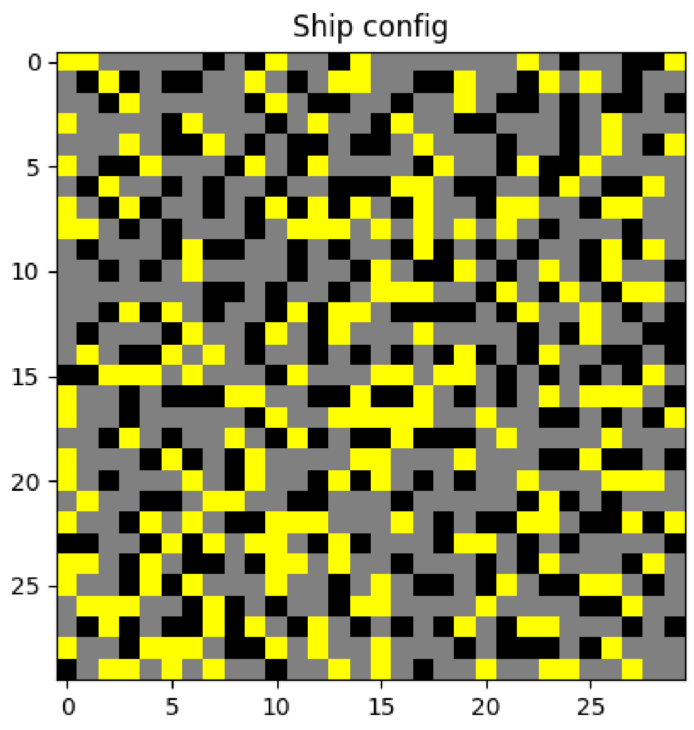


Localization, Utility, ML, and Me

PROJECT 3 WRITEUP

Abhishu Oza | Intro to AI | 16 May 2025

0) The Code

- **localization_functions.py** contains functions for the ship, updating L, input encoder (to feed to ml model for pi 1) as well as pi_0, pi_1 and pi_2 strategies
- **pi_0_work.py/pi_1_work.py** runs required number of tests of the strategy and optionally stores the set L with no. of moves needed to localize
- **actual_plots.py/prediction_plots.py** creates the |L| vs actual/predicted AverageNumberOfMovesNeeded graph
- **models.py** – defines model architecture for both pi_0 and pi_1 move count predictors
- **split_train_save_model.py** splits into train/test, trains and saves the model (yes, really)
- **compare_pi_0_pi_1_pi_2.py** compares average number of moves needed for all 3 strategies on the same ships
- **ship_visualiser.py** You can paste in any set L and it will visualize what the bot sees on the map (this helped me realize why localization is taking so many steps, it's a harder problem than I initially estimated)

1) Representing The Problem

No of possible sets L - On a given ship, for each open cell, it can either be part of L or not a part of L. If the set of locations of all open cells is U, then the possible sets L are all the subsets of U, so 2^|U| possible sets.

Following is the function L_next, which takes in L, a direction, and outputs L_new that the set of locations the bot might be in after attempting to move in that direction –

```
def L_next(grid, L, direction):
    L_new = set()
    for loc in L:
        next_loc = loc + direction
        if next_loc is not an open cell:     L_new.add(loc)
        else:                                L_new.add(next_loc)
    return L_new
```

This means that |L_t+1| < |L_t| when there are cells that get merged, because some specific cells moved into the location of some of the cells which stayed in place.

L for which C*(Map,L) is obvious – I could not think of other types of L than simply sets L for which |L| = 1, i.e there is only one element in the set. There are 0 steps needed to localize L here. Because even for 2 cells remaining, their locations end up determining how many steps are needed, and C* must be using a policy PI* clever enough that it varies significantly depending upon L.

Recursive Formulation for C∗ - For actions that have deterministic transition, Bellman Equation takes the minimum of {cost(action) + C*(state)} for all possible state action pairs that leads to the current state. Cost of action is the same – 1 – so it can be taken out. Meaning –

$$C*(L) = 1+\min(C*(L\_up), C*(L\_down), C*(L\_left), C*(L\_right))$$

This is not a finite horizon problem because we can technically make a set of movements for which after one point the size of our set L never decreases (for example only moving left, right, left, right... and so on).

However, it's not an infinite horizon problem necessarily because the ship is bounded by a 30 by 30 grid, which implies that there is an upper bound on the number of actions needed to reduce the set of possible locations to 1. Hence for reasonable policies, this will not be an infinite horizon problem.

For our maps, we have opened all dead ends, meaning no 2 locations have the exact same neighborhood (because the ship is finite). So, a finite number of actions must be able to break the symmetry between any 2 locations. This means it is impossible for 2 locations to remain in L. So, we can ensure that we are able to eliminate all but 1 location using a finite number of steps.

In fact, I encountered this when I created pi_0 policy – for some cases, when there were only 2 locations left in L, the policy was unable to break the neighborhood symmetry. So, I had to assign it a new target if the same set L looped more than 5 times. This verifies that although some policies may lead to an infinite horizon, the best one will not.

If the function C∗ were easy to calculate, we calculate C*(Map, L_next(L, direction)) for all directions and choose the direction with lowest cost (using argmin(direction) for C*).

1.1) Connection with graph search

Our search space is basically a tree where the edges are actions linking a state to its next state. (This is possible because there is only 1 state with probability 1 and rest all being 0, so an action directly leads to the next state instead of a probability distribution, along with gamma = 1). Hence, we can apply graph search here to find goal nodes which correspond to $|L| = 1$.

$|L|$ can be a good heuristic for A* here. It's not an admissible heuristic, but we can roughly say that if there are more locations left then we will need to take more actions to reduce them.

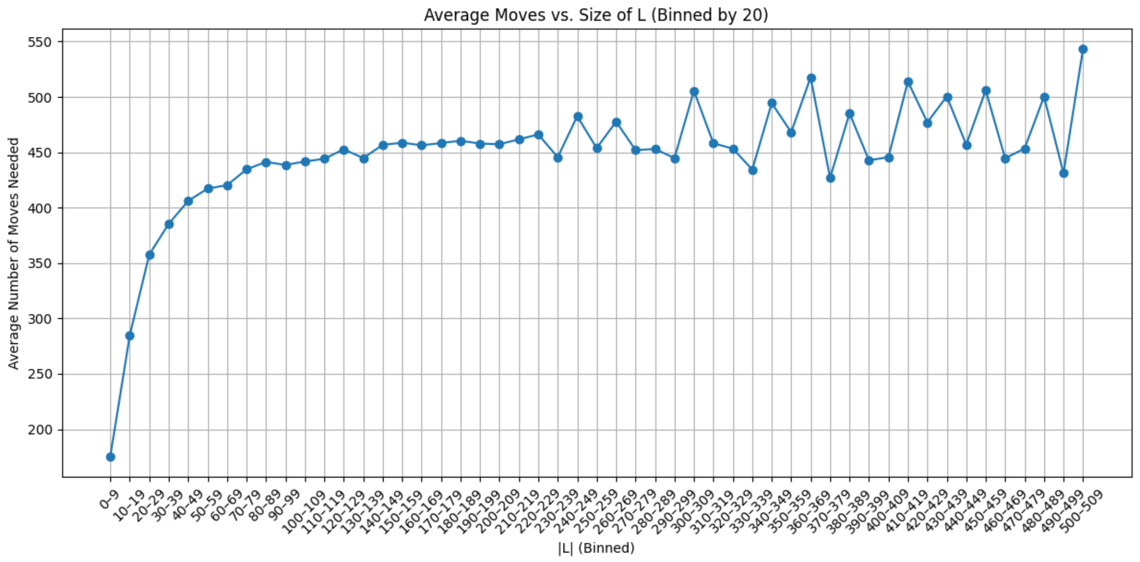
From a start state, A* uses an estimate of the number of actions to the nearest goal state and explores the best path relative to the heuristic. C* is the actual least number of actions from any start node to the nearest goal node. So, if we can find an admissible heuristic for our problem, A* will find C* by finding the optimal path to any goal state.

A* is essentially a special case solver for C* when action outcomes are non-probabilistic and at least an admissible heuristic is known. But it finds C* only for the states along the optimal path, not the whole state space.

2) Strategy PI_0

For PI_0, I implemented the strategy as described in the pdf with a minor tweak. Because I observed that after narrowing L to 2 cells, for some test runs, it would go into an infinite loop as described before. So, I added code to check how many times the current L appears in the past 5 states encountered. If it is 5 out of 5, it decides that it is stuck in a loop and chooses a new target state. PI_0 is tested on random L0 generation

Following is the plot for |L| vs AverageNumberOfMovesNeeded (|L| is binned into sizes of 10) –



This takes the entire localisation dataset (meaning all the intermediate L along with L_0), which is why there are a much, much more datapoints for lower sizes (thus accounting for higher variability as |L| increases).

3) Machine Learning

Following is the description of my model and training pipeline –

1) Input Space – It is a 2 by 30 by 30 matrix. The ship map is the 1st matrix with BLOCKED = 0 and rest are 1. The 2nd matrix has 1 for cells that are in L and 0 for cells not in L.

2) Output Space – This is a regression problem, with output as a single real number – prediction of number of moves needed for localisation.

3) Model Space – My model is a CNN with the following configuration –

```
self.conv = nn.Sequential(
        nn.Conv2d(in_channels=2, out_channels=16, kernel_size=3, padding=1), nn.ReLU(),
        nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3, padding=1), nn.ReLU(),
        nn.MaxPool2d(2),
        nn.Conv2d(in_channels = 32, 64, kernel_size=3, padding=1), nn.ReLU(),
        nn.MaxPool2d(2)
    )
reduced_size = input_size // 4
self.fc = nn.Sequential(
        nn.Flatten(),
        nn.Linear(64 * reduced_size * reduced_size, 128), nn.ReLU(),
        nn.Linear(128, 1)
    )
```

The convolutional model helps extract important localised information for cells – because the walls around cells dictate how amenable is it to merging with other cells via moves. The fully connected model uses extracted spatial features for move count prediction.

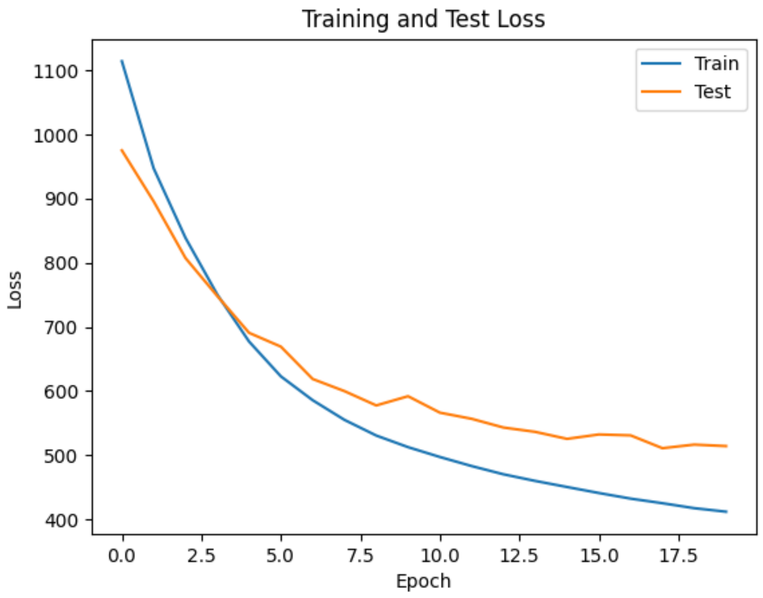
4) Loss Function – I am using squared error loss – $L = 1/N * \sum (y i - yi)^2$

5) Training Algorithm – I am taking standard values for Neural Network training –

- Adam optimizer
- Learning rate of 0.001
- 20 epochs over batched data (batch size = 32)

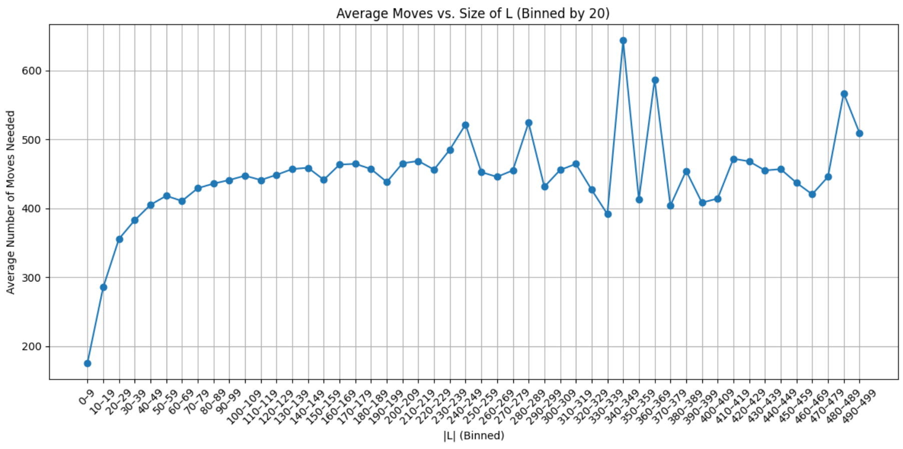
Data Generation and split – For PI_0, I ran 1000 test simulations and collected L_i, |L| and t_n – t_i at each timestep 'i' in each of the tests, giving me 446114 datapoints. For PI_1, I ran 50000 test simulations and only collected L_0, |L|, and t_n for each test run. For both, I used a train/test split of 80/20 percent.

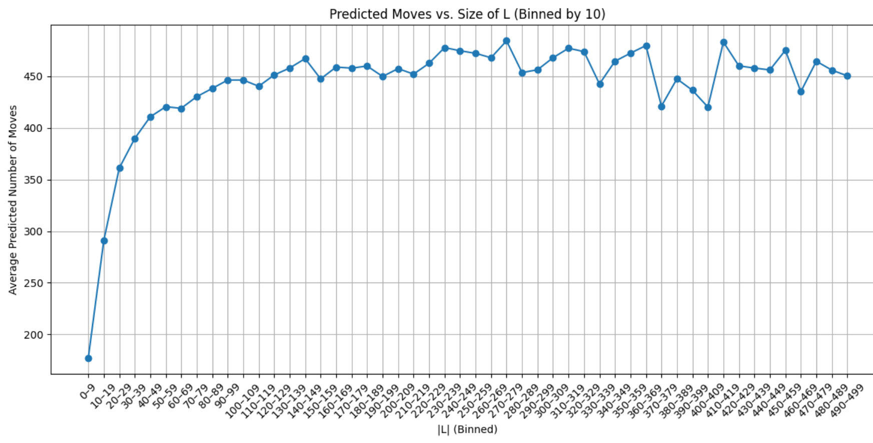
Following is the plot for Train and test loss per datapoint vs epochs. We can see a steady decrease of train and test losses -



Although the loss value is quite high, my guess is that the model was sufficiently able to distinguish between moves that took more steps vs those that took less, because the upcoming section will show results for PI_1 strategy.

Following are the graphs for average actual move value and average predicted move value vs |L| for the test dataset (test_pi_0).





So there was higher variance in some of the larger |L| sizes which was not captured by the model, but overall the plot values look quite similar. The second graph here also matches a lot with the first AverageNumberOfMovesNeeded graph (which was created on the entire localisation_dataset data).

Currently the model was training well enough to not warrant any techniques for reducing overfitting so I did not apply any.

4) Strategy PI_1

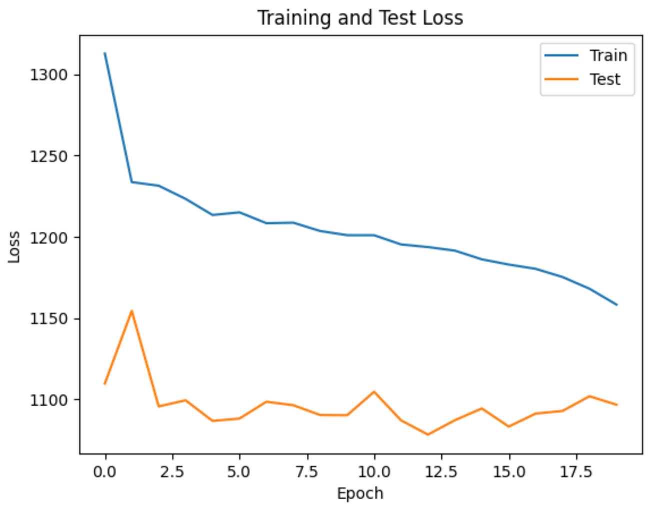
I used the above estimator for creating the new policy PI_1 as described in the pdf. In compare_pi_0_pi_1_pi_2.py, I test 1000 ships and compare the strategies that I implemented. <u>To compare them fairly, I gave as L_0 input all the unblocked cells of the ship, so that we can measure performance over the entire episode</u>. Following is the output

- Average moves for pi_0 over 1000 tests: 484.67
- Average moves for pi_1 over 1000 tests: 458.32
- Average moves for pi_2 over 1000 tests: 460.89

So PI_1 was sufficiently able to improve on PI_0. I will discuss my problems with PI_2 in the next section.

4.1) Estimator of C_1 and Strategy PI_2

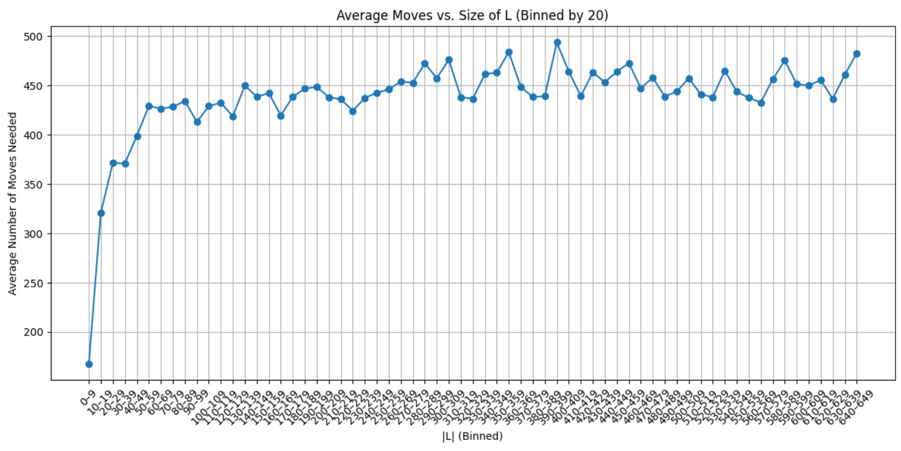
I used the above estimator for creating the new policy PI_1 as described in the pdf. As explained in the Machine Learning section, I ran 50000 test simulations and only collected L_0, |L|, and t_n for each test run. Only the initial state matters because for the rest of the steps, PI_1 is only using PI_0 as a subroutine and so data for those states does not help us evalueate PI_1. Following is the training curve –

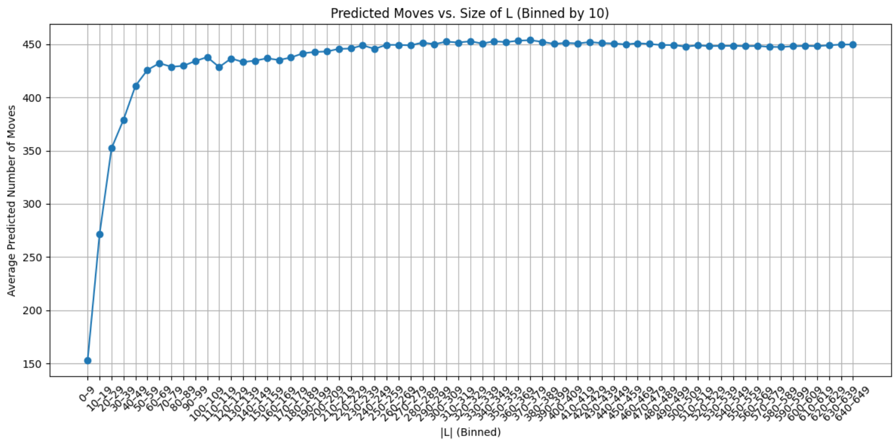


This is a horrible training curve, and is the reason why the PI_2 policy could not improve upon PI_1. I doubt that this was an overfitting or architecture issue (although the architecture and training pipeline could also be improved) due to this - I observed that one major difference between C_0 predictor and C_1 predictor is that the former had only 1000 test runs and a gradually decreasing L_0 size for all its (approx. 44000) datapoints, whereas the latter had only a single datapoint, but from 50000 runs. This may have led to a lot more variance for the latter vs the former looking at very similar kinds of sets lots of times.

However, PI_1 only uses the C_0 predictor for its first step, so I could not understand how to overcome this issue. Perhaps I made some mistake along the way but could not exactly pinpoint what it was.

Following are the graphs for average actual move value and average predicted move value vs |L| for the test dataset (test_pi_1) –





Bad results. I ended up being short of time from here on out so was unable to impove the C_1 predictor, and was also unable to work on training the optimal utility directly. This is entirely on me. I apologize for an underwhelming conclusion.