Project 3 - MDPs and ML

Eric Lin, Rui Zhang

July 18, 2025

1 Introduction

This project focuses on developing a Markov Decision Process to help a bot catch a rat on a simulated maze spaceship. The bot has knowledge of the ship layout and the location of the rat at all times. The bot and rat alternate one-space moves, but the rat moves randomly while we the designer must select moves for the bot. Therefore, we will be implementing an MDP to guide the bot in selecting the optimal moves for the bot to catch the rat in as few moves as necessary. Finally, we will train a machine learning model to be able to predict the expected number of moves needed to catch the rat.

2 Methodology

2.1 Ship Generation

The ship generation process follows the same procedure as in Project 1. We added a field `ratCell` to the Ship class to store the location of the rat. A rat is generated and randomly placed when the ship is created. To ensure consistency across different bot runs, we use the same seeds when generating ships.

2.2 Markov Decision Process

To guide the bot in selecting optimal moves that minimize the expected number of steps to capture the rat, we model the problem as a Markov Decision Process (MDP). An MDP consists of four components:

1. State Space S: our state space is defined by all possible configurations of bot and rat positions on the ship: (b_x, b_y, r_x, r_y).

2. Action Set A: our action set $A(x, y)$ is defined by all valid moves $a \in A(x, y)$ a bot or rat can take from position (x, y) into adjacent open cells. Since movement is restricted to the four cardinal directions (up, down, left, right), $A(x, y) \leq 4$. Let $A_{bot}(x, y)$ be the action sets for bot moves and $A_{rat}(x, y)$ be the action sets for rat moves.

3. Transition Probabilities: the rat moves uniformly at random, so for any move $a \in A(x, y)$, the probability $P(a)$ from one state to another is $\frac{1}{N}$ where N is the number of open neighbors from the rat's current position (x, y). For the bot, we will be deciding its moves based on the smallest T value among its open neighbors.

4. Cost Structure: our cost is defined by the number of moves the bot takes to catch the rat. Therefore, for all positions (x, y), the cost $c^a_{(x,y)}$ for any given action a is always 1.

Given the above state space S, action sets $A(x, y)$, transition probabilities, and costs, we can compute for any state (b, r) where b is the bot coordinates and r is the rat coordinates, the optimal utility:

$$U^*(b,r) = \begin{cases} 0, & \text{if } b = r, \\ 1, & \text{if ManhattanDist}(b,r) = 1, \\ \min\limits_{b' \in A_{bot}} [1 + \frac{1}{N} \times \sum\limits_{r' \in A_{rat}} U^*(b', r')], & \text{otherwise} \end{cases} \tag{1}$$

The above equation represents the base cases (easiest to compute T values) and the recursive case. We know immediately that if the bot and rat are on top of each other, the game should be over and the expected utility, or number of moves is 0. Similarly, if the Manhattan Distance between the bot and the rat equals 1, that means the bot and rat are directly next to each other. As such, 1 move is all it takes to reach the rat, so the expected utility is 1.

In the recursive case, we essentially need to compute the immediate cost of a single bot move plus the future expected cost because we changed coordinates by moving and the rat did so as well. For the immediate cost, the value 1 the bot making the first move. The set A_{bot} includes all possible actions the bot can take from its current position and b' is their resulting coordinates. A_{rat} includes all possible actions the rat can take from its position and r' is their resulting coordinates. The inner summation calculates the future expected utility over all possible rat actions given each possible bot action. The $\frac{1}{N}$ is because we assume the rat moves uniformly at random to one of its N adjacent open cells. Over all those actions, we then select the smallest expected cost from a given bot move, resulting in the fewest steps needed to catch the rat.

Additionally, although our MDP operates under a finite horizon, the exact maximum number of steps t required for the bot to capture the rat and terminate the game is unknown. As such, we can instead approximate the optimal utility function by initializing with a heuristic estimate and applying value iteration until convergence. Specifically, we use the Manhattan distance between the bot and the rat as the initial estimate for all state configurations, as it provides a reasonable bound on the expected number of steps. The value iteration procedure then updates these estimates iteratively according to the equation above until the maximum change in utility across all states falls below a convergence threshold that we set to 10^{-6}.

Once we have T fully calculated, we can determine the optimal action for the bot to take in every configuration. We do this in the method **makeOptimalMove** where we simply go over every current bot action, access their resulting expected cost from the T array, and then perform the move with the smallest T value.

> Notably, the coordinates of blocked cells cannot contain the bot or rat. Therefore their T values are also immediately easy to compute as $+\infty$, effectively representing unreachable states. However, in our implementation, we left them at Java default 0.0. This is because in our code, we explicitly skip all blocked cells when initializing and computing T values. And when we iterate over possible moves of the bot cells to select from, we will never access those blocked cells. So although insignificant in saving computational time, we simply skip over them.

2.3 Training Model on Utility Function T

Once the expected number of moves $T(b, r)$ for all valid configurations of bot and rat positions has been computed using value iteration, we generate a training dataset consisting of input–output pairs $(\mathbf{x}, t)$, where:
.

- $\mathbf{x} = (b_x, b_y, r_x, r_y)$ is the 4-tuple of coordinates,

- $t = T(b, r)$ is the expected number of moves from that state.

Each $(\mathbf{x}, t)$ pair is treated as a supervised training data point, and the goal is to learn a function $f_\theta(\mathbf{x}) \approx t$ that approximates the optimal value function T, using a parameterized neural network f_θ with weights θ.

Note that the current model is trained specifically for a single, fixed ship layout. Both the input features (b_x, b_y, r_x, r_y) and the target t-values are computed with respect to that specific ship's dimension and set of blocked or open cells. As a result, the learned model will not generalize to different ship configurations with different structures or open-close cell arrangements.

Therefore, we consider extending the input space to include information about the ship layout itself. By providing the model with a representation of the ship layout, it may be possible for the neural network to learn not only spatial relationships between bot/rat positions, but also how those relationships interact with the structure of the ship. This could lead to a more general model that is able to make predictions across different ship layouts, and be more precise in predicting since more relevant information is given.

2.4 Training Model - CNN

Motivation: To improve generalization across different ship layouts, we extend the input to include the structure of the ship itself. Rather than inputting only the bot and rat positions as coordinates, we use a convolutional neural network, which performs well when processing images and presumably two-dimensional structured data such as our 2-D ship map.

The input to the model is represented as a three-layer tensor of shape $[3, 30, 30]$, which are three 30x30 grids (s, b, r). More details in Section 3.3.

We use a simple convolutional structure consisting of two convolutional layers (with ReLU activations, same as the previous NN model). The model is trained using mean squared error (MSE) loss between predicted and true t-values.

Basically there are two questions that I think worth exploring:

- If we train a model to cater for a single ship, how much will it improve the accuracy of the model by adding the ship layout as additional input?
- If we instead train the model on multiple ship layouts along with their corresponding configurations, can the model generalize well enough to make accurate predictions on unseen ships or configurations?

3 Experimental Setup

3.1 Setup for MDP

As previously mentioned, the state is represented by the positions of the bot and rat, (b_x, b_y, r_x, r_y), and the objective is to compute the minimal expected number of moves, denoted as $T(b_x, b_y, r_x, r_y)$, required for the bot to catch the rat. Both agents move alternately: the rat selects a neighboring open cell uniformly at random, while the bot chooses an optimal action to minimize the expected capture time. Therefore, we store the value function T as a 4-D array of size $D \times D \times D \times D$, where D is the ship dimension. Before computing T with value iteration, we give the initial guess of the Manhattan Distance between (b_x, b_y) and (r_x, r_y).

For each experiment, we will use dimensions $D = 30$ and $p = 0.5$ for consistency. For the sample size, we use the formula below where $z = 1.96$, $s =$ sample standard deviation, and $E = 0.05 \times \mu$. This calculates

the number of trials needed such that we are 95% confident that the observed sample mean is within 5% of the true mean. Using the formula, we approximate that roughly $N = 300$ trials would be sufficient.

$$N = (\frac{z \cdot s}{E})^2 \tag{2}$$

3.2 Setup for Training Model (Model A)

For a ship with N open cells, the will be N^2 possible configurations of the rat and bot's position with corresponding T values. Before training, we apply min-max normalization to both input features and target values to scale all quantities into the range $[-1, 1]$. This normalization improves training stability and convergence rate:

$$x_{\text{norm}} = 2 \cdot \frac{x - x_{\min}}{x_{\max} - x_{\min}} - 1, \quad t_{\text{norm}} = 2 \cdot \frac{t - t_{\min}}{t_{\max} - t_{\min}} - 1$$

The dataset is randomly split into training and testing sets with a standard 80/20 split. The test data will be used later to validate and evaluate whether the model is overfitting.
The model is as follows:

- Input layer: 4 neurons (bot and rat positions)
- Hidden layer 1: 64 neurons, ReLU activation
- Hidden layer 2: 32 neurons, ReLU activation
- Output layer: 1 neuron, linear activation (regression target)

We decided to include two hidden layers for the model. We use 64 neurons in the first hidden layer and 32 in the second. We also experimented with larger widths (e.g., 128 or 256 neurons per layer), but found that the performance remained nearly identical while the training time per epoch increased. This makes sense since wider layers introduce more parameters and thus more computation per forward and backward pass. Since our input dimensionality is low (only four features), we think the network is sufficient to grasp the feature and possibly without overfitting.

We use the ReLU activation function in all hidden layers:

$$\text{ReLU}(x) = \max(0, x)$$

We did try some other activation functions such as *tanh* and *sigmoid*. In all cases, the final predictive performance was similar, but ReLU consistently required fewer epochs to reach lower loss and took slightly less time for each epoch.
So essentially, the model computes:

$$f(\mathbf{x}) = W_3 \cdot \text{ReLU}(W_2 \cdot \text{ReLU}(W_1 \cdot \mathbf{x} + b_1) + b_2) + b_3$$

The network is trained to minimize the mean squared error between predicted and actual normalized t-values:

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} (f_\theta(\mathbf{x}_i) - t_i)^2$$

We use the Adam optimizer which is a variant of SGD, and set the the learning rate to $\alpha = 10^{-3}$. This value seems reasonable: the training and testing loss curves display smooth and steadily decreasing trends over the epochs, indicating that the model is learning stably without diverging or overfitting.

Training is performed over 100 epochs using mini-batches of size 320. The mini-batch strategy helps smooth out gradient updates and improves generalization compared to training on the full dataset, as we have discussed in class. The batch size was chosen to be large to produce stable gradient estimates, while still allowing for some randomness in the updates, which hopefully can help to avoid overfitting and local minima. Both training and testing loss are logged and plotted over time to visualize learning progress and check for potential overfitting.

Visualization and Evaluation Tools. To evaluate model behavior and performance, we generate several plots thought out:

- **Loss Curves:** We plot training and testing loss in MSE across all 100 epochs to check for convergence and potential overfitting.
- **Prediction Scatter Plots:** We visualize predicted vs actual t-values on the testing set, both before and after training. This idea came up thanks to TA Jack, appreciate it :)

3.3 Setup for Training Model - CNN on Single Ship (Model B)

Some thinking process of representation of input:

For the representation of ship layout, I let 1 indicate open and 0 to be blocked, so that we have a 30×30 binary matrix.

The representation for bot and rat position is a bit tricker. Originally we used (x, y), an integer pair to represent the location, but that has no spatial feature and I cannot imagine a way of doing CNN on a pair. Another thing is that it is hard to be bundled with the layout representation, like a dot and a plane.

I got some inspiration in the lecture when the professor mentioned how the words or tokens in natural language are represented in the 1-hot encoding so that I thought maybe I can do something like that. So binary representation, plus 30x30 format. At this point my option is rather limited and obvious: that we have the same 30x30 grid, use 1 to represent the position of bot/rat, the rest will be filled with zeros.

The input to the model is represented as a three-layers tensor of shape $[3, 30, 30]$, which are three 30x30 grids (s, b, r):

- Layer 1: s[a][b] = 1 if (a,b) is open; 0 if blocked;
- Layer 2: b[bx][by] = 1 if bot is at (bx, by), others are filled by zeros;
- Layer 3: r[rx][ry] = 1 if rat is at (rx, ry), others are filled by zeros.

Data Splitting and Preprocessing. We split the dataset randomly into training and testing sets using a 80/20 split. No normalization is needed because the values are already in a consistent range - binary. The target t-values are used in their raw form.

Model Structure The CNN model consists of two convolutional layers with 16 and 32 filters, each with a ReLU activation function for nonlinearity. (Since in the previous model we observed that ReLU is able to make pretty good results.)

We train the model using mean squared error (MSE) loss:

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N}\left(f_\theta(X_i) - t_i\right)^2 \tag{3}$$

with the Adam optimizer and a learning rate $\alpha = 10^{-3}$.

Batch Size and Epoch. The model processes 64 training samples at a time. Each batch consists of 64 input tensors of shape $[3, 30, 30]$, representing the layout, bot map, and rat map. Each of these is paired with a corresponding normalized t_value. We include 30 epoch for training, considering that CNN's model training is more efficient compared with the previous method. Plus each epoch took about 12 minutes on average, so a larger number of epochs will lead to significantly longer training time.

Evaluation and Visualization. We evaluate model performance using root mean squared error (RMSE) on the test set and visualize predictions with scatter plots:

- Training Loss Curve: A plot of MSE loss across training epochs to monitor learning progress and overfitting.
- Initial/Final Scatter: A pre/post training scatter plot of predicted vs. actual t-values

3.4 Setup for Training Model - CNN on Multiple Ships(Model C)

The setup structures are almost the same, with some modifications to allow training on multiple ships at once.

Instead of loading data from one JSON file, the new version takes a list* of JSON files (`ship_1_data_with_layout.json`, ...). Each file contains a different ship layout and a set of configurations (bot position, rat position, t value). All of these are combined into one dataset class (`MultiShipDataset`).

Since there are more total configurations when using multiple ships, I increased the batch size from 64 to 320 to speed up training and possibly make better use of resources. My reasoning on this is that since we are including more information, and the variation would be larger between each tensor – two tensors could be configurations with different ship layouts, thus increasing the batch size may potentially allow the model to observe the feature better.

I did a data split for training and testing here as well. I made sure of a random split so that there is no artificial bias.

I also added checkpoint support. After each epoch, the model and optimizer states, the current epoch, and the loss are saved. I did not realize how much I needed to add this until the jobs I submitted on iLab got interrupted and terminated midway through the training multiple times :/

*: To be specific we used 10 JSON files due to the time limit. I tried 30 input files and even on a virtual computing environment like iLab, which has much more computing resources, it will cost over an hour for each epoch; thus we reduce the size of the input.

4 Results and Analysis

4.1 MDP Results

Using $N = 300$ trials, we compute an average number of moves taken to catch the rat of **23.00** moves. Interestingly, if we compare these results with a greedy shortest-path approach, we see some similarities.

```
Average Moves: 23.533333333333335
Percentage: 94.75920679886686
```

Figure 1: Percentage of Optimal T Actions to Shortest Path Actions with 30 trials

Looking at Figure 1, we see that for $N = 30$ trials, the bot chooses the shortest path action 94.76% of the time. This makes sense since they basically do the same thing: trying to minimize the distance from the bot to the rat. However, the MDP approach takes into account the expected movement of the rat, while the shortest path approach only looks at the rat's current location at one point in time. This is likely why we see that 5.24% of the time, the MDP approach chooses actions different from the shortest path. This potentially allows it to perform slightly better than shortest path, although in our tests comparing them, their performance was roughly the same. Especially in denser maps with more obstacles, MDP accounts for the rat's probabilistic movements and may choose routes that reduce escape options or corner the rat, rather than blindly following the shortest distance.

4.2 T Value Insights

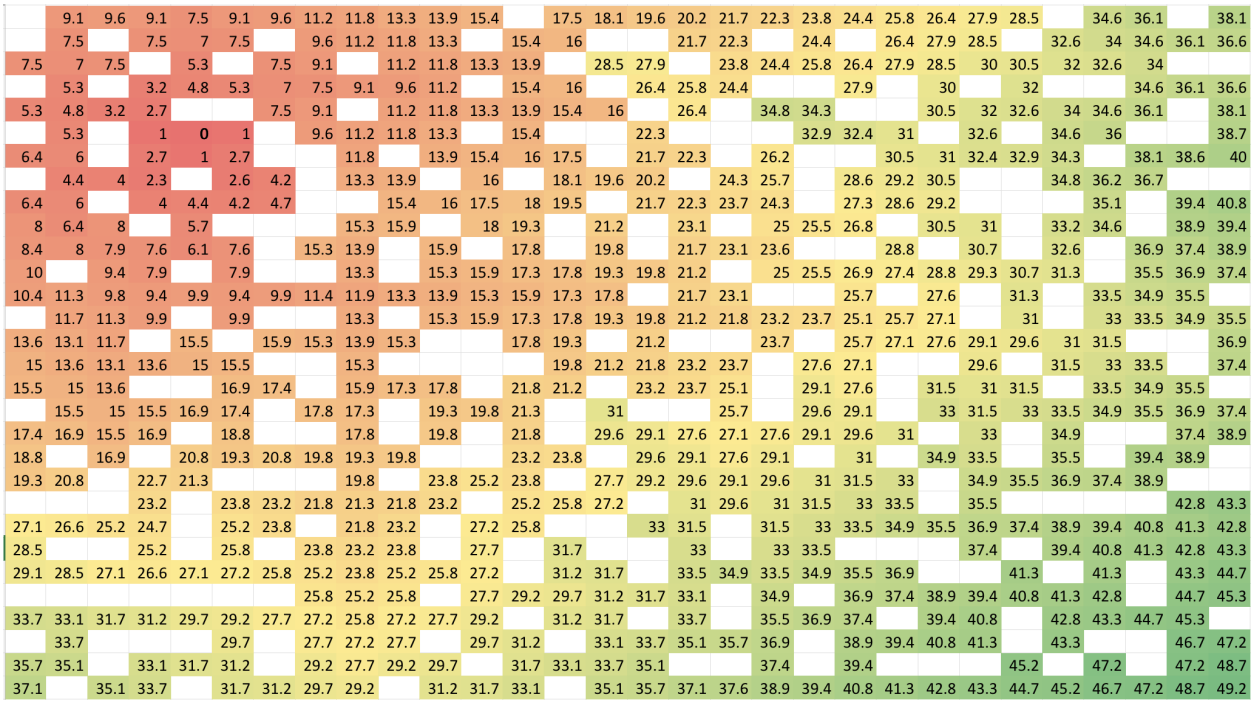

Figure 2: Heat Map of T for Random 30x30 Ship

Looking at Figure 2, we have generated a random ship with a rat randomly placed on it. The white spaces indicate blocked cells. We can see that for any bot location, the minimal expected number of moves T is 0 where the rat is because then the rat and bot would be on top of each other. Beyond that the T value increases outward from the rat's location with the maximum T value being at the furthest corner $(29, 29)$. The maximum T is also notably 49.2. Given a ship of $D \times D$ dimensions, the maximum T value possible should be less than or equal $D + D$. In our case, it should be less than or equal to 60 which it is, and the reason it is much less is because the rat is not in the opposite corner and we must take into account blocked cells.

Additionally, we can see that the heatmap is only one possible rat configuration. If N represents the number of open cells, then there are N possible locations the rat can be, and also N possible locations the bot can be. Therefore, we can conclude that there is $N \times N$ total number of bot-rat configurations, if we also include when the bot and rat are in the same cell.

4.3 Training Result - Single Ship (Model A)

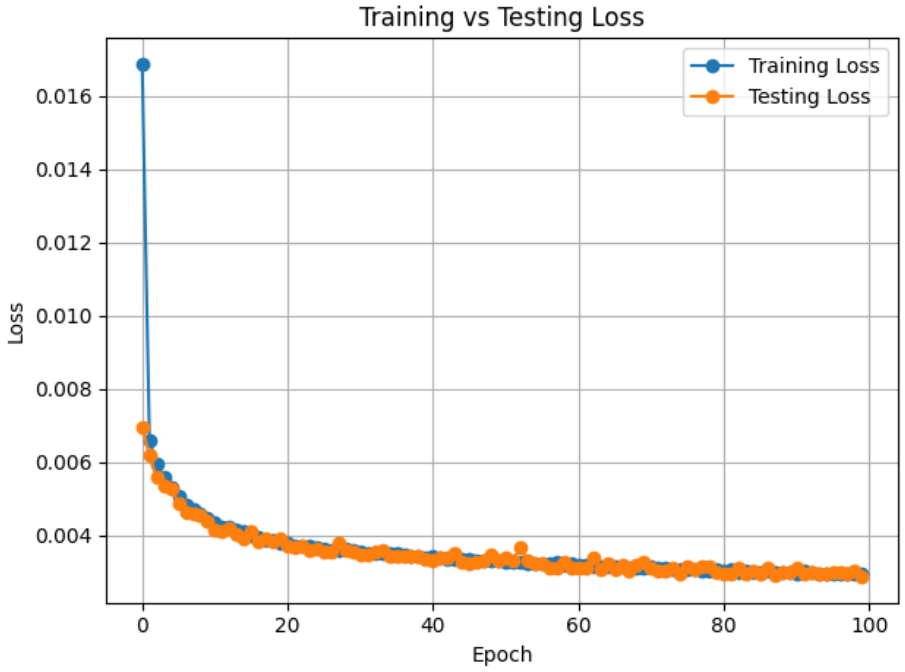

Figure 3: Model A loss curve

The training and testing losses (measured as mean squared error) decreased steadily during the first 20–30 epochs and gradually plateaued after that. Fig. 3 shows the loss curves over training epochs. The test loss remained very close to the training loss throughout. If the testing loss curve plateaued early, it would indicate that the model is likely overfitting on the training set. Instead, we see that they are aligning close, showing that our model is not simply memorizing the data points, nor overfitting to them.

To check the model's predictive performance on unseen data (the 20% test data we split out earlier), we compute the Root Mean Squared Error in the original scale of t-values, which will represent the number of steps to capture the rat. The final actual RMSE is 1.4041 steps. Since t-values can range broadly depending on rat distance and ship layout, an average prediction error of approximately 1.4 steps is small, showing that the network has learned what features are important to compute the value function $T(b, r)$ to some extent.

Note that the graph appears to drop significantly at first and then seems to flatten out. But we think this doesn't necessarily mean the model stops improving. One likely reason for that early sharp drop is that the initial predictions are completely random, since the model starts with no information. As a result, just after the first epoch, we already see a large improvement—essentially the jump from a random guess to the model's first meaningful update. This big initial change makes the rest of the curve look flat in comparison, but if we look closely, the later improvements are actually smooth and consistent, just on a smaller scale.

Another thing we considered was whether the learning rate might be too high, possibly causing the graph to plateau too early. To test this, we tried using a smaller learning rate$(10^-4, 10^05)$, but the resulting graph showed a very similar pattern. Thus we believe the flattening is more about the scale of the plot and the size of the initial improvement.

This also applies for the plots for Model B and C as well.

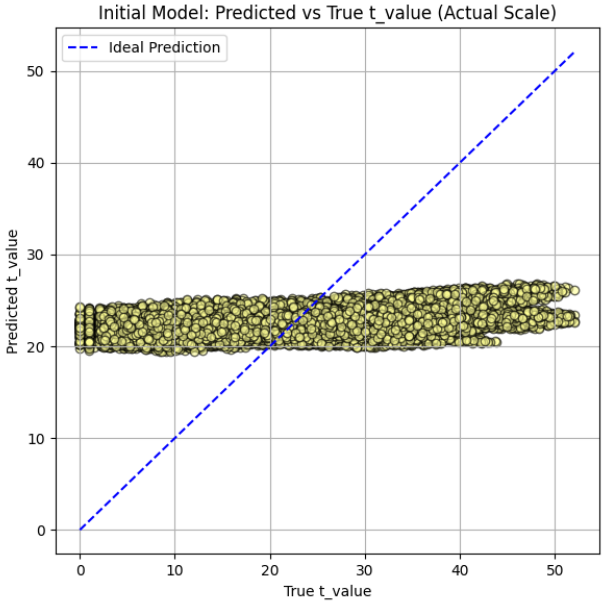

Figure 4: Model A - initial model scatter

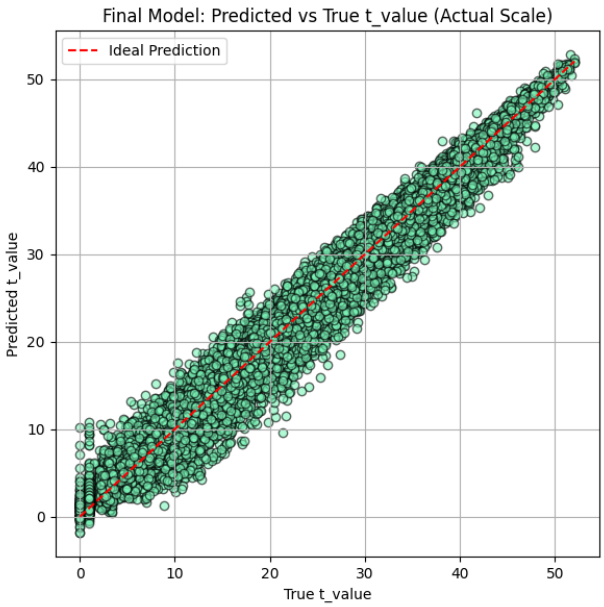

Figure 5: Model A - final model scatter

We visualize the relationship between true and predicted t-values on the test set before and after training. Fig.4 shows the scatter plot from the untrained model (random weights), while Fig.5 shows the plot from the fully trained model. The initial model produces very scattered predictions with not much implication, while the final model produces predictions closely aligned with the diagonal line compared with the initial one, which says that the model has learned a meaningful linear approximation.

4.4 Training Result - CNN - Single Ship (Model B)

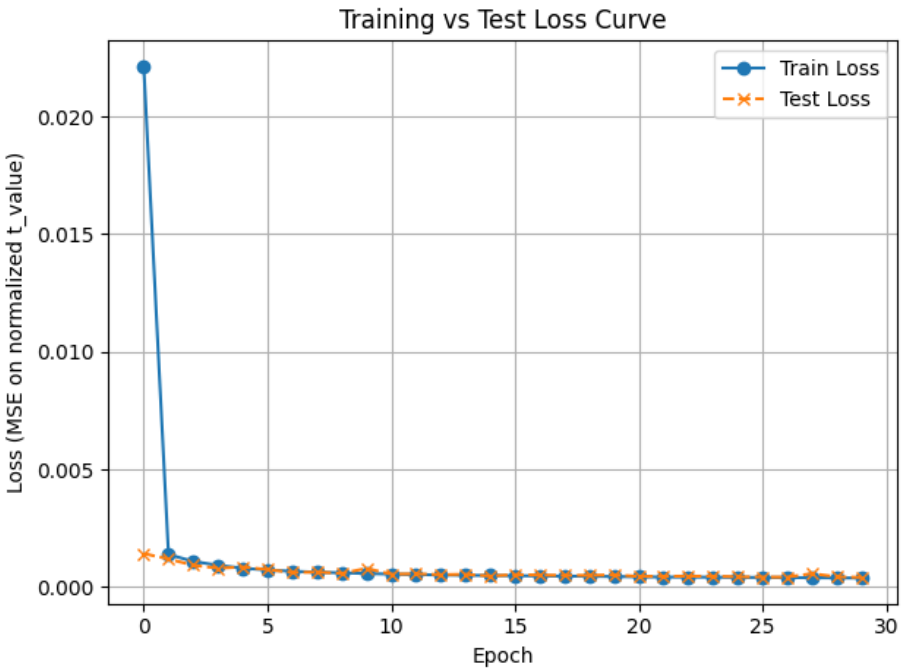

Figure 6: Model B - loss curve

Similarly for the standard NN on a single ship, we compare the training and testing loss measured as mean squared error in Fig. 6. The training and testing curve both follow a downward slope, with even less loss than the previous NN test despite far less epoch. Notably, this training and testing loss curve both follow the same trend, plateauing near 0.0004 together, indicating the model is not overfitting. These results show better performance than the previous NN in less epoch, implying that taking into account spatial features of the ship can help our model better predict the t-value.

Checking the model's predictive performance on our test data, we compute the Root Mean Squared Error de-normalized, which will represent the average error in number of steps to catch the rat. We get a final RMSE of 0.5387 steps. Compared to the standard NN-version with bot and rat positions as inputs, this is a much better average prediction error, indicating that spatial features like the layout the ship are very important to computing the expected number of moves in a bot-rat configuration.

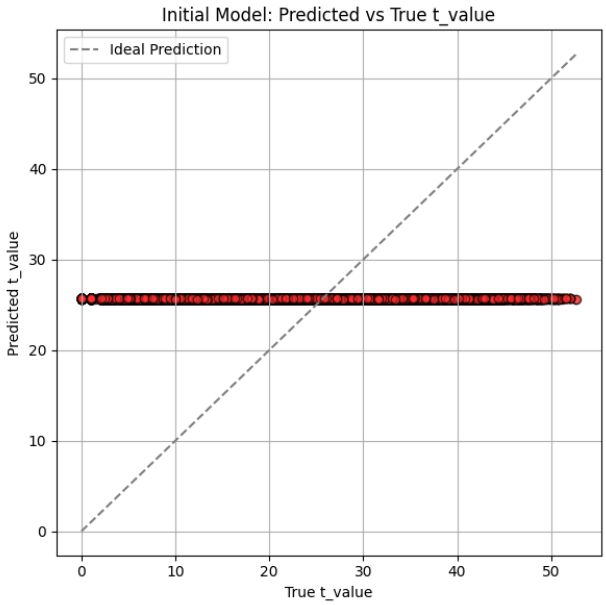

Figure 7: Model B - initial scatter

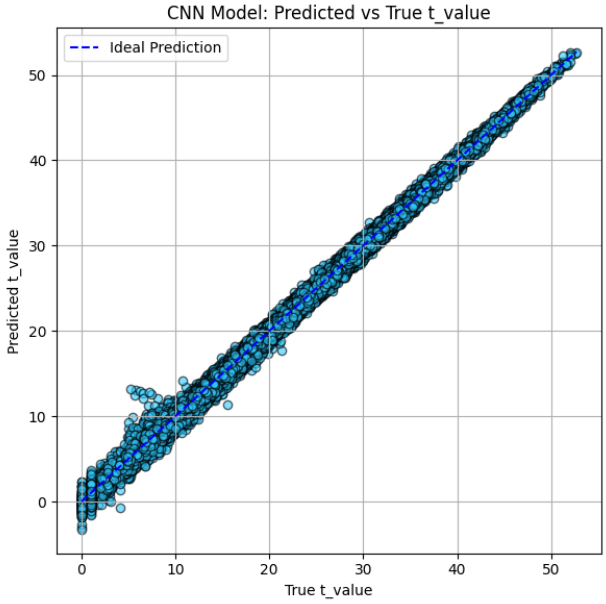

Figure 8: Model B - final model scatter

The above figures visualize the relationship between the true and predict t-values on the test set before and after training. In Fig. 7, the untrained model is predicting roughly the same t-value for each configuration. This is likely because having not learned any spatial features yet, the convolutional layers are initialized with random weights at first. Therefore, the model defaults to predicting values close to the mean of the training targets. However, after training, we can see a clear linear relationship in the predictions versus true t-value in Fig. 8, indicating the model has learned some meaningful relationships to approximate t.

4.5 Training Result - CNN - Multiple Ships (Model C)

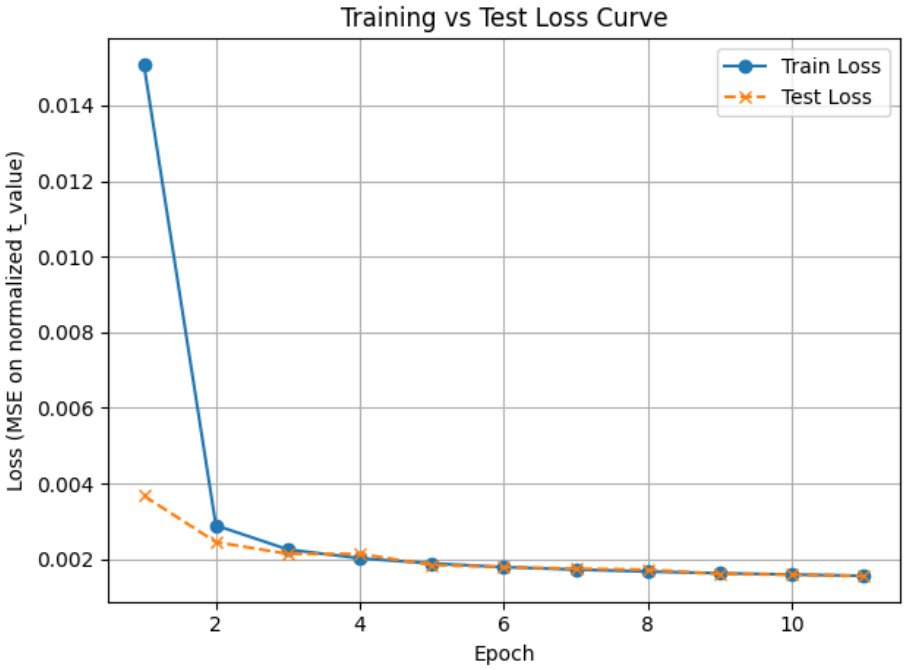

Figure 9: Model C - loss curve

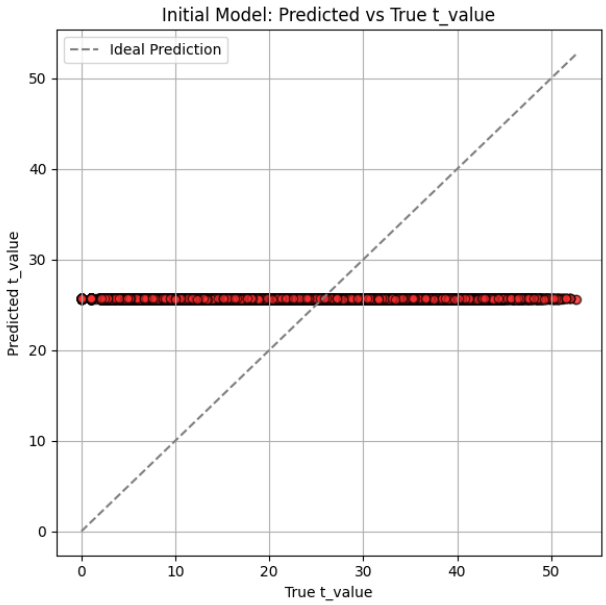

Figure 10: Model C - initial scatter

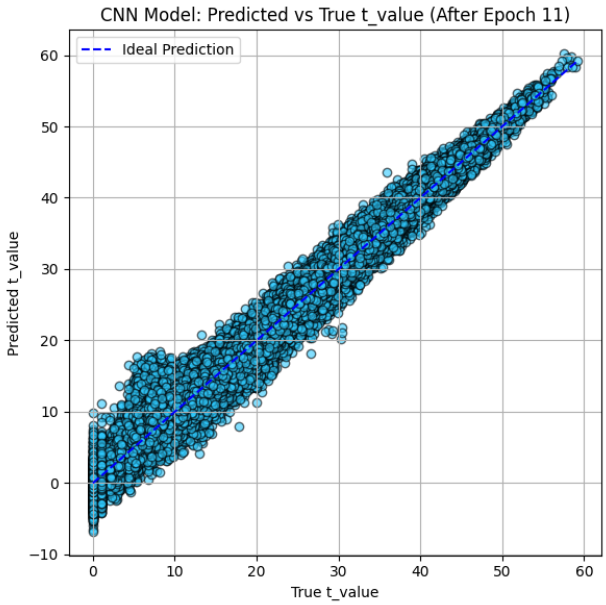

Figure 11: Model C - final scatter

Observing from the plots, we can see that they have almost identical patterns with the results from Models A and B, thus we can conclude that Model C is indeed learning from the data points and has the capability of making predictions on unseen data with relatively high accuracy. It is not simply memorizing the data points themselves.

We did notice that Model C converges more slowly and slightly outperforms Model A and B in accuracy. I think it could be caused by several reasons:

Firstly, Model C was trained on only 10 epoch, while Model B was trained on 30. That's a huge difference in data volume, which can explain the slower convergence and the need for more training to reach full potential.

In intuition, we're now considering a much larger context. It deals with 10 different ship layouts and configurations rather than just one. The model must now generalize across a diverse range of spatial structures and patterns, rather than fitting to a single fixed environment. In terms of algorithm and computation, the input for Model C has a more complex structure, identifying consistent patterns, which in turn requires more training data and time for the model to stabilize and generalize effectively. The increased dimension in input also leads to a higher number of parameters being updated and more intricate gradient computations across the network.

One thing we noticed is that training Model C took the most time to run while model A took the least. Model C is both time and computation-intensive, but it also has the advantage of being more general. That's a trade-off we're seeing here. If we were to consider in real-world scenarios, when working on machine learning problems, we will face choices about how much information we want to include, how to represent our input, how complex we want the model to be, how general or accurate we need, and how much time and memory we can afford to spend.

4.6 Comparison of Three Models

The convergence speed of the first two models differs significantly. Model A starts with a test loss of 0.0070 and requires more than 60 epochs to consistently reduce it to around 0.0028. However the Model B begins with a test loss of 0.0014 and quickly improves to below 0.0010 within the first three epochs. By epoch 10, Model B already achieves test losses lower than Model A ever does, which is a much faster convergence.

After 100 epochs, Model A achieves a best test loss of approximately 0.0028. But the Model B reaches a test loss as low as 0.00041 after just 30 epochs, showing significant improvement. We can also observe the training progress by looking at the scattered plots. Looking at the plot for the Model A (Fig.5), data points are concentrated along the diagonal line, but has a relatively large width; while for Model B (Fig.8), it gives a much tighter and narrower band of points closely hugging the diagonal line, meaning it can generate more consistent and precise predictions.

Interestingly, for all three models, predictions for smaller true t-values show greater variability, as seen in the scatter plots where points for low t are more dispersed around the diagonal. This indicates that short expected capture times (close bot-rat configurations) are harder to predict accurately. One possible reason is that t in these configurations is highly sensitive to obstacles: even a single blocked cell can change the expected number of moves from 2 to 4, a 100% increase. In contrast, when the bot and rat are farther apart, similar changes from obstacles have a smaller relative effect on large t, likely resulting in smoother patterns that the model can approximate more easily. Models B and C, which incorporate spatial features using convolutional layers, were trained for fewer epochs; additional training may allow these models to better capture the local structure of the layout, reducing variability for small t-values.

Even though such a high precision for our bot catching rat problem is not necessary, this makes us realize how good a convolutional neural network can be when dealing with image-like data (like maps) by building models and playing with parameters ourselves.

Possible reasons of the difference: The CNN was trained on full 2-D inputs, so it can learn from the spatial structure of the ship (open and blocked cells) and the relative positions of the bot and rat. In contrast, the first model receives only numerical input. Moreover, convolutional layers are able to capture local features, thus more suitable for image-like problems.

5 Partner Contributions

We discussed all ideas together before implementation for both parts and both contributed to merging and bug fixing on our code.

Rui did:

- NN and CNN Model Implementation
- section 2.3 - Training Model on Utility Function T
- section 2.4 - Training Model on Utility Function T - CNN
- section 3.2 - Setup for Training Model - NN
- section 3.3 - Setup for Training Model - CNN
- section 4.3 - Training Result - Single Ship (Model A)
- section 4.4 - Training Result - CNN - Single Ship (Model B)

- section 4.5 - Training Result - CNN - Multiple Ships (Model C)

Eric did

- MDP Implementation
- section 2.2 - Markov Decision Process
- section 3.1 - Setup for MDP
- section 4.1 - MDP Results
- section 4.2 - T Value Insights
- some analysis paragraphs for NN/CNN Models in section 4

The rest of the sections were written jointly.

6 Running A Massive CNN

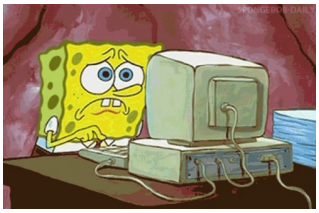

Figure 12: Two hours before ddl but the cnn is still running

Figure 13: on last epoch...

7 Reference

https://docs.pytorch.org/docs/stable/nn.html
https://docs.pytorch.org/docs/stable/generated/torch.nn.Conv2d.html