Project 3 Writeup

Krithish Ayyappan and Lakshya Gour

December 2024

1 Introduction

We have created a neural work to predict the number of steps remaining based of taking 2 probability matrices, one for where the bot can be (represented as 1s and 0s), and the other for where the rat can be (floats from our Bayesian Network). At any given point during the simulation when the bot performs an action, our neural network predicts the remaining timesteps to catch the rat.

2 Representing Data

Our data is derived from Project 2 and is stored in a CSV file with the following columns:

- **bot_type**: The type of bot used in the simulation (e.g., `baseline`).
- **timesteps_to_find_self**: The number of timesteps required for the bot to determine its location.
- **timesteps_to_find_rat**: The number of timesteps it takes the bot to find the rat.
- **bot_start**: The starting position of the bot on the grid.
- **rat_start**: The starting position of the rat on the grid.
- **move_count**: The number of move actions performed by the bot.
- **sense_count**: The number of sense actions performed by the bot.
- **ping_count**: The number of ping actions used to detect the rat.
- **failed_move_count**: The number of failed move attempts made by the bot.
- **alpha**: The openness parameter of the grid, controlling the density of accessible cells.

For the neural network input, we have extended this dataset by adding an additional column: **probability_matrices**. This column contains two 30x30 matrices that represent the possible locations of the bot and the rat at each timestep. These matrices are described below:

- **Bot Probability Matrix**: A 30x30 matrix with binary values (1s and 0s). A cell value of 1 means the bot could potentially be in that position, while a 0 means it cannot be there. This matrix is updated dynamically as the bot refines its candidate list based on its sensing actions (as described in Project 2).
- **Rat Probability Matrix**: A 30x30 matrix of floating-point values representing the Bayesian probability of the rat's location in each cell. The probabilities are computed based on pings and are updated over time, reflecting the bot's increasing knowledge of the rat's location using its Bayesian network.

As the simulation progresses:

- The **Bot Probability Matrix** becomes more sparse as the bot eliminates unlikely locations based on its observations.
- The **Rat Probability Matrix** updates in response to pings, improving the bot's understanding of the rat's possible locations.

The input to the neural network is a 2x30x30 matrix:

- The first matrix represents the bot's probability (binary matrix).
- The second matrix represents the rat's probability (floating-point matrix).

This structure enables the neural network to make predictions or decisions based on both the bot's and rat's probability distributions, as well as the bot's performance metrics over time.

3 Data Collection

The data for this project was derived from simulations based on our work in Project 2. We adapted the Simulation Class to ensure consistency across all n simulations by maintaining the same grid layout. The only variation introduced was the random placement of the bot and the rat within each simulation.

A new column, `probability_matrices`, was added to store the total probability matrices for each simulation. The length of this column was equivalent to the value in the `timesteps_to_find_rat` column. Each probability matrix was a $2 \times 30 \times 30$ representation, where:

- The first matrix indicated the bot's location, and
- The second matrix represented the probabilities of the rat's location.

To create these matrices, we utilized the list of candidate positions for the bot from Project 2. This list was converted into a 30×30 matrix of 1s and 0s based on the grid size. Similarly, for the rat's probabilities, we transformed the dictionary of coordinates and their associated probabilities into a 30×30 matrix. This matrix held the respective probabilities of the rat's location at each timestep.

During the simulation, whenever an action occurred, the probability matrices were updated and appended to the class, allowing us to log and store them. At the end of each simulation, the dataset contained the complete sequence of probability matrices for every timestep, along with the total number of timesteps recorded in the `timesteps_to_find_rat` column. This approach ensured a comprehensive representation of both the bot's and rat's movements throughout each simulation.

4 Model Architecture

For our model architecture, we are using a lightweight convolutional regression network designed to take a 2x30x30 input and predict the remaining time steps of the bot.

The decision to use a convolutional neural network (CNN) was driven by the input dimensionality. The input consists of two 30x30 matrices, almost like the CNN model with images we went over in class.

We chose to simplify the problem by converting the output of the convolutional layers into a linear model. This id done by flattening the feature maps into a one-dimensional vector, which is then passed through fully connected layers for regression. The final output predicts the number of time steps that remain for the bot.

The overall architecture is efficient for handling spatially structured input and simplifies the regression task by leveraging the CNN's ability to extract relevant features, while reducing the complexity of the problem through the use of a linear output layer.

Summary of Architectural Choices

- **2 Convolutional Layers:** These layers are chosen based on the input size (2x30x30) and the need to extract characteristics. The choice of 2 convolutional layers was chose for easiness between the 2 matrices and computational workload.
- **Batch Normalization:** Helps stabilize the training process and accelerates convergence by normalizing the activations, reducing internal covariate shift. This helped our model convolutionalize better and help with training our data easier.
- **Dropout:** Prevents overfitting, especially given the small model size. Dropout randomly sets 50% of the neurons to zero during training, promoting robustness and preventing reliance on specific features. We included dropout as we had a larger dataset and ran into a lot of overfitting problems when first starting off, adding this really helped.
- **ReLU Activation:** Provides non-linearity and helps with faster convergence by mitigating the vanishing gradient problem. Simplistic and easy to use based on lecture.

- **MSE and MAE Loss:** Suitable for regression tasks, penalizing large errors and encouraging the model to predict values closer to the true values. Utilizing both errors helped up compare models faster and test easier side by side.
- **Mini-batch SGD:** Efficient training, particularly for large datasets, allowing parallel processing and more stable convergence compared to pure stochastic gradient descent.
- **Residual Connections**: Connects the output of one earlier convolutional layer to the input of another future convolutional layer several layers later (e.g. a numer of intermediate convolutional steps are skipped). Truth be told, I do not understand this fully but we did modify our forward function to include this aspect as specified by our TA.

Each architectural decision was made with the goal of balancing model complexity, computational efficiency, and generalization. The design reflects a good starting point for a lightweight CNN that should perform well on regression tasks, with room for further tuning based on the dataset. The reason why we finalized this architecture was that with our large dataset (size of 27,000), we easily ran into problems with overfitting if we made the model too complex in terms of parameters. Thus, we settled on a lighter model with low-level parameters and just 2 layers to simplify our approach and gain better results to improve test loss to generalize. Taken it even further, we testing our model and realized because we had so much data, it was better to add more than 1 dropout layer. So our final architecture:

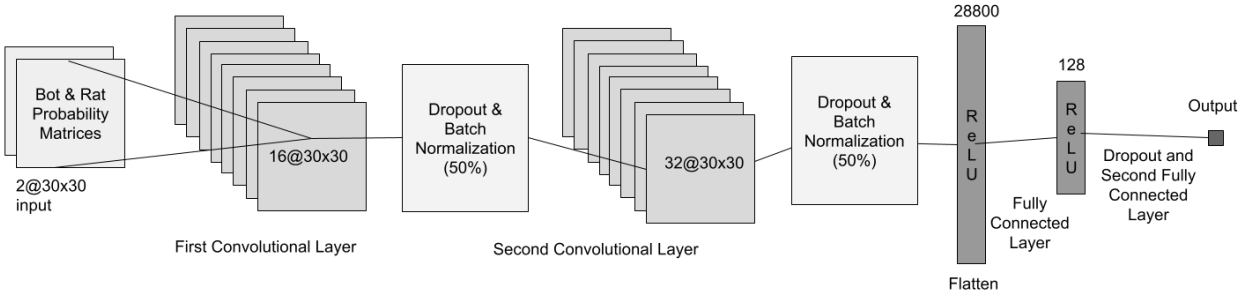

Figure 1: Neural Network Architecture of the Lightweight Convolutional Regression Model.

5 Loss and Results

We utilize MSE and MAE to represent our loss function for this problem. This is because we are predicting timesteps remaining which is a regression problem, not a classification problem. MAE is the Mean Absolute Error, representing how far the predicted value is from the actual value during training. MSE is the Mean Squared Error, representing how far the average squared difference between the predicted and the actual target values within a dataset.

MSE: Penalizes larger errors more significantly, making it useful for highlighting large deviations between predicted and actual values.
MAE: Provides a more interpretable measure of average error, as it is in the same units as the target variable.

Upon analyzing the results, we realized that our MSE and MAE values were quite good when compared to the variance of our target data. The targets had a mean of 213.38 and a standard deviation of 98.60, indicating that our model's predictions were relatively accurate.

Analysis of Train and Test Losses

Our model's performance can be evaluated by comparing the train and test losses to the variance and standard deviation of the target data. Here are the key metrics:
Train Loss (MSE): 1725.13
Train Loss (MAE): 28.74
Test Loss (MSE): 974.85
Test Loss (MAE): 17.71
The target data had a mean of 213.38 and a standard deviation of 98.60.

Mean Squared Error (MSE)

- MSE measures the average squared difference between the predicted and actual values. It penalizes larger errors more significantly.
- Our train MSE of 1725.13 and test MSE of 974.85 are relatively low compared to the variance of the target data. The variance (standard deviation squared) is approximately 9720. This indicates that our model's predictions are much closer to the actual values than the variance of the target data.

Mean Absolute Error (MAE)

- MAE measures the average absolute difference between the predicted and actual values. It is in the same units as the target variable, making it more interpretable.
- Our train MAE of 28.74 and test MAE of 17.71 are significantly lower than the standard deviation of 98.60. This shows that, on average, our model's predictions are within a small range of the actual values, indicating high accuracy.

The model with three dropout layers (final_model_2.pth) performed better than the one with one dropout layer (final_model_1.pth). In our case, the model with three dropout layers generalized better to the testing data, as evidenced by higher accuracy metrics (shown in the figures: 2, 3, 4, 5). At each threshold point, the model with three dropout layers was more accurate for the data that it had never seen before (figures 3 vs 5). This indicates that the additional dropout layers helped the model avoid overfitting, leading to improved performance on unseen data.

Training and Testing Loss Graphs Below are the graphs showing the training and testing loss over epochs, validating that our model is learning something meaningful.

Threshold Accuracy Results:

THRESHOLD	Step Accuracy (%)
0	0.000000
10	61.035614
25	87.464855
50	94.704780
100	98.922212

Percentage Threshold Accuracy Results:

PERCENTAGE_THRESHOLD	Percentage Accuracy (%)
0.10	58.762887
0.25	85.051546
0.50	93.345829
0.75	95.665417
0.90	96.344892
0.95	96.462043

Figure 2: Final Model 1 (1 Dropout Layer) Performance on Training Data

```
Threshold Accuracy Results:
 THRESHOLD  Step Accuracy (%)
         0           0.000000
        10          14.804965
        25          33.289007
        50          56.892730
       100          80.629433

Percentage Threshold Accuracy Results:
 PERCENTAGE_THRESHOLD  Percentage Accuracy (%)
                 0.10                13.253546
                 0.25                30.429965
                 0.50                58.865248
                 0.75                75.842199
                 0.90                81.449468
                 0.95                82.491135
```

Figure 3: Final Model 1 (1 Dropout Layer) Performance on Testing Data (hasn't seen before)

```
Threshold Accuracy Results:
 THRESHOLD  Step Accuracy (%)
         0           0.000000
        10          51.663543
        25          81.490159
        50          92.057170
       100          97.656982

Percentage Threshold Accuracy Results:
 PERCENTAGE_THRESHOLD  Percentage Accuracy (%)
                 0.10                47.774133
                 0.25                79.381443
                 0.50                90.885661
                 0.75                94.119025
                 0.90                95.149953
                 0.95                95.571696
```

Figure 4: Final Model 2 (3 Dropout Layers) Performance on Training Data

```
Threshold Accuracy Results:
 THRESHOLD  Step Accuracy (%)
         0           0.000000
        10          15.824468
        25          36.945922
        50          60.571809
       100          83.311170

Percentage Threshold Accuracy Results:
 PERCENTAGE_THRESHOLD  Percentage Accuracy (%)
                 0.10                14.516844
                 0.25                35.638298
                 0.50                62.854610
                 0.75                77.836879
                 0.90                83.023050
                 0.95                84.064716
```

Figure 5: Final Model 2 (3 Dropout Layers) Performance on Testing Data (hasn't seen before)

When we look at the statistics table for the Threshold Accuracy Results, we notice that Model 1 tends to have higher accuracy on its training data. The `THRESHOLD` represents how far the predicted accuracy is from the actual in timesteps. The `PERCENTAGE_THRESHOLD` does the same but in quartiles based on the standard deviation.

Initially, we thought this was a good sign, as high training accuracy typically implies high testing accuracy. However, after running a separate testing set to see the results, we quickly observed that the model was overfitting, as the performance on the testing data was poor.

To address this, we added two more dropout layers and finalized our model. Although this new model scored slightly lower on training accuracy, it was able to achieve better results on the testing set. This indicates that the model generalized well and was not overtrained on the training data.

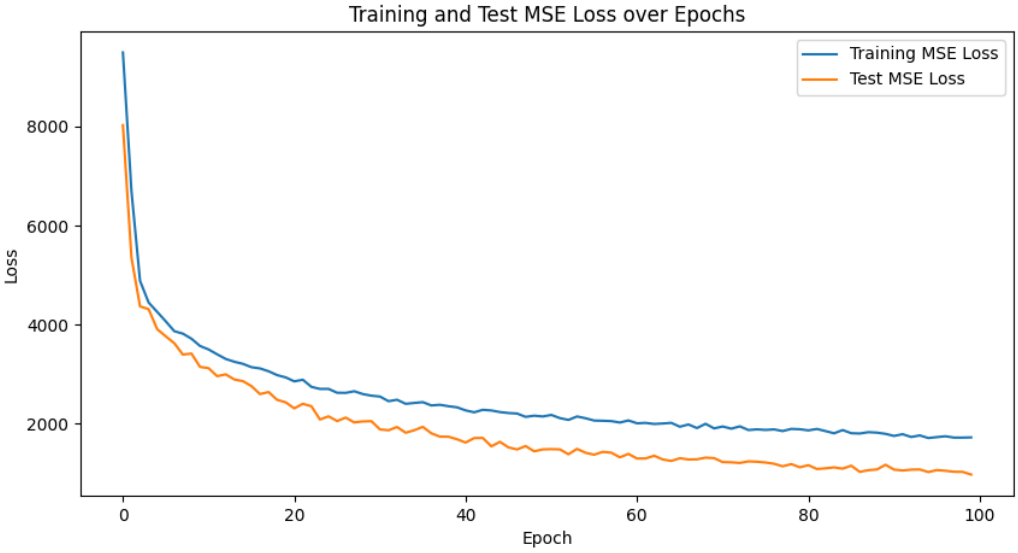

Figure 6: MSE Loss for Training Final Model 2

Figure 7: MAE Loss for Training Final Model 2

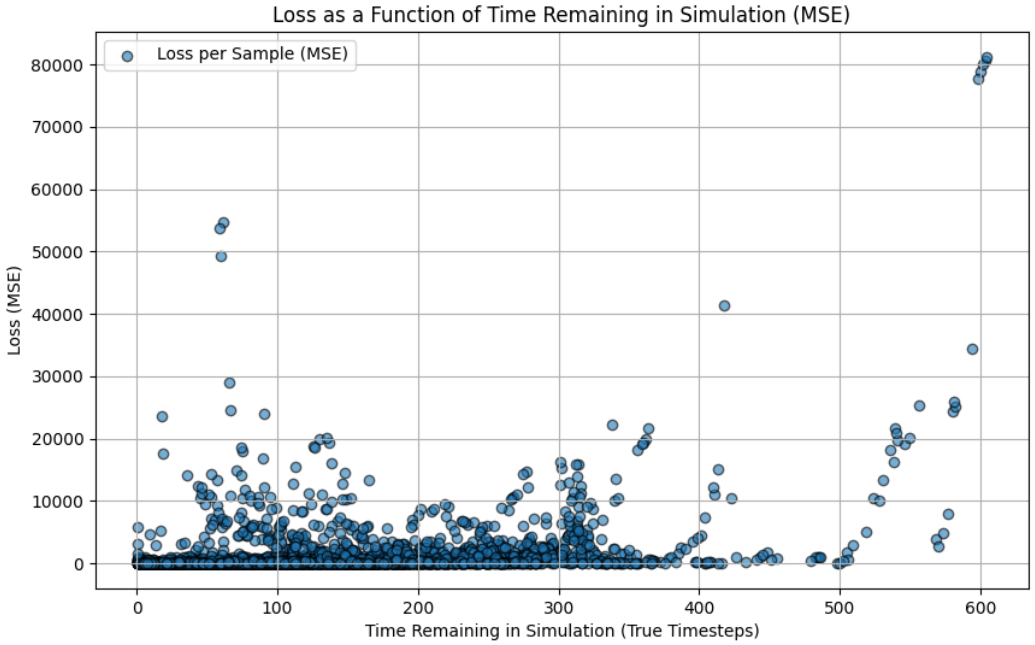

Figure 8: Loss Function Comparison Over Time

6 Model Analysis

As shown in the training and test loss graphs (Figure 6 for MSE, Figure 7 for MAE), both start high but decrease as the number of epochs increases. The MSE for both training and testing gradually decreases with a similar trend, indicating that the model is being trained effectively without overfitting, thanks to the large dataset and well-tuned parameters. The test MAE starts slightly higher but also decreases over time. The lower values of MAE in the graph are due to the scaling; while the MAE typically ranged in the 20s, the graph's loss values are in the thousands.

This graph (Figure 8) represents the loss per sample in terms of the true remaining time steps (i.e., the actual timesteps) versus the predicted loss. As shown, most of our training samples are very close to 0, indicating that our model accurately predicted the remaining timesteps for the majority of the data. However, there are some outliers at around 600 timesteps. This is because the mean timesteps for the training data was 213.38, and the model struggles to accurately predict timesteps outside of this range.