice that for some $|L|$, Bot π_2 outperforms Bot π_0. Specifically, looking at the fourth pot in the figure above, which plots the difference in strategy for various ranges of $|L|$, we see that |Moves taken by Bot π_0 - Moves taken by Bot π_1| tends to be positive for the range of L with the largest size $|L|$ (indicating Bot π_2 has better performance) for L of those ranges. Particularly, we notice improvement for L of size 500-549.

This is promising because it means that Bot π_2 is doing particularly well when it starts with $|L|$ close to a realistic starting L of all the open cells being possible cells. It also seems to do well once it has already narrowed down to between 50-200 cells. However, it struggles in cases where $|L|$ is moderately sized (250-350) and in endgame like scenarios (0-49).

Overall, there is not too much difference in performance. This can be explained by a few reasons. First, this strategy is only looking one move into the future, so picking the best first move based on predicted length, may not have a significant effect on the overall execution steps of the policy. Another consideration is that it is possible that the optimal first move calculated by this strategy is nullified by the switch back to the second phase of the strategy, which reverts to π_0, in which the bot repeatedly plans paths to a specific target cell. It is possible that the target cell the Bot is targeting is not a great target cell, undoing the potential benefit of making a good first move.

6 Bot Reinforcement Learning

6.1 Bot π_k, Policy Iteration, Bootstrapping

In this section we discuss some results of continuing the iteration in the style of Bot π_1 and Bot π_2.

As a summary:

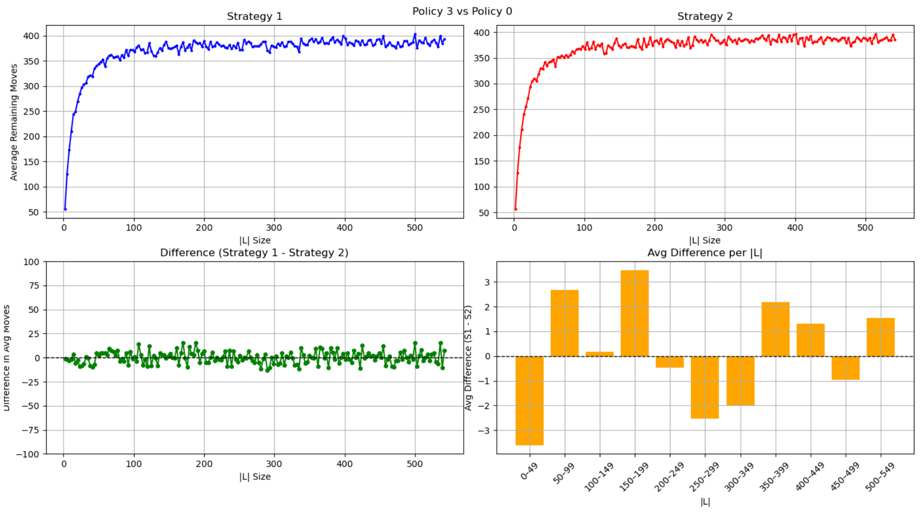
Bot π_1 takes a best first move based on model trained using π_0, and then reverts back to π_0.

Bot π_2 takes a best first move based on model trained using π_2, and then reverts back to π_1.

If we continue this process up till π_k, we would expect to see improvements in the bot.

We repeated the process described for π_1 and π_2 where we build upon the previous strategies to train and test a model on π_3

Here are the results we had, when expanding this approach to Bot π_3.



Strategy 1 $= \pi_0$
Strategy 2 $= \pi_3$

We notice further improvements for some ranges of $|L|$, such as between 50 and 200, 300 and 400, and over 500. This shows improvement in the policy as we iterate. For the 500+ case, these are L that are very close to having all open cells be possible cells, which makes it very reflective of localization from a starting set of all open cells. The positive performance of the 300-400 range is also a good sign because in many cases of localization, after making one move, the bot will narrow down its set of possible locations to a set of size in this range. It is possible that the bot seems to be doing better in the case of L's it is likely to encounter during localization processes that start out with $L =$ all open cells.

However, overall, there is not too much improvement in number of steps as evidenced by a marginal difference in average number of moves per $|L|$. Also, the strategy does continue to struggle in the endgame-like scenarios of 0-49 cells remaining.

Reinforcement Learning Strategy:

Initial Dataset Collection (π_0) We begin by collecting an initial dataset using the baseline policy π_0, which repeatedly plans paths from one of the possible cells to a predetermiend target cell until localization is achieved or a step limit is reached. We keep track of the number of moves needed to localize.

Training Initial Model $\hat{C}_0^*$ Using the dataset generated from π_0, we train an initial neural network model $\hat{C}_0^*$ to predict the expected number of steps to localization from a given belief map. The model is trained to minimize a Bellman-Equation like loss:

$$\mathcal{L} = \left(\hat{C}^*(L) - \min_{d \in \{U,D,L,R\}} \left[1 + \hat{C}^*(L_d)\right]\right)^2$$

This loss encourages the model to produce predictions consistent with expected future costs under possible actions.

Improved Policy π_1 After training $\hat{C}_0^*$, we define a new greedy policy π_1 as follows:

$$\pi_1(L) = \arg \min_{d \in \{U,D,L,R\}} \left[1 + \hat{C}_0^*(L_d)\right]$$

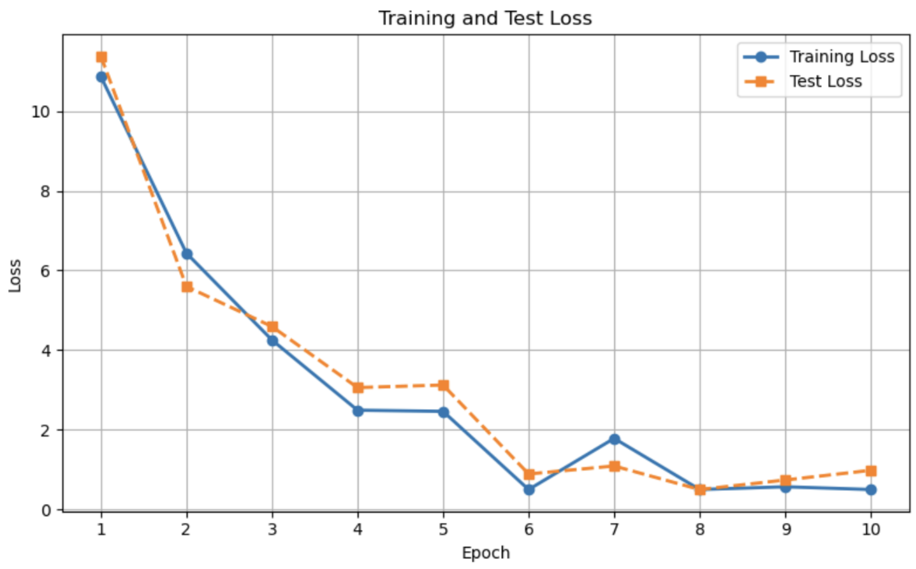
This policy selects the action that minimizes the predicted number of steps remaining until localization.

Generate Data based on π_1

Train a model $\hat{C}_1^*$ **to minimize the loss function**

Iterate this process until convergence

Discussion



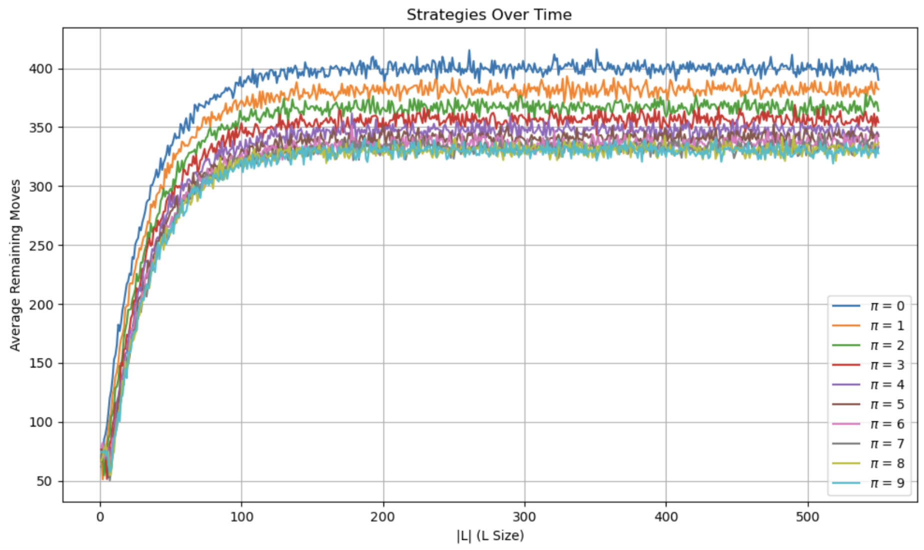
We would expect to see training and test losses like the one above. The number of optimal moves at a given L should be equal to 1 + the number of optimal moves at any of the L's that would occur by taking a move in any direction. The mean squared error between these two values should approach zero as the strategy improves.

How to handle cases where π_1 does not solve the problem

In cases where π_1 does not solve the problem, first we can ensure early termination - exit after the strategy has tried 1000 moves. If the strategy takes 1000 moves, meaning it was unsuccessful, we can assign a some arbitrary high cost, X to the corresponding L. This will help the model learn in future iterations to avoid L's like this because they will be predicted as having a high cost. It is important to carefully select a value of X that pushes the model away from unhelpful L without causing instability or overfitting.

Why is novel data generation necessary at each policy update?

Novel data generation at each policy is important for the following reasons. Each policy has distinct types of situations, decisions, and intermediate moves it will encounter. (It is also a good idea to use an approach that samples intermediate L's along with the corresponding moves remaining till localization so that each policy can learn better next states). Generating new data at each policy update ensures that each new policy learns the states of maps and value of choices based on the previous policy. This ensures that the model builds upon its decision making, iteratively improving. Also, it is because of the recursive nature of the problem, outlined above. We have a base case, and the next strategy will build off the data generated by the base case. The strategy after that will build off the data generated by case 1, and so on.



We would expect to see something like this for strategy π_k as we do more and more iterations of this reinforcement learning process, where the number of iterations decreases from policy to policy, eventually converging at an optimal policy.