# Lifelong Learning for Mobile Robot Task Completion

**Scott Sikorski**
CS6501-11 – Learning for Interactive Robots
University of Virginia
nqj5ak@virginia.edu

**Abstract:** I present an extension of previous lifelong learning frameworks and contributions for navigation to mobile robot task completion. Typically task completion is isolated to a small discretized environment without robot movement while navigation is done statically or improved with learning from experience. Recent work in navigation helps prevent previous mistakes without user interaction but causes the robot to suffer from catastrophic forgetting. This is when learning from new experiences causes previous knowledge to be overwritten. Lifelong Learning for Task Completion (LLfTC) aims to continuously learn how to move, interact, and complete tasks without forgetting what it learnt about previous environment. These objectives are balanced to accommodate onboard mobile robot hardware constraints. Simulations are used to display correctness of LLfTC with considerations of memory and computing expenses.

## 1 Introduction

Classical mobile robots have long been static planners that did not learn about the world, often causing bottlenecks without parameter tuning or expert domain knowledge. However, with the introduction of vision understanding networks and general navigation learning frameworks, mobile robots can begin to learn about environments they are placed into by exploring and reasoning about the world's properties. This method has been optimized well when presented expert demonstrations or if the robot is contained in the same environment.

However, the mobile robot and its underlying model become prone to catastrophic forgetting. This is a phenomenon where after learning about a new environment the robot completely forgets about previous environments. As a result, it must relearn everything. This is an inviable option when we are dealing with resource constrained mobile robots.

Lifelong Learning for Task Completion is a framework that combats catastrophic forgetting by ensuring that we do not update our parameters beyond the point of negatively effecting previous task learning it accomplished. This is done through an initial random policy that explores the environment and learns about it through a series of interactive actions. These actions are then identified as being useful to completing a goal task defined by us. This strategy is then bolstered by a learnable policy that can help guide our mobile robot to find actions that are deemed more useful given the state it finds itself in. In conjunction, a limited capacity of examples is given to the model to simulate the physical onboard memory and computation limitations that mobile robots face. The main contributes of this work are

- A review of previous robotic control and task completion methods
- A self-improving model that is used to guide task completion in a multitude of different environments
- A framework for learning a task in a new environment without forgetting the previous ones based on an optimality function

## 2 Related Work

### 2.1 Classical Planning Approaches

Within the realm of robotic planning exists two planning approaches, one for navigation and one for task planning. Both have different approaches to statically planning and learning from experience.

*A) Navigation* Typically, mobile robot navigation has been done using static planning algorithms such as Rapidly-exploring Random Trees (RRTs), A*, Dijkstra's, or Dynamic Window Approach (DWA). These all lack the ability to learn and can easily get stuck which requires human domain knowledge to tune away these problems. Recent work to help mobile robots learn have produce promising results when operating in single environments. However, these suffer from forgetting when there is no supervised data that the agent was trained. LLfN [1] focused on lifelong learning for navigation and produced results that showed navigation not suffering from forgetting over time.

*B) Task Planning* Work in task planning can be defined statically using Planning Domain Definition Language (PDDL) which defines preconditions, postconditions, states, and operators to achieve a goal. Recent work such as ALFRED developed by Shridhar et al. [2] explores learning from expert demonstrations to complete complex tasks.

### 2.2 Continual Learning

Powers et al. [3] explored efforts in continuous learning of home environments with new objects being added and for the robot to reason about their new importance to environment. They developed SANER and ABIP as methods to learn new skills. Other work in continual learning outside of pure robotics has mostly involved using regularization to prevent significant weight deviation or using a generative model to recover data or adapt based on the task and environment. Generative models are considered unviable on resource constrained machines, so we focus on weight deviation prevention.

### 2.3 Gradient Episodic Memory

As mentioned above, a motivating factor is to prevent catastrophic forgetting for a general use mobile robot operating in multiple environments. Gradient Episodic Memory (GEM) [4] is the primary method used to combat forgetting while continuously learning. The foundation is built on the idea that updating the model parameters, just $\theta$ in this case, for the current environment will not cause the loss on the collection of previous environments to increase. As a result, we will continue to learn through each iteration of navigating and learning about an environment without forgetting important information about the previous environments. The authors observe that GEM is unique in its ability to allow positive backward knowledge transfer. This means that previous environments can learn from the current environment so long as the loss does not decrease.

An important note to make is that GEM is not about generalizing. Rather it is focused on learning from concrete training examples of the best action to do in a given state/scenario. This point becomes more evident as GEM is limited to a subset of observed examples which we derive from our environment transitions. This is exceptional in our case due to the hardware resource constraints where heavy computation cannot take place.

## 3 Methodology

### 3.1 State and Action Definition

In an effort to reduce the memory footprint of LLfTC and eliminate object detection handling, states are abstracted away from the environment metadata. AI2THOR supports describing the current state that the robot sees as the raw image data. This would involve object detection from the pixel values of the image or mapping to the object ids, which was considered out of scope of this project. Thus, a state is defined as 3 main properties. 1) The agent's (x, y, z) position 2) The visible objects 3) The reachable objects. The position and visible objects are easily extracted from the metadata; while the reachable objects is defined over an API call that defines all reachable positions and if any visible objects are within the interaction distance. 0.25. The reachable objects will always be a subset of visible objects and are defined to heuristically guide our initial policy.

An action is simply a command given to the AI2THOR controller with a object id if necessary and a goal completion flag. All actions are compiled of 3 different types.

1. Movement - This includes moving forward or backward and rotating left or right. To reduce complexity, I only allow the robot to move in one dimension, x, causing movement in other dimensions to depend on rotating. Thus, they are considered independent actions.
2. Interactive Actions - These actions interact with objects in the environment and change properties of these objects. These range from picking up, cooking, breaking, etc. These types of actions require supplying the object to the controller and are handled differently with scoring.
3. Task Done - A terminating token action type that is used to help the model learn about the steps towards complex task completion.

## 3.2 Goal Tasks

We can define a goal task as an object, its goal end status to complete the task, and a dependent position to bring that object to. This position is only supplied if the end status requires it such as ḿove object. This goal task is only modeled as a subset of properties of the state of the environment which once match declares the state as completed. As a result, the task is learnt as a collection of state-action transitions. Additionally, a goal task cannot be repeated for multiple environments because the object is defined as the unique object id in the environment metadata.

For my experiments, I trained and tested on picking up an apple in 5 different environments. My goal task JSON for picking up an apple is defined as

{ "objectId": "Apple - x - y - z ", "status": "PickUp" }

where x, y, z is the unique position of the object for the specified environment. These tasks are naively directly indexed to the environment. So, if this id does not exist in the environment then the robot will never be able to finish the task and so will learn nothing.

## 3.3 Formal Problem Setup and Notation

- $M_k$: A set of states for a given environment $k$.
- $Buffer$: A set of state-action transition pairs from training through all environments. This is used to update $\pi_\theta$ after we finish an epoch of training.
- Difference Function, $\texttt{diff}(s_1, s_2)$. This assigns a difference value between two states and used to determine optimality of actions.

$$\begin{aligned}\texttt{diff}(s_1, s_2) = &\sqrt{(s_{1_x} - s_{2_x})^2 + (s_{1_y} - x_{2_y})^2 + (s_{1_z} - s_{2_z})^2} \\ &+ 10 \cdot (|s1.Visible| + |s2.Visible| - |s1.Visible \cup s2.Visible|) \\ &+ \sum_{i=1}^{n=|s1.Visible \cup s2.Visible|} \sqrt{(s_{1_{x,i}} - s_{2_{x,i}})^2 + (s_{1_{y,i}} - x_{2_{y,i}})^2 + (s_{1_{z,i}} - s_{2_{z,i}})^2}\end{aligned} \tag{1}$$

  - We can see that if $\texttt{diff} = 0$, the states are considered the same.

- Optimality Function, $B(s,a)$. This determines the optimality of an action in a state given the goal state. We use a simple Euclidean distance difference for movement actions. I prioritize actions that could provide more value while preventing undoable actions without being totally sure that this aids in completing the goal. This scoring function would benefit from more logic about other actions and be adapted to use reinforcement learning to better reasoning about the possible reward of an action towards more complex tasks. But, this work well for simple tasks not involving multi step actions of an object or involving receptacles.

$$B(s,a) = \begin{cases} diff(s, goalState) - diff(\pi(s,a), goalState) & \text{if } a \in \text{MovementType} \\ \sqrt{(s_x - o_{goal_x})^2 + (s_y - o_{goal_y})^2 + (s_z - o_{goal_z})^2} & \text{if } a \in \text{MoveObject} \\ 0 & \text{if } obj \in \text{Reachable Objects} \\ \infty & \text{if } a \in \text{Undoable Action} \end{cases} \tag{2}$$

- Initial Policy, $\pi$: $\pi$ is implemented as a simple RRT with one heuristic. If the state has the goal object in its reachable objects list, then it will automatically opt to interact with the object with the corresponding action. Else, it will randomly choose to move, rotate, or interact with any reachable object in the state, if any.
  - There was work on additional heuristics such as moving towards the goal object if visible. This didn't work due to the agent getting stuck when doing collision avoidance at times.
- Learnable Policy, $\pi_\theta$: The $\pi_\theta$ policy is defined as a GEM model that is parametrized over $\theta$ and updated after finishing a task in an environment. $\pi_\theta$ outputs any action given a state as input.

### 3.4 Training and Testing

Training the agent takes place over 10 epochs, each of which iterates through each environment specified by the user. We begin by initializing $Buffer = \emptyset$, determining the goal based on the environment the agent is moving in, and $\theta$ = random. We now begin engagement with the environment that we are in. The memory of states is set to empty for the current environment $k$, $M_k = \emptyset$. After the environment is setup properly, the agent will follow the following steps until timeout or completion.

- Pick an action according to the $\pi$ policy.
- Execute the action
- If the action cannot be executed, such as a table restricting movement, simply discard the action and pick a new action according to $\pi$
- Check if the goal flag is completed
- Add the $(s,a)$ pair to $Buffer$ and progress to next state

Once the robot finishes the task, $Buffer$ is shrunk to $n = 300$ $(s,a)$ pairs. I elect to eject pairs that are accompanied by the highest $B$ values. The model is then given $Buffer$ as new inputs to learn from in its existing dataset. The GEM model observes all 300 example data points and aims to optimize the following

$$\underset{\theta}{\operatorname{argmin}}\, l(\pi_\theta, \epsilon_k) \; s.t. \; l(\pi_\theta, M_k) \leq l(\pi_{\theta_{k-1}}, M), \forall M \in B \tag{3}$$

The gradient of the cross entropy loss function is calculated for all of the previous memories and updates $\theta$ to obey the above condition of the optimization.

Testing then takes place following a relatively similar process as training. The only difference is that we sample actions from both $\pi$ and $\pi_\theta$. We take the action that is successful or has the highest benefit to achieve goal. This is simply defined by the optimality function $B(s,a)$ so simply

$$a = \underset{s',a'}{\operatorname{argmin}} B(s', a') \tag{4}$$

## 4 Experimental Results

### 4.1 Model Design

| | Layers | Hidden Nodes | Learning Rate | Num of Memories | Memory Strength |
|---|---|---|---|---|---|
| LLfTC GEM Model | 2 | 100 | 0.001 | 300 | 0.5 |

| | Num of Inputs | Num of Outputs | Num of Tasks | Epochs | Loss Function |
|---|---|---|---|---|---|
| LLfTC GEM Model | 3 | 25 | 5 | 10 | Cross Entropy |

Table 1: LLfTC $\pi_\theta$ Specifications

The GEM model that was used for LLfTC is specified with the above hyperparameters. More importantly, the structure of the model is a simple MLP that is a fully connected network of 3 inputs, 2 layers of 100 hidden nodes, and outputs a classification of 25 outputs corresponding to an action. These inputs are the agent's (x, y, z) position within the environment. GEM is then built on the number of memories (300) for instances of this network which we can view in GEM's memory dataset to ensure that the loss over these examples does not decrease with more examples.

We keep our budget of 300 memories across all environments as the continuously updating training dataset. These examples are not evenly distributed as they are kept based on lowest difference score, which is computed during buffer shrinking. As a result, there could be bias in which environments' memories have more influence. This bias could lead to some environments being highly optimized while others get stuck in corner cases. Additionally, GEM assigns a 0.5 weight to previous memories so that we can still learn at a high enough level from the current environment task completion.

Lastly, during training we train for 10 epochs to provide the mobile robot the ability to continuously learn from all the environments after having finishing the task. This helps to optimize the robot's movement in the environment and to reach the goal area more quickly. This would also prove to be beneficial when we start assigning the agent complex tasks to do. For example, if we were to grab an egg from the fridge, the agent would learn that the egg is in the fridge and it needs to open the fridge to see it. This would taken multiple training epochs to achieve this. Additionally, this epoch value could be increased to provide the agent more opportunities to finish tasks during completion, which was a problem has seen in Table 2 and discussed more.

### 4.2 Results

| Environment | # of LLfTC Actions | # of RRT Actions | Completed |
|---|---|---|---|
| 1 | 4 | 10 | T |
| 2 | 200 | 200 | F |
| 3 | 50 | 134 | T |
| 4 | 51 | 99 | T |
| 5 | 200 | 200 | F |

Table 2: Testing Results for Pickup Apple Goal

We can observe that during testing there was a success rate of $\frac{3}{5}$. Environments 2 and 5 reached the max number of actions that I elected to use based on a timeout of 100 seconds. Overall, LLfTC decreased the number of actions taken in an environment by more than 2 times if there was just a regular RRT used for exploration and finding the apple. These results more so showcase the ability of the robot to explore and find where the apple is located.

As well, the model seemed to be bias towards certain environments, most of which depended on the starting position of the agent and the position of the apple. For instance, if there were in separate corners of the environment, it was hard to expose the agent to all parts of the environments. As well, the model does not explicit reason or learn about the properties about the objects. It is merely learning the most optimal actions to execute in any given state. And for states that it has not seen before, it is able to predict which action to execute based on the similarity, or low difference in our definition, of that state to one of the states in its data points.

| Environment | Avg # of Completions | Avg # of Fails | Avg # of Actions |
|---|---|---|---|
| 1 | 1 | 1 | 8 |
| 2 | 0 | 183 | 200 |
| 3 | 0.1 | 22 | 155 |
| 4 | 0.8 | 5 | 120 |
| 5 | 0 | 200 | 200 |

Table 3: Training Results for Pickup Apple Goal

We can also just the training results as some insight into some of the hidden benefits that LLfTC contains. During training if the agent never completes the task, then the model has no ability to actually learn from any mistakes it was making or rather find the sequence of actions that would lead to task completion. On the other hand, if the agent was able to find a completion path during training, even if it is just one, then the model is able to complete the task during testing. This is observed in environment 3. As well, if the task is completed then on additional epochs, the agent does not forget the path and serves to optimize the path and reduce action amounts.

Another trend from these results is that the number of actions necessary to complete a task and number of fails in any given environment begins to decrease once the model learns the completion sequence. Environments 1 and 4 demonstrate lower fail rates and lowering number of actions from training to testing. This shows LLfTC's ability to aid continual learning over time while also preventing tasks to fail from previous environment.

## 5 Limitations and Improvements

As evident from Table 2, the agent failed to complete the task given to it in the given time. This means that that agent did not learn how to properly complete the task or more so find the state(s) that would lead to task completion. This is a severe dysfunction in the agent, especially if it cannot guarantee that it will eventually lead to success. Better training would need to take place.

One of the main experimental limitations is that there was a sole focus on one type of task, picking up an apple shown above. There was some training on other objectives that were single object and can be found just exploring the entire environment and it being visible. These include Toggling the Light Switch On or opening the fridge door. However, there was a lack of testing on complex tasks like cooking an egg, which would involve opening the fridge, grabbing the egg, and cooking it on the stove. These would take too long in the current implementation due to the agent needing to find the path randomly. But LLfTC would mostly likely shine with more work in improving the training process so that the path can be found more easily during testing.

Like briefly noted, during training the agent finds the path and so learns the optimal actions to achieve the goal state. This propagation of reward is only realized when the agent finishes the task successfully. So a possible improvement would be to increase training time by letting it run until completion or increase epochs. That approach is messy and is heavily reliant on luck. Integration of reinforcement learning into the training process would serve to bolster the exploration process and be more applicable to finding actions that would provide benefit in complex task sequences. This obviously would inherently benefit our optimality function and may even remove the need for it.

Finally, GEM was released in 2017 and the model that was used with GEM was a simple MLP. While GEM was a one of a kind method to prevent catastrophic forgetting, it also set a hard constraint of loss never decreasing. This is the best case scenario; however, we are possibly forging massive improvements to the theta parameter or learning in later environment. This is compounded that data is incrementally given to the model, environment buffers one at a time. So possibly the model has low loss and cannot back out of that now, causing bias to the earlier environments. Including a threshold parameter that decays over time could help us balance this. Or more work in catastrophic forgetting prevention can be explored.

## 6 Conclusion

In this report, I present a extension of current work in lifelong learning approaches for mobile robot. Previously, lifelong learning was showcased to be effective in learn how to navigate in new environments without losing the ability to navigating previous environments. LLfTC is a new framework for mobile robot task completion. The primary motivation behind LLfTC was to address the challenge of catastrophic forgetting in the context of mobile robots operating in diverse environments while continually learning. A task adapted RRT was used to explore the world and its properties. A learnable policy $\pi_\theta$ was implemented to act as a guide for this planner to help optimize approaches and prevent failures.

LLfTC demonstrates promise in enhancing mobile robot learning for task completion across various environments. Future work could focus on refining the training process, incorporating reinforcement learning, and expanding the scope to tackle more complex tasks, ultimately advancing the capabilities of lifelong learning for interactive robots.

## 7 Supplementary Material

The artifact for this report can be found at https://github.com/sgsikorski/LLfTC. Most of the source code lives in the root folder within main.py, Agent.py, and lifelonglearning.py, barring reorganization into a more comprehensive src folder. GradientEpisodicMemory acts as a submodule from Lopez-Paz and Ranzato [4], which can be found at https://github.com/facebookresearch/GradientEpisodicMemory/tree/master.

## References

[1] B. Liu, X. Xiao, and P. Stone. A lifelong learning approach to mobile robot navigation, 2021.

[2] M. Shridhar, J. Thomason, D. Gordon, Y. Bisk, W. Han, R. Mottaghi, L. Zettlemoyer, and D. Fox. Alfred: A benchmark for interpreting grounded instructions for everyday tasks. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 10737–10746, 2020. doi:10.1109/CVPR42600.2020.01075.

[3] S. Powers, A. Gupta, and C. Paxton. Evaluating continual learning on a home robot, 2023.

[4] D. Lopez-Paz and M. Ranzato. Gradient episodic memory for continual learning, 2022.