# Project 3: Localization, Utility, ML, and You

16:198:520

**The Problem:** You are again a bot on a spaceship (the spaceship generated exactly as in the previous two projects). However, a neutrino storm has knocked out some of your memory and sensors. You do not know where exactly you are in the ship. What's worse, if you try to move in a given direction, you can't even tell if you are successful, because your sensors aren't registering bumping into walls anymore.

After some internal diagnostics, you determine that your motors are fine - this means that if you try to move in a given direction, and it is not blocked, you know you will be successful. If it is blocked, you will be unsuccessful, and stay in place.

You also have a map of the ship stored in your memory, and an orientation to know what directions correspond to which of your motors. Your memory is corrupted, but you know that you are in one of a number of possible locations.

Based on this - can you determine where you are? *Spoilers: the answer will be yes*

## 1 Representing The Problem

We can take the map of the ship as given. At any time, what the bot knows is a set of locations $L$ that it might be at, within the map of the ship. At the start of the problem, the bot has some initial $L_0$.

**Question to Answer:** How many possible sets $L$ are there?

Note that given a set of locations $L$ within the map of the ship, if the bot attempts to move in a given direction, the set of locations it can be will change.

**Question to Answer:** For a set of locations $L$ where the bot knows it might be, after attempting to move *left*, what does the bot know about where it might be? *Right*? *Up or Down*? Show that in each of these cases, the set of 'next possible locations' is well defined.

It may be useful to write a function that takes as input a ship map, a set of locations $L$, and a desired direction, and returns as output $L_{\text{next}}$, the set of locations the bot might be in after attempting to move in that direction.

In particular, after attempting to move some direction, it is possible that the total number of possible locations goes down, i.e., $|L_{t+1}| < |L_t|$. Notice that if the number of possible locations goes to 1, then we know precisely where we are in the ship.

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 $L$.

**Questions to Answer:**

- Show that for a map Map, there are some sets $L$ for which $C^*(\text{Map}, L)$ is *easy/immediate/obvious* to calculate. What are these $L$?
- For any general $L$, however, give a recursive formulation for $C^*$, in the spirit of the Bellman Equations.
- This is not a finite horizon problem, but it's also not an infinite horizon problem necessarily (hence needs no discounting). Why?
- 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.
- 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.

The problem is now one of optimal planning - finding a sequence of actions to get us to a goal state (knowing where we are), ideally as short a sequence as possible.

**Questions to Answer:**

- How could you apply our graph search techniques to this problem?
- What might a good heuristic for $A^*$ be here? *Note: does not have to be consistent or admissible, it just needs to meaningfully estimate work remaining.*
- 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.

## 2 A Solution Emerges

For this section, you can generate a single map, and consider it held constant for the following experiments.

Given a map, and a set of initial locations $L$, consider the following strategy:

- Pick an open cell that is a corner or a dead end. Call this cell **target**.
- While the number of possible locations is greater than 1:
  - Pick a location $(x, y) \in L$ at random.
  - Compute the shortest path from $(x, y)$ to **target**, and the moves to execute this path.
  - Execute this sequence of moves, updating $L$ as you do.
- Return the total sequence of moves produced.

*Note: This strategy does not exit until the problem is localized. Hopefully it does so in finite time!*

I'd like you to experimentally determine how effective this strategy is, by generating a number of initial sets $L$, and running this strategy to determine how many moves it takes to solve the problem.

Note, you have basically everything you need to implement this - the ship generation, you can take from previous projects. The shortest path planning between two points comes straight from Project 1 (no fire this time!). The only additional elements you need to consider here are a) representing and managing the set $L$ of possible locations, and b) the update of $L$ for a given move, as discussed in the previous section.

**Data to Generate:**

- Create a plot with $|L|$ on the $x$-axis (running from 1 to the total number of open cells), and *AverageNumberOfMovesNeeded* on the $y$-axis.
- For each value of $|L|$ (or enough to generate a good plot), randomly generate a number of different initial $L$ of that size (hint: lots of pre-built functions exist for choosing random sets from specified sets).
- For each value of $|L|$, run this strategy on the generated $L$, and average the results together to get an estimate for AverageNumberOfMovesNeeded.

## 3 What Does This Have To Do With Machine Learning, Dr. Cowan?

From before, there are a *lot* of possible initial states $L$ we might have to deal with, too many to solve them all. Instead, what I would like to do is create a model that takes as input a given $L$, and predicts, as output, the number of moves the above strategy needs to solve the problem.

**Questions to Answer / Things to Implement:**

1) **Input Space: a set of locations** $L$ The question you have to answer is, how do you want to represent $L$ in a meaningful way?

   Note: do you want to also represent the ship map as part of the input, and if so, how?

2) **Output Space: a real value** this is pretty straightforward, we are trying to predict a real number.
3) **Model Space: a set of functions mapping an input $L$ to a real value** you obviously have some flexibility here, as the remainder of the course is going to be about different kinds of models, but you do need to settle on something. How can you construct a model that is meaningful to the problem in some way? What parameters or decisions does your model depend on?
4) **Loss Function: how to measure error** notice that because we are trying to predict a real value, squared error loss is the natural (though not only) loss function to consider.

   Note: since we know that the number of moves remaining is always going to be non-negative, does that suggest or change anything to you about how to construct the loss function or model?

5) **Training Algorithm:** Whatever your model space, you need some kind of algorithm that will produce the best (or at least, a very good) model from the model space to minimize the loss over your data set.

A note on data: Training data is easy to generate - we can simply generate a large collection of initial $L$ sets, run the strategy on each of them, and record what the final number of moves is. It is useful to think of each simulation as an **episode**, each step of the episode providing a new data point about the current set of possible locations, and the number of moves needed to solve it (episode length - current timestep).

**Questions to Answer / Analysis to Include:**

- Be clear and specific as to your answers to the defining five questions in the previous gray box.
- Be clear and specific as to how you generated your training (and testing) data, and how much training (and testing) data you used.
- Generate a plot of training loss (per data point) over training time (to verify that the model is learning, and that loss is decreasing)
- Generate a plot (on the same axes) of testing loss (per data point) over training time (to verify the model is generalizing well and not overfitting).
- Be clear about any steps you took to reduce overfitting.
- Once your model is trained, create a plot with $|L|$ on the $x$-axis, average predicted number of moves on the $y$-axis, and for each $|L|$, generate a number of sets $L$, evaluate your model on each of them, and average the results for that $|L|$ together. How does this graph compare to the one from the previous section?

## 4 Improving Your Strategy

If you are able to successfully train the model from the previous section, a problem arises: all you've really done is model the performance of the strategy I gave you. This doesn't answer the question at all of 'what is the best strategy for localizing the bot'?

*Everything in the previous sections was, I feel, relatively straightforward (generate data, build and train a model). Indulge my ambitions for a moment.*

Let's call the strategy of the previous section $\pi_0$. What you accomplished was building a model to predict or estimate the cost of executing that policy on any set of locations $L$. If the exact cost might be denoted $C_0(L)$, what you did was create an estimator, we might denote $\hat{C}_0(L)$ of $C_0(L)$, where $\hat{C}_0$ is the model you trained. (This 'hat' notation is commonly used for estimators.)

We might try to improve the policy in the following way: construct a new policy, $\pi_1$, that works as follows:

- For an initial set of locations $L$
- Compute $L_{\text{up}}, L_{\text{down}}, L_{\text{left}}, L_{\text{right}}$, the sets that would result from attempting to move up, down, left, or right, from initial state $L$.
- Compute $C_0(L_{\text{up}}), C_0(L_{\text{down}}), C_0(L_{\text{left}}), C_0(L_{\text{right}})$
- Whichever next state has the smallest total cost, take that action, and update the set of locations appropriately.
- For all future steps, follow strategy $\pi_0$ on the current state.

In this way, we would expect that $\pi_1$ should perform a little better, on average, than $\pi_0$ - because $\pi_1$ is free to take actions in that first step that $\pi_0$ is not, it can potentially take better initial actions than what $\pi_0$ can do. This shouldn't necessarily be a major improvement - but if we can get any improvement, we can start to iterate on it.

*Note: $C_0$ is hard to calculate. So what you would end up doing is evaluating $\hat{C}_0$ instead, and using these values to choose the next action. If it is a good estimator, then it should be close enough for the previous argument to apply.*

**Questions to Answer / Things to Implement:** Is this policy $\pi_1$, using the estimator $\hat{C}_0$, effective (on average)? Does it perform better or worse than policy $\pi_0$? How can you compare them?

Note, if it doesn't perform at least as well, on average, it potentially suggests that your estimator $\hat{C}_0$ is not as good as it should be, and could be improved.

If however policy $\pi_1$ *is* as good or better - this suggests the natural next step: construct a new policy, $\pi_2$, that works as follows:

- For an initial set of locations $L$
- Compute $L_{\text{up}}, L_{\text{down}}, L_{\text{left}}, L_{\text{right}}$, the sets that would result from attempting to move up, down, left, or right, from initial state $L$.
- Compute $C_1(L_{\text{up}}), C_1(L_{\text{down}}), C_1(L_{\text{left}}), C_1(L_{\text{right}})$
- Whichever next state has the smallest total cost, take that action, and update the set of locations appropriately.
- For all future steps, follow strategy $\pi_1$ on the current state.

However, the expected cost $C_1$ of policy $\pi_1$ is hard to calculate, so implementing the above is difficult. If we had an *estimator* for $C_1$ though...

**Questions to Answer / Things to Implement:**

- Build a data set on the number of moves needed to solve initial location sets, under policy $\pi_1$. Be clear as to your methodology, especially how you are getting the desired output / number of moves remaining for a given state. *Hint: Why does only the initial state really matter?*
- Build and train a model $\hat{C}_1$ to predict the performance of policy $\pi_1$ on a set of locations $L$.
- Show the training curves, demonstrate learning and lack of overfitting.
- How does a policy $\pi_2$, implemented using $\hat{C}_1$ perform? If you aren't able to get something that performs better than $\pi_1$, why do you think that is?

We could potentially try to iterate this process, pushing our use of policy $\pi_0$ further and further into the future as we learn what better steps to take now.

Alternately, we could potentially try to compute the optimal utility of a policy directly, and skip this kind of iterative approach. We know that for terminal localized $L$, we should have $C^*(L) =$ (your answer from the first bullet point in the gray box on Page 2). For non localized $L$, we should have

$$C^*(L) = \text{YourFormulaFromSecondBulletPointFirstGrayBoxPage2}(C^*(L_{\text{up}}), C^*(L_{\text{down}}), C^*(L_{\text{left}}), C^*(L_{\text{right}})) \quad (1)$$

If we want to train a model $\hat{C}^*$ to estimate $C^*$, this suggests a loss function based on

$$\left[\hat{C}^*(L) - \text{YourFormulaFromSecondBulletPointFirstGrayBoxPage2}(\hat{C}^*(L_{\text{up}}), \hat{C}^*(L_{\text{down}}), \hat{C}^*(L_{\text{left}}), \hat{C}^*(L_{\text{right}}))\right]^2 \quad (2)$$

**Questions to Answer / Things to Implement**

- Collect an intial data set based on policy $\pi_0$. *We need to get our initial data from somewhere.*
- Train a model $\hat{C}^*_0$ based on minimizing the above loss function on your data.

  Tip: It may be useful to promote convergence to introduce discounting into your cost function, even though it isn't strictly needed.

- Show that your model learned and did not overfit.
- Define a policy $\pi_1$ (different from the previous $\pi_1$), defined by always taking the action that produces the smallest value of $\hat{C}^*_0(L_{\text{action}})$.
- Generate a data set based on policy $\pi_1$. *How can you / should you handle cases where policy $\pi_1$ doesn't solve the problem?*
- Train a model $\hat{C}^*1$ based on minimizing the above loss function on this data.
- Iterate this process: train a model to satisfy the $C^*$ equations (or close to it), define a policy based on this model, get new data, refine the model based on the new data.
- How does the policy this process produces perform? Are you able to get comparable or better results from previous policies (specifically $\pi_0$)? Why or why not?

The above outlines a basic **Reinforcement Learning** strategy - the bot has a policy for playing a game, generates data based on that policy. It models the data (to try to project to game states it did not necessarily see before), and then uses what it extracted from the data to try to determine better moves, iterating this process and (hopefully!) producing an optimal strategy.

*Note: Convergence of reinforcement learning strategies is often hard to guarantee or achieve, because we don't actually know what the value of the target $C^*(L)$ should be (in this case), we just know an equation it should satisfy.*

**Question to Answer:** Why is generating novel data each time you update the policy necessary?