Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Global Way-point LSTM Planner: An Online Machine Learning Solution for Robotic Path Planning

Supervisor:
Sajad Saeedi

Author:
Alexandru Iosif Toma

Co-supervisor:
Ronald Clark

Second Marker:
Andrew Davison

June 17, 2019

Abstract

Path planners have indispensable applications in physical autonomous systems such as robots and self-driving cars, in which human intervention is not desirable: bomb defusing, search and rescue, large scale manufacturing, warehouse management and many more. However, some of the classic solutions satisfy only a subset of the real-world application requirements imposed by hardware limitations. The work from this report shows that by adopting a hybrid approach between classic solutions and Machine Learning methods, we can develop a path planning solution that has support for partial knowledge environments while theoretically satisfying the real-world constraints. We create a platform for testing and developing the proposed solution while offering support for *ROS*, a defacto in robotic systems. We theoretically and empirically assess the performance of the proposed solution against the well-known pathfinding solution, A*. We prove that the proposed solution achieves theoretically lower average case time and space complexity and significantly reduces the memory load compared to A*. Moreover, we show that the proposed solution is robust to unknown environments by evaluating it on real-world occupancy grid maps. Lastly, we run the proposed solution on a real-world robot and show that it can find a reasonable path in partial knowledge environments in which offline solutions such as A* cannot.

Acknowledgements

I would like to thank my supervisor, Sajad Saeedi for offering continuous support and invaluable advice in regards to the project. I would like to thank my co-supervisor, Ronald Clark for motivating me to pursue different objectives. I would like to thank my second marker, Andrew Davison for inspiring me to address real-world robotic applications in the Interim Report. I would like to thank the Percepbot team for offering me the robot hardware for the real-world evaluation. I would like to thank my friends, Alexandru Dan and Andrei Isaila for proof-reading my report. Lastly, I would like to thank my parents and all my friends for their emotional support during my university years.

Contents

Chapter 1

Introduction

Path planners have essential applications in physical autonomous systems, such as robots and self-driving cars, as they have to safely follow an efficient, collision-free trajectory to their destination [1, 2]. Nowadays, autonomous systems are indispensable, their purpose being, the automation of tasks that cannot be performed at large scale or in a safe manner by a human being. Some examples include military applications (e.g. bomb-defusing robots), search-and-rescue robots, large scale manufacturing robots and warehouses management robots [1]. As a consequence, it is crucial to develop an efficient path planning algorithm that would plan a safe journey for the robot [2].

We argue that in real-world path planning, the robots interact with the real world environment and thus are subject to physical constraints. Therefore, the path planning decision algorithm has to satisfy various requirements imposed by hardware limitations [3]:

- **Resource Load** - The solution has to be computationally and memory efficient by taking into account the hardware limitations imposed by the robot architecture
- **Partial Knowledge (Online Planners)** - Usually in robotics, the planner has to find a solution with partial knowledge about the map. Thus, the algorithm has to explore the map while searching for a solution (i.e. the algorithm becomes online; e.g. a robot with a SLAM sensor has only localised information, until more areas are discovered)
- **Dynamic Environments** - The real world is highly dynamic and local path planning algorithms should support real-time path generation and collision detection
- **Robustness to Unknown Environments** - The solution should generalise well in unknown environments by preserving all other constraints
- **Non-holonomic Constraints** - Most robots (e.g. self-driving cars) do not possess the ability to move in all available degrees of freedom instantaneously. Therefore, the solution should plan a trajectory that can be followed with maximal efficiency (e.g. a car has to follow a curved path when turning to the left or right). Lastly, the path should also take into consideration the available physical capabilities of the robot (e.g. a robot has limited manoeuvring)
- **Randomized Kinodynamic Planning** - Path planners should generate a path based on the imposed vehicle constraints such as velocity, acceleration and torque. Therefore, we can avoid collisions with obstacles due to high velocity
- **Higher Dimensional Scaling (3D)** - The solution should be easily and feasibly extended to robots with higher degrees of freedom (e.g. drones and flying robots)

Currently, there are a lot of classic solutions to the pathfinding problem. To mention some of the most important ones: A* [4, 5, 6, 7], Rapidly-exploring Random Tree (RRT) family [8, 9, 3, 10], Value Iteration on Markovian Decision Processes (MDP) [11, 12]. However, most of the

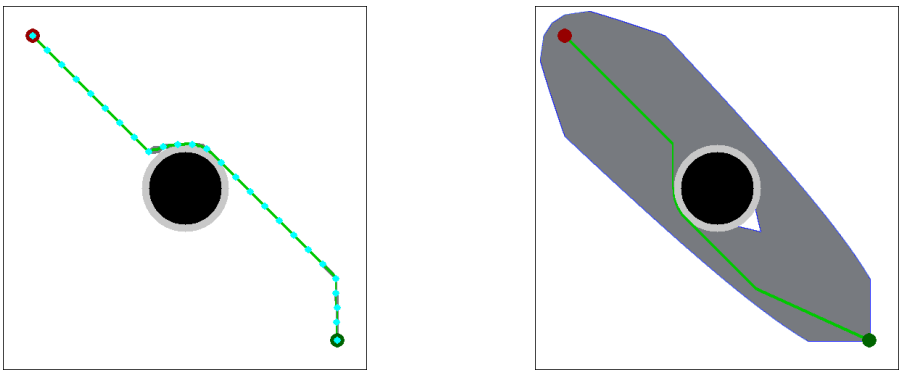
classic algorithms are computationally expensive because they have to search a vast area. A* and Informed RRT* are pruning the search space by adopting a heuristic function h (e.g. the Euclidean distance between the agent and the goal or the ellipsoid heuristic respectively), but even so, if the environment is complex (e.g. contains unusually shaped obstacles, the path is meandering), the search is not pruned enough. Moreover, the computational cost and memory increase exponentially with the dimension of the environment. Additionally, there exist environments (e.g. maps without a metric space such as networks) where finding a proper heuristic function is not trivial. Lastly, most of the classic algorithms are offline, and thus, they require full knowledge about the map.

1.1 Objective

This project aims to investigate if Machine Learning (ML) methods are a viable option for path planning applications. We will attempt to develop a ML solution that solves the path planning problem while satisfying the real world industrial applications requirements. We are going to focus mainly on the partial knowledge, resource load reduction (memory efficiency; See Figure 1.1) and robustness to unknown environments properties by running empirical evaluations while only theoretically prove the others.

The proposed solution will be developed using an LSTM [13] (Long Short-Term Memory) recurrent neural network architecture. The motivation behind choosing an LSTM architecture is derived by the fact that the path planning algorithm can be described as a sequential problem and it has been proved that LSTMs produce excellent results in this area (e.g. speech recognition, time series, anomaly detection, text generation, machine translation and many more) [14]. There has been some work done with LSTMs [15, 16, 1] in this field, but their approach had a low success rate of finding a path to the goal (even if one exists). Therefore, we will attempt to boost the success rate by creating hybrid solutions between pure ML solutions and classic solutions such as A*.

In order to achieve the objectives mentioned above, we are going to use a simulation platform. Currently, there are a variety of standardised motion planning libraries such as: *ROS* [17], *OMPL* [18], *MoveIt* [19] (which has benchmarking capabilities [20]). However, we are going to build our development platform, PathBench. The main reason behind this choice is that we are going to focus on the actual generation of the path instead of the physical interactions between the robot and the environment. Thus, we boost development productivity while focusing on the key components of the solution. Furthermore, by creating standardised APIs, we can easily port the path planners to the specified libraries. Finally, we need training data, a ML pipeline and a benchmarking module which the current libraries do not provide.



(a) Global Way-point LSTM Planner (proposed solution)

(b) A*

Figure 1.1: Memory load comparison between the Global Way-point LSTM Planner (proposed solution) and A*. The red circle represents the agent (robot) position, the green circle represents the goal position and the black/light-grey regions represent obstacles. The dark grey regions represent the total search space used by the respective algorithms

1.2 Contributions

The contributions of this report include (the source code can be found at https://gitlab.doc.ic.ac.uk/ait15/individual-project):

- Three algorithmic contributions:
  - **CAE Online LSTM Planner** - The first proposed solution, Online LSTM Planner, is almost identical to the paper implementation from [15] with different input selection and the addition of a max iterations argument. The CAE Online LSTM Planner is a hybrid solution between [15] and [1]. The Convolutional Auto-encoder (CAE) architecture was built by us, and the LSTM architecture was borrowed from [15]
  - **LSTM Bagging Planner** - This algorithm is inspired by ensemble learning solutions, and it is used to boost the performance of the Online LSTM and CAE Online LSTM Planners
  - **Global Way-point LSTM Planner** - The final proposed solution uses the LSTM Bagging Planner to create a global way-point suggestion algorithm. It has nice flexibility properties, significantly reduced memory load compared to A*, support for partial knowledge environments, robustness to unknown environments and a theoretically lower average case time and space complexity compared to A*
- PathBench - A benchmarking platform for classic and learned path planning algorithms with the following four main components and extension:
  - **Simulator** - We have built a simulator as a practical way of visualising the behaviour of the path planners. It includes different features such as animations, control for animations (stop, resume), custom displays for individual algorithms and it is highly extensible to support new solutions
  - **Generator** - The generator was build to acquire training data for our ML solutions. It can generate three types of maps: uniform random fill maps, block maps and house maps. It can also be extended to include more synthetically generated data (e.g. cellular automata cave generation and maze generation) as well as convert real-world datasets (e.g. Simultaneous Localisation and Mapping (SLAM) images [21]) into internal simulator environments. All maps can be converted into training datasets which contain a variety of features and labels generated using A* as ground truth
  - **Trainer** - We have built a training environment to boost the productivity of testing new ML architectures. It has an automatic pipeline for extracting a subset of the generated training data and caching subroutines to increase the speed of second runs
  - **Analyser** - We have developed an analyser tool to assess the performance of the proposed algorithms against classical solutions such as A*. The analyser contains multiple assessment routines which stress the performance of the tested algorithms in multiple areas (e.g. speed, efficiency and memory)
  - **ROS Real-time Extension** - We have added support for real-world simulation by implementing an updatable map environment which is compatible with the *gmapping ROS* package (i.e. SLAM scan)
- Theoretic and real-world evaluations performed as follows:
  - **Complexity and Theoretical Analysis** - For each proposed solution, we have theoretically proven the worst case time and space complexity. Moreover, we have discussed the worst case time and space complexity of higher dimensional scaling. Lastly, we have proved that the proposed solution achieves theoretically lower average case time and space complexity compared to A*
  - **Empirical Methods** - We have created specific statistical metrics for each proposed solution. The metrics are used to gain insight into the general behaviour of the proposed solutions and their real-world applications effectiveness. We have showed that the proposed solution significantly reduces the memory load compared to A* (See Figure 1.1) and that it generalises well in unknown environments

– **Real-world Evaluation** - We have tested the performance of the proposed solution on real-world occupancy grid maps generated by real-world robots. Furthermore, we have implemented the proposed solution on a real-world robot and tested it at Imperial College London. We have proved that the proposed solution is applicable in real-world scenarios. Moreover, we have proved the partial knowledge property by showing that the algorithm can run in environments with partial information in which some classic offline algorithms such as A* are unable to find a solution

1.3 Report Outline

Literature Review (Chapter 2). In this chapter, we will study the current path planning solutions by categorising them into separate sections based on the type of planner. We will assess the optimality conditions, give a brief time and space complexity analysis, study the structure of the algorithms and state the advantages and disadvantages of each solution.

Background (Chapter 3). The Background Chapter will cover the basics of ML and explain the structure of the LSTM network. We will discuss data pre-processing, training routines, evaluation methods and hyper-parameter tuning.

PathBench (Chapter 4). In this chapter, we are going to describe PathBench, the platform used to develop and test the classic and proposed solutions. We will cover the architecture design of all platform components and explain the current capabilities and limitations by comparing it to the standard motion planning libraries.

Methods (Chapter 5). In this section, we are going to describe the proposed solutions by following the same investigations as in Chapter 2 (Literature Review). Moreover, we will compare the theoretical performance against the well-known algorithm A* by giving a thorough time and space complexity analysis for all proposed solutions.

Evaluation (Chapter 6). In this chapter, we are going to run a series of empirical evaluation routines which will stress the proposed solution performance. We are going to examine each empirical run and give a theoretical interpretation of the results. Moreover, we will test the performance of the path planner on real-world occupancy grid maps produced by real-world robots. Lastly, we will run the proposed solution on a real robot at Imperial College London and assess its performance.

Conclusion (Chapter 7). In the final chapter, we are going to summarise our findings and address future work.

Chapter 2

Literature Review

We are going to divide the classic algorithms into four sections following the model from [2]: Graph Search Planners (Section 2.1), Sampling Based Planners (Section 2.2), Interpolating Curve Planners (Section 2.3), Numerical Optimization Approaches (Section 2.4).

When discussing each algorithm, we are going to state how it solves the problem, how it is implemented, optimality conditions, worst case time and space complexity analysis and some of the advantages and disadvantages.

However, before starting, let us quickly formalise the pathfinding problem so that we will have a standard notation throughout the review. We have an agent A that wants to get to a goal G and a set of obstacles Os which the agent tries to avoid. Each of the entities belongs to a map $M = (A, Os, G)$. The purpose of the algorithm is to produce a trace, denoted by T, which represents the history of the agent moves from the initial agent position to the goal position. The agent can move one step at a time to a position that is valid within the map (not out of bounds or not colliding with any obstacles). The goal is reached by the agent only when the agent position matches the goal position exactly. Furthermore, we are going to assume that the environment is static and fully discovered (the algorithm does not need to explore the map while searching for a solution).

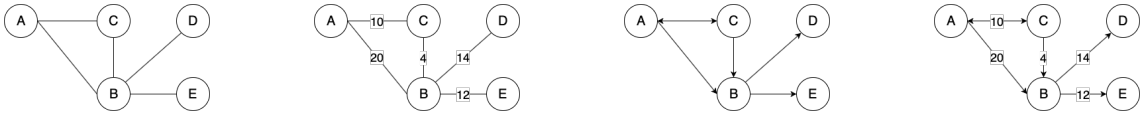
As a general rule, when we inspect the map figures, the entities will be represented by circles or squares. The colour convention will be the following: the agent is red, the initial agent position is dark red, obstacles are black, the goal is dark green (or magenta in some maps where the goal is not noticeable), the trace is light green, and the clear path is white. All map figures have been generated using the simulator from PathBench (Section 4).

2.1 Graph Search Planners

The discussed algorithms from this section represent a classic solution to the path finding problem. The majority of them require the world map to be represented as a graph or grid [4].

A graph (See Figure 2.1) is a data structure that is composed of nodes (vertices) and edges usually represented as $G = (V, E)$ where V is a collection of nodes (vertices) and E is a collection of edges. The edges can be undirected (undirected graph; bidirectional movement) or directed (directed graph; unidirectional movement). Each edge can have an associated weight (weighted graph) or not (unweighted graph), which might represent the movement cost between two nodes [4].

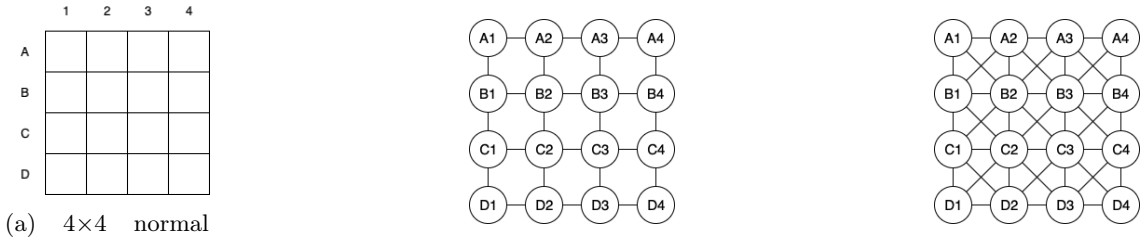
A grid (See Figure 2.2) is a two-dimensional table that allows movement to nearby cells. This data structure can be easily translated to an undirected unweighted graph where grid cells represent



(a) Undirected unweighted graph (b) Undirected weighted graph (c) Directed unweighted graph (d) Directed weighted graph

Figure 2.1: Graph examples. The nodes are represented with a circle, and the identifier is the letter inside them. The edges are represented by lines or arrows, and the values from them represent weight values

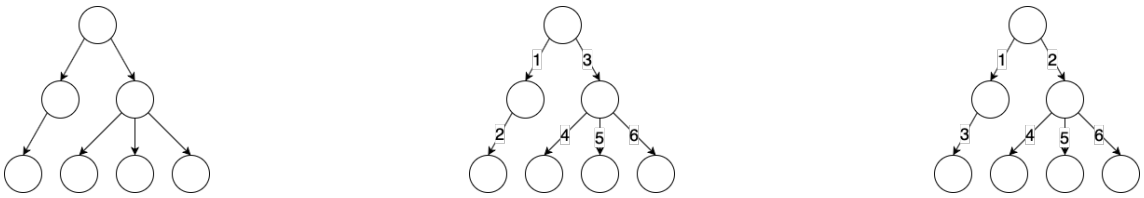
nodes and neighbours represent undirected edges. The neighbours are defined in terms of the grid connectivity type which can be 4-point connectivity (up, down, left, right) or 8-point connectivity (all 4-point connectivity neighbours, principal diagonal and secondary diagonal) [4]. We are going to assume 8-point connectivity for all grids unless explicitly stated.



(a) 4×4 normal grid (b) 4-point connectivity graph representation (c) 8-point connectivity graph representation

Figure 2.2: 4x4 grid example with associated 4-point and 8-point connectivity graphs. In 4-point connectivity, the neighbours are up, down, left and right. In 8-point connectivity, the neighbours are all 4-point connectivity neighbours, the principal diagonal and the secondary diagonal. The numbers at the top and left of the grid represent coordinates

A tree (See Figure 2.3) is a directed graph with a root (i.e. a node that has no incoming edges) where each node has directed edges to their children. A particular property of the tree is that it contains no cycles. This data structure is essential as most algorithms have to walk the graph in some way (depth-first search, breadth-first search) in order to discover a possible path. Depth-first search (DFS) [22] is a graph walking method that can be implemented using a stack (Last In First Out (LIFO) data structure) or recursion (stack is preferred, due to the recursion depth constraint that most programming languages incorporate). DFS starts by "expanding" the root (i.e. visit the node and push the node's children onto the stack) and then iteratively expands the latest node from the stack. Thus, we initially visit the first children of the node we expand and then visit its children recursively before continuing with the second child. Breadth-first search (BFS) [22] is another graph walking method that uses a queue (First In First Out FIFO data structure). BFS starts by expanding the root and then it repeats the process iteratively for the children. Thus we first visit **all** of the children of the node we expand and then we continue to expand each child. When BFS has to expand a node, it chooses the one in the front of the queue, and then it puts **all** of the children at the end of the queue. The result of both methods is a tree [4].



(a) A tree (b) DFS walk (c) BFS walk

Figure 2.3: An example of a tree along with a DFS and BFS walk. The numbers indicate the order of expansions

The worst case time complexity of DFS is $\mathcal{O}(b^d)$ and the worst case space complexity is $\mathcal{O}(b \cdot d)$ if a stack is used and $\mathcal{O}(d)$ if recursion is used, where b is the branching factor (the average number of a node's children) and d is the visiting depth. Time complexity is trivial as at each step we run a new search for all the node children b, and we do this d times. If we use a stack, for each node, we push all children onto the stack. We do this d times, and each node has b children. Therefore the space complexity is $\mathcal{O}(b \cdot d)$. When using recursion, the space complexity is lower as it is defined in terms of the recursion depth (we do not use all children at once at each recursion step). However, we still prefer to use the stack in practice, due to the programming language limitations [22].

The worst case time and space complexity of BFS is $\mathcal{O}(b^d)$. The time complexity follows the same reasoning as DFS. Space complexity is given by the number of nodes in the queue at one time which is equal to the number of the nodes on each layer of the tree which is $\mathcal{O}(b^d)$ as each layer node count grows exponentially [22].

When talking about complexities we have opted for the b (branching factor) and d (depth) notation instead of the $|E|$ (number of edges) and $|V|$ (number of vertices/nodes) (e.g. DFS space complexity is the same as BFS space complexity $\mathcal{O}(|V|)$). We are going to discuss algorithms which attempt to prune the search space, and we can infer more information from the first notation rather than the second one [22].

In practice, we prefer to use BFS when dealing with problems that attempt to find an optimal solution (due to the visiting pattern) and DFS when we want to visit the whole tree without caring about the visiting pattern or when we have memory constraints.

2.1.1 Wave-front Planner

The Wave-front Planner algorithm [4, 23] (See Figure 2.4) is one of the simplest solutions to the pathfinding problem. The algorithm can only run on grids (two dimensional or higher). The main idea is to have a separate grid with initial values of 0 then "propagate" a wave from the goal position to the agent position. Thus, we essentially create a potential function on the grid.

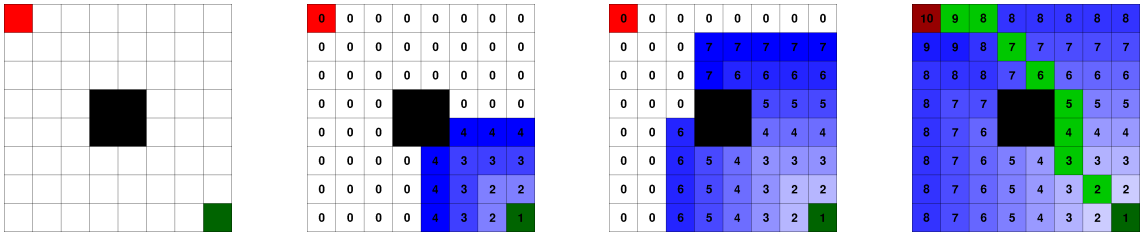
The wave is propagated by applying BFS to the separate grid from the goal position and then labelling the nodes on the same level of the tree with the level number. Thus, we first visit all the (valid) neighbours of the goal and mark the positions on the new grid with a 2 (as the goal position is marked with 1). Then we repeat the process for the nodes labelled with a 2 by expanding them and marking their **not visited** neighbours with a 3. The process is repeated until we hit the agent position. After that, we apply gradient descent from the agent position to the goal position. We start from the agent marked with number x and then search its neighbours for a node marked with $x - 1$. If there are multiple choices we can choose a random number as the invariant of the algorithm states that, given any node, the distance between itself and the agent is the absolute difference between their grid values (See Algorithm 1).

$$dist(n) = abs(grid(G) - grid(n))$$

We can easily prove that the algorithm finds the optimal solution in terms of minimum distance as we have used BFS to expand the nodes. If we would have used DFS instead, we could have still found a solution, but it would not necessarily be optimal due to the visiting pattern in DFS. The path might not be optimal in environments where the edge transition cost is not equal in all directions (e.g. moving on the diagonal ($\sqrt{2}$) is more expensive than moving vertically or horizontally (1) based on the Euclidean distance).

The worst case time and space complexity are given by the visiting method (BFS in our case) which is $\mathcal{O}(b^d)$, where b is the branching factor, and d is the depth of the solution (*Get-Backtrace* is $\mathcal{O}(d)$).

One of the major drawbacks to this approach is that the optimal path is dangerously close to the obstacles (as only the attraction function is used) and in the real world, it might lead to collisions.



(a) Starting 8x8 grid (b) Iteration 1 (c) Iteration 2 (d) Final trace

Figure 2.4: Wave-front Planner algorithm run on an 8x8 grid (4 iteration points shown: start, 1, 2, final trace). The red square represents the agent start position, the dark green square represents the goal position, black squares represent obstacles, white squares represent the clear path, light green squares represent the final path chosen by the algorithm. The numbers in each grid cell represent the gradient map and the white (min) - dark blue (max) gradient colour is another representation of the gradient map

It is also computationally expensive as the planner's search space is quite large since it does not apply any pruning. Moreover, the time and space complexity increase exponentially with the dimension of the environment. In a 2D grid with 8-point connectivity b is 8, but in a 3D grid world b is $3 \times 9 - 1 = 26$ (we subtract the (0, 0, 0) direction), where b is the branching factor. A nice way to visualise the increase of b is to count the number of combinations on each direction. Each coordinate has 3 configurations (1, 0, -1) and the number of combinations is given by 3^D, where D is the dimension. Therefore, $b = 3^D - 1$ (we have to subtract the (0, 0, ..., 0) configuration as it is not a valid neighbour). Therefore, time and space complexity becomes $\mathcal{O}((3^D)^d)$. However, because we are working with real-world robots, we can only consider 2D and 3D environments (2D for ground robot, 3D for drones and flying robots).

Algorithm 1 Wave-Front Planner

```
1:  procedure GET-BACKTRACE(step_grid, M: (A, Os, G))
2:      trace ← [A]
3:      while current is not A do
4:          current ← ∀n ∈ Neighbours(current).step_grid[n] = step_grid[current] − 1
5:          add current to trace
6:      return trace
7:
8:  procedure WAVE-FRONT-PLANNER(M: (A, Os, G))
9:      Initialize queue q with (1, G)
10:     Initialize step_grid as array with same size as map
11:     visited ← {}
12:
13:     repeat
14:         (current_count, current_node) ← pop front q
15:         add current_node to visited
16:         step_grid[current_node] ← current_count
17:
18:         if current_node is A then
19:             follow Get-Backtrace(step_grid, M)
20:             return
21:
22:         for each neighbour in Neighbours(current_node) do
23:             if neighbour is not in visited then
24:                 append (current_count + 1, neighbour) to q
25:     until q is empty
26:
27:     goal was not found
```

2.1.2 A*

The A* algorithm [4, 5, 6, 7] (See figures 2.6, 2.7) is mostly designed to run on weighted graphs, but can be adapted to run on grids as well if we convert the grid to a graph with weights based on the Euclidean distance (1 for vertical/horizontal movement and $\sqrt{2}$ for diagonal movement).

The difference between A* and Wave-front Planner is that A* aims to prune the search space by using a heuristic function h which is usually the Euclidean distance (8-point connectivity) or the Manhattan distance (4-point connectivity) (See Figure 2.5).

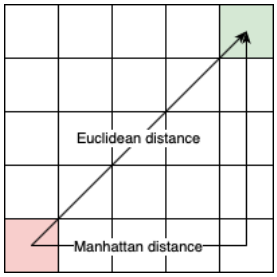


Figure 2.5: Euclidean and Manhattan distances. Red square is agent and green square is goal

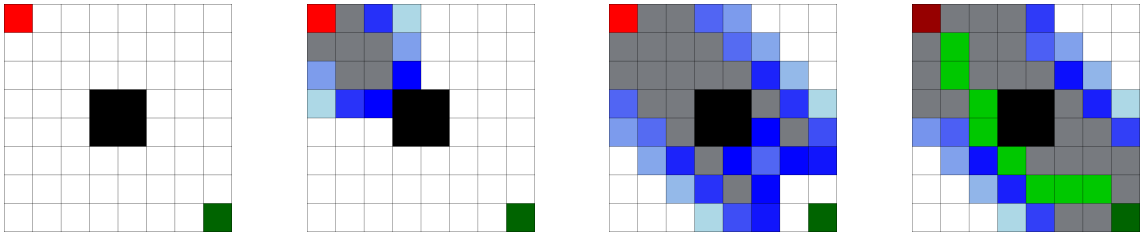
It is implemented using a priority queue in which the priorities are a function $f(n) = g(n) + h(n)$, where h is the heuristic function mentioned above and g is a function which represents the total actual distance travelled from the agent to the node n. The cost function $c(x, y)$ returns the graph weight cost from node x to node y (x and y have to be connected). The algorithm starts with the agent node in the priority queue. Then, the element with the highest priority (i.e. lowest $f(n)$) is picked and expanded, and its children are inserted into the priority queue. Each child will have a parent p, which will help us trace back the path at the end. When we expand a child we set $p(child) = n$. If a path is found, we update the goal parent. We stop the process only when the priority queue is empty or when the next priority is less than or equal to the optimal distance found. All expanded nodes are marked as seen, so we do not have to revisit them. When we expand a node, if any of its children are already in the queue we add them again if the path through the current node gives a higher priority ($g(n) < g(child)$) and update the child's parent ($p(child) = n, g(child) = g(n) + c(n, child)$). It does not matter if we add the children again to the queue as we know that $f_{cur}(child) < f_{old}(child)$ so we will visit the higher priority option first. When we stop, we compute the trace by recursively looking at the next parent from the goal until we find the agent (See Figure 2.7, Algorithm 2).

A* finds the shortest path, if and only if, the heuristic function is optimistic. An optimistic heuristic function is always less than or equal to the shortest distance.

The worst case time and space complexity of A* is $\mathcal{O}(b^d)$, where b is the branching factor, and d is the depth of the solution because the underlying algorithm structure is similar to BFS (*Get-Backtrace* is $\mathcal{O}(d)$). However, by using a good heuristic function, the time and space complexity becomes $\mathcal{O}(\hat{b}^d)$ where $\hat{b}$ is the reduced branching factor (i.e. A* prunes the search).

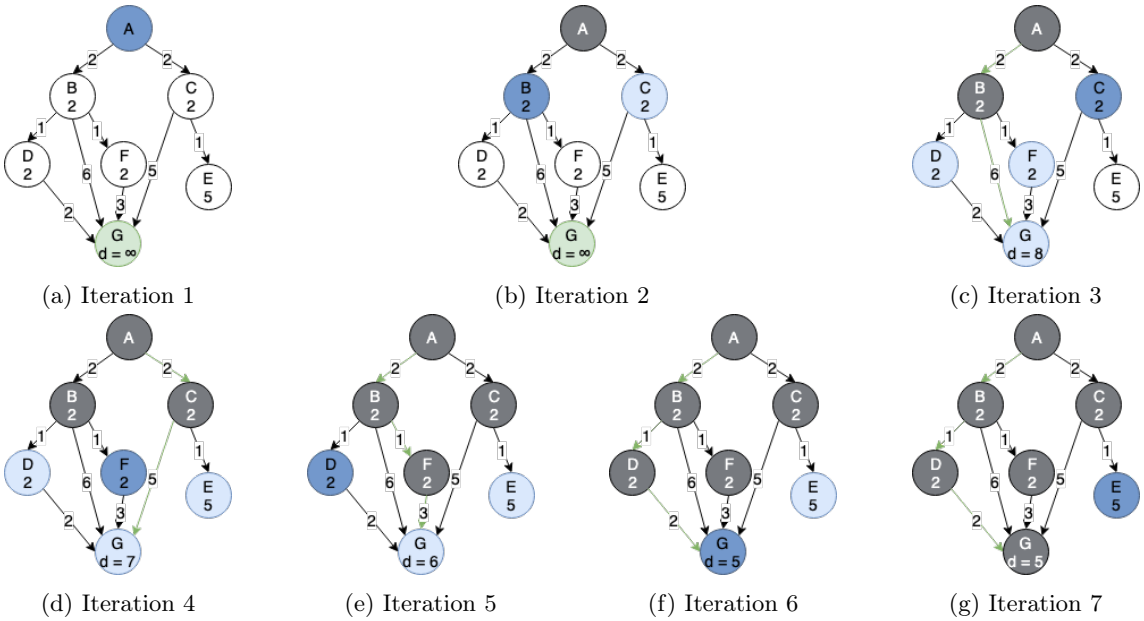
The significant advantage of A* is that the search space is pruned quite a lot, but it still shares the same issue as the other graph search planners: the search space grows exponentially with the grid dimension.

One of the major drawbacks is that it is not trivial to find a proper heuristic function for some environments (e.g. maps that do not define a metric space such as networks). In general, the Euclidean distance or the Manhattan distance are good choices for the heuristic function. Lastly, A* is an offline method (the internal state of the algorithm cannot be externally modified) meaning that external updates to the map are forbidden. Therefore, offline algorithms such as A* do not support dynamic and partial knowledge environments.



(a) Starting 8x8 grid (b) Iteration 1 (c) Iteration 2 (d) Final trace

Figure 2.6: A* algorithm run on an 8×8 grid (4 iteration points shown: start, 1, 2, final trace). The red square represents agent start position, the dark green square represents the goal position, black squares represent obstacles, white squares represent the clear path, light green squares represent the final path chosen by the algorithm. The dark grey squares represent the visited set, and the blue squares represent the priority queue (the darker the blue, the higher the priority)



(a) Iteration 1 (b) Iteration 2 (c) Iteration 3

(d) Iteration 4 (e) Iteration 5 (f) Iteration 6 (g) Iteration 7

Figure 2.7: A* algorithm run on a directed weighted graph. Nodes are labelled with a letter that represents its name (A and G are special nodes agent and goal respectively) and the optimistic heuristic value. Edges have an associated value with them, which represents the actual distance between them. The green node is the goal, dark grey nodes are visited nodes, light blue and dark blue nodes are belonging to the priority queue, dark blue nodes are the next nodes to be expanded, green arrow trace is the optimal solution so far and $d = x$ label is optimal total distance so far

Algorithm 2 A*

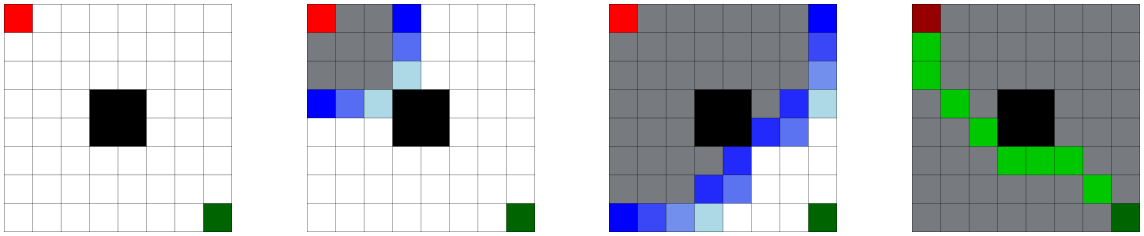
```
1: procedure GET-BACKTRACE(p, M: (A, Os, G))
2:     current ← G
3:     trace ← [current]
4:
5:     while current is not A do
6:         current ← p[current]
7:         add current to trace
8:
9:     return reversed trace
10:
11: procedure A*(M: (A, Os, G))
12:     Initialize priority queue pq with (f(A), A)
13:     visited ← {}
14:     p ← {:}
15:     repeat
16:         current_node ← best n_best from pq where ∀n.f(n_best) ≤ f(n)
17:         add current_node to visited
18:
19:         if current_node is G then
20:             follow Get-Backtrace(p, M)
21:             return
22:
23:         for each neighbour in Neighbours(current_node) do
24:             if neighbour is not in visited or g(current_node) + c(current_node, neighbour) <
    g(neighbour) then
25:                 g(neighbour) ← g(current_node) + c(current_node, neighbour)
26:                 add neighbour to pq
27:                 update p[neighbour] to current_node
28:     until visited is empty
29:
30:     goal was not found
```

2.1.3 Dijkstra

The Dijkstra algorithm [4, 6] (See Figures 2.8, 2.9) is a variation of the A* algorithm where we omit the heuristic function ($f(n) = g(n)$). Therefore the algorithm becomes greedy, meaning that we expand the node with lowest distance from the agent position. (See Figure 2.9). The algorithm is identical to A* (See Algorithm 2), but with $f(n) = g(n)$. Thus, the space and time complexity is the same as A* ($\mathcal{O}(\hat{b}^d)$), but $\hat{b}$ is usually not as efficient as A*.

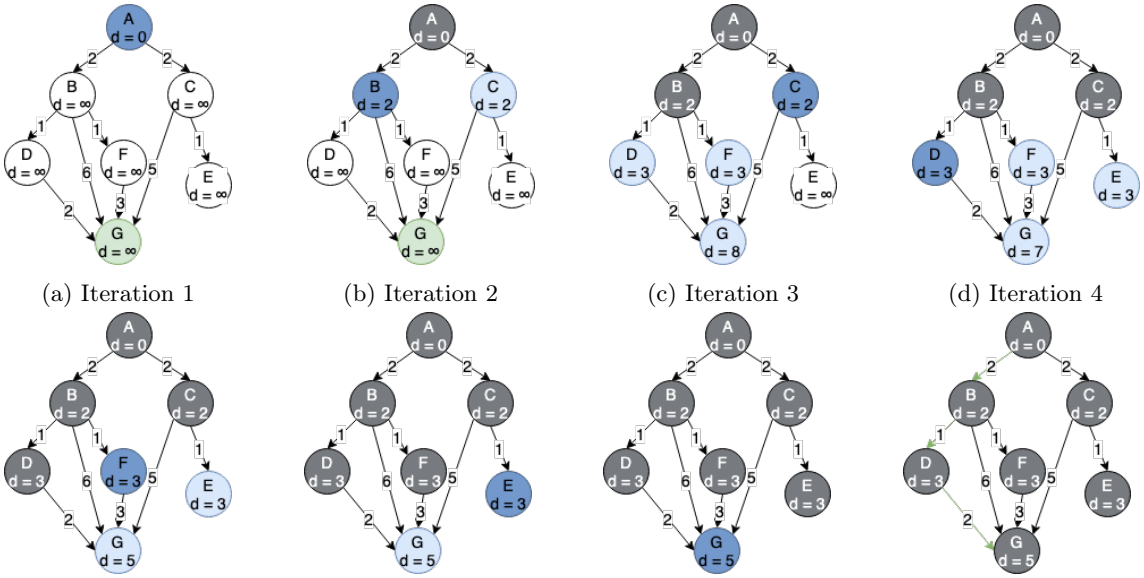
The major difference between A* and Dijkstra is that, because it is a greedy algorithm, once we expand the goal node, we find the final solution. The solution is optimal in terms of minimal distance because after we expand a node, its distance will not be modified in the future (the invariant holds for all nodes, not only for the goal).

The drawback of using this method against A* is that it usually explores more than A* and thus the memory gets quite high.



(a) Starting 8x8 grid (b) Iteration 1 (c) Iteration 2 (d) Final trace

Figure 2.8: Dijkstra algorithm run on an 8×8 grid (4 iteration points shown: start, 1, 2, final trace). The red square represents the agent start position, the dark green square represents the goal position, black squares represent obstacles, white squares represent the clear path, light green squares represent the final path chosen by the algorithm. The dark grey squares represent the visited set, and the blue squares represent the priority queue (the darker the blue, the higher the priority)



(a) Iteration 1 (b) Iteration 2 (c) Iteration 3 (d) Iteration 4

(e) Iteration 5 (f) Iteration 6 (g) Iteration 7 (h) Iteration 8

Figure 2.9: Dijkstra algorithm run on a directed weighted graph. Nodes are labelled with a letter identifier (A and G are special nodes agent and goal respectively) and the current distance. The distance is final when the node is marked as visited. Edges have an associated value with them, which represents edge weight. The green node is the goal, dark grey nodes are visited nodes, light blue and dark blue nodes are belonging to the priority queue, dark blue nodes are the next nodes to be expanded, green arrow trace is the optimal solution

2.1.4 Bug Algorithms

The bug algorithms [4, 24] are one of the earliest and simplest sensor-based solutions for the pathfinding problem, and we will cover two implementations: Bug1 and Bug2. There exist other algorithms which are more advanced such as Tangent Bug described in [4, 24, 25], but we will not cover them as it exceeds the scope of our report.

The idea behind bug algorithms is based on the instinctual behaviour of a bug moving directly towards a destination (goal) and turning around encountered obstacles. The algorithms have two phases: straight line movement (phase 1) and object boundary following (phase 2). We will assume that the agent has a contact sensor that detects if the agent is in the proximity of the boundary of an obstacle.

The bug algorithms are trivial to implement, not computationally expensive, and it has been shown that their success is guaranteed, meaning that they can find a path to the goal if one exists. However, they do not find the optimal path.

2.1.4.1 Bug1

First, we label the direction from the agent position to the goal position with $dir_{AG} = \frac{d(A,G)}{\|d(A,G)\|} = \frac{(G-A)}{\|G-A\|}$. In the first phase, the algorithm follows $dir_{A,G}$ until an obstacle is detected and we mark this position as P_i. Afterwards, it proceeds with the second phase by doing a complete loop around the obstacle while registering the closest distance on the boundary to the goal (find $\hat{Ob} = \arg\min_{Ob} d_{Ob,G}, Ob \in Boundary(O)$) until we reach P_i. After that, we follow the boundary again until we reach $\hat{Ob}$ and return to the first phase. We repeat this process until the goal is found. If the agent were to move, but it can't from $\hat{Ob}$, then we conclude the algorithm as the goal is unreachable (See Figure 2.10, See Algorithm 3) [26].

As it is quite hard to estimate the worst case time complexity, we are going to express it as the number of steps upper bound ($\mathcal{O}(upper_{Bug1})$). The upper bound on the number of steps is given by the total distance from the agent to the goal $d(A,G)$, in the case of no obstacles, plus the worst case traversed boundary length for all obstacles $1.5\sum_{o\in O} length(Boundary(o))$. The space complexity is $\mathcal{O}(1)$ (not counting recursion steps as it can be collapsed in a while loop).

$$upper_{Bug1} = d(A,G) + 1.5\sum_{o\in O} length(Boundary(o))$$

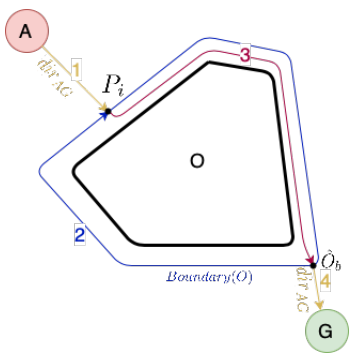


Figure 2.10: Bug1: Arrows represent the movement direction and numbers represent the traversal order. Yellow arrows represent the first phase. Blue and red arrows represent the second phase

Algorithm 3 Bug1

```
1: procedure BUG1(M: (A, Os, G))
2:     Phase1(M)
3:
4: procedure PHASE1(M: (A, Os, G))
5:     move agent towards goal along dir_{A,G} until hits P_i or G
6:     if G was reached then
7:         return
8:     Phase2(M, P_i)
9:
10: procedure PHASE2(M: (A, Os, G), P_i)
11:     find obstacle O with hit point P_i from Os
12:     do a full loop around Boundary(O) while computing Ôb
13:     follow Boundary(O) until Ôb is reached
14:
15:     if can't move from Ôb to G then
16:         goal can't be reached
17:         return
18:
19:     Phase1(M)
```

2.1.4.2 Bug2

The first stage is identical to the one in Bug1, but the second stage follows a greedy approach. Instead of making a full obstacle loop in the second stage, we try to find the point $P \in Boundary(O)$ which belongs to the line segment determined by the original starting agent position and the goal position $P \in LS_{A_{initial},G}$. P should also be chosen in such a way that it is closer than the original point of contact with the obstacle P_i. If $P = P_i$ then the goal can't be reached (See Figure 2.11, See Algorithm 4) [26].

However, this does not imply that Bug2 outperforms Bug1 in all cases. The time complexity (given as the upper bound) for Bug2 is determined by the total distance from the agent to the goal $d(A, G)$, in the case of no obstacles, plus the worst case traversed boundary length for all obstacles $0.5 \sum_{o \in O} n_o \cdot length(Boundary(o))$. The issue here is that, because we follow a greedy approach, we might reencounter the same obstacle, thus n_o represents how many times we have encountered obstacle o. Therefore, Bug1 might yield better performance in cases where the environment contains complex obstacles. The space complexity is still $\mathcal{O}(1)$.

$$upper_{Bug2} = d(A, G) + 0.5 \sum_{o \in O} n_o \cdot length(Boundary(o))$$

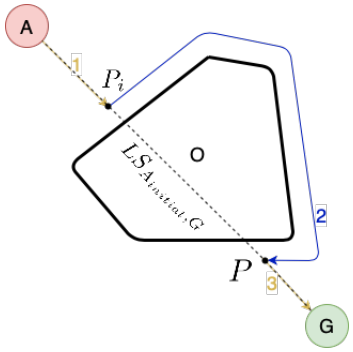


Figure 2.11: Bug2: Arrows represent the movement direction and numbers represent the traversal order. Yellow arrows represent the first phase. Blue arrows represent the second phase

Algorithm 4 Bug2

```
1: procedure BUG2(M: (A, Os, G))
2:     Phase1()
3:
4: procedure PHASE1(M: (A, Os, G))
5:     move agent towards goal along LS_{A_initial,G} until hits P_i or G
6:     if G was reached then
7:         return
8:     Phase2(M, P_i)
9:
10: procedure PHASE2(M: (A, Os, G), P_i)
11:     find obstacle O with hit point P_i from Os
12:     follow Boundary(O) until we hit P ∈ LS_{A_initial,G}, d(P_i, G) ≥ d(P, G)
13:
14:     if P = P_i then
15:         goal can't be reached
16:         return
17:
18:     Phase1(M)
```

2.1.5 Value Iteration on Markovian Decision Processes (MDP)

The Markovian Decision Process (MDP) [11, 12] (See Figure 2.12) represents a state environment in which an agent can transition from one state to the next one until it reaches a terminating state. Whenever the agent chooses an action by moving from a state to another, it is rewarded based on the type of action that it took and the previous and next states.

Formally, a Markovian Decision Process is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}^a_{s,s'}, \mathcal{R}^a_{s,s'}, \gamma \in [0,1], \pi \rangle$. $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space (what actions are available to the agent), $\mathcal{P}^a_{s,s'}$ is the probability transition matrix which gives the probability of transitioning with action a from state s to state s', $\mathcal{R}^a_{s,s'}$ is the reward matrix and states the reward for taking action a from state s to state s', γ is the discounted rewards factor and π is the policy which can be deterministic or stochastic.

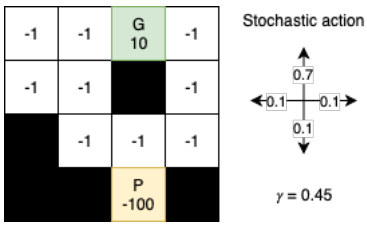


Figure 2.12: An example of an MDP world. White squares represent possible states. The agent can start from **any** of the white squares. There are 2 special terminal states: the green square represents the goal, and the yellow square represents the penalty state. Each state has an associated reward value which is collected whenever the agent leaves the state (for the terminal states, the reward is collected instantaneously). There are 4 possible actions (up, right, down, left; 4-point connectivity), but each action is stochastic meaning that if the agent chooses a specific direction it will move in that direction with 0.7 probability or it will move to the other directions with equal probability

R_t represents the total discounted reward from time step t, and it is defined in terms of the collected rewards r_t.

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

The value function $V^{\pi}(s)$, where π is the policy, with signature $V^{\pi}: \mathcal{S} \to \mathbb{R}$ represents a method for assessing how "good" a state is. It is defined as the expected total discounted reward.

$$V^{\pi}(s) = \mathbb{E}_{\pi}[R_t|S_t = s] = \sum_{a \in \mathcal{A}} \pi(s,a) \sum_{s' \in S} \mathcal{P}^{a}_{s,s'}(\mathcal{R}^{a}_{s,s'} + \gamma V^{\pi}(s'))$$

The state-action value function $Q^{\pi}(s,a)$ with signature $Q^{\pi}: \{\mathcal{S}, \mathcal{A}\} \to \mathbb{R}$ represents a method for assessing how "good" a state is, given a certain action.

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}[R_t|S_t = s, A_t = a] = \sum_{s' \in S} \mathcal{P}^{a}_{s,s'}(\mathcal{R}^{a}_{s,s'} + \gamma V^{\pi}(s'))$$

This also means that we can define $V^{\pi}(s)$ in terms of $Q^{\pi}(s,a)$.

$$V^{\pi}(s) = \sum_{a \in \mathcal{A}} \pi(s,a) Q^{\pi}(s,a)$$

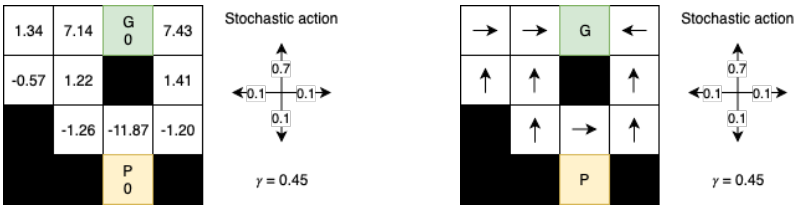
Theorem 2.1: Bellman Optimality Equation

The Bellman Optimality Equation states that the optimal value function is given by the following formula:

$$V^{\pi^*}(s) = \max_a \sum_{s' \in S} \mathcal{P}^{a}_{s,s'}(\mathcal{R}^{a}_{s,s'} + \gamma V^{\pi^*}(s')) = \max_a Q^{\pi^*}(s,a)$$

$$Q^{\pi^*}(s,a) = \sum_{s' \in S} \mathcal{P}^{a}_{s,s'}(\mathcal{R}^{a}_{s,s'} + \gamma V^{\pi^*}(s'))$$

By following the Bellman Optimality Equation (Theorem 2.1), we can devise a dynamic programming algorithm called Value Iteration. The algorithm updates $\mathcal{V}^{\pi}$ in place by applying the Bellman Optimality Equation for each state, thus finding better $\mathcal{V}^{\pi}$ on each run. We repeat the process until we see no further changes in $\mathcal{V}^{\pi}$. This means that $\mathcal{V}^{\pi}$ has converged for all $s \in \mathcal{S}$. After that, we return the optimal policy by choosing the optimal action at each state $s \in \mathcal{S}$ (See Algorithm 5). If we run the algorithm on the MDP world defined in Figure 2.12, we will get the results presented in Figure 2.13.



(a) Optimal value function (b) Optimal policy

Figure 2.13: The results after applying the Value Iteration algorithm on the MDP World defined in figure 2.12. On the left, each cell contains the optimal value function $V^{\pi}(s)$. The optimal policy is displayed on the right

The time complexity is $\mathcal{O}(c|S||\mathcal{A}|)$ and the space complexity is $\mathcal{O}(|S|)$, where c is the convergence rate, $|S|$ is the number of states and $|\mathcal{A}|$ is the number of actions. The time complexity is given by the update rule (which is $\mathcal{O}(|S||\mathcal{A}|)$, for each state we find the maximising action) being run c times (some notebooks state that c is bounded by $|S|$) and the final policy evaluation (which is still $\mathcal{O}(|S||\mathcal{A}|)$). The space complexity is given by the number of elements in $\mathcal{V}^{\pi}$ and π, which is the number of states $|S|$.

Algorithm 5 Value Iteration

procedure VALUE-ITERATION($\mathcal{S}$, $\mathcal{A}$, $\mathcal{P}$, $\mathcal{R}, \gamma$)
    Initialize $\mathcal{V}^\pi$, π arbitrarily (e.g. all 0)

    repeat
        for each $s \in \mathcal{S}$ **do**
            $V^\pi(s) = \max_{a \in \mathcal{A}} \sum_{s' \in S} \mathcal{P}^a_{s,s'}(\mathcal{R}^a_{s,s'} + \gamma V^\pi(s'))$
    until we have no change in $\mathcal{V}^\pi$

    for each $s \in \mathcal{S}$ **do**
        $\pi(s) = \arg\max_{a \in \mathcal{A}} \sum_{s' \in S} \mathcal{P}^a_{s,s'}(\mathcal{R}^a_{s,s'} + \gamma V^\pi(s'))$
    return π

The advantage of using MDPs is that we can model a more complex world for the agent in which we can define areas which should be avoided (such as bridges, because they draw more battery power) by associating a negative reward with them. However, it can also be considered a disadvantage for worlds that cannot be easily modelled, as we have to know the transition probabilities and rewards apriori before we can apply Value Iteration. Not to mention that the number of states can get quite big. For instance, let us assume that we have a robotic arm with 3 joints, and each join can rotate 180°. If we discretise each angle into 1°angles, we will have a total number of $180^3 \simeq 5.8$ million states. There exist algorithms which can handle a large number of states such as Monte Carlo (sampling episode traces) and Temporal Difference Learning (combines dynamic programming and sampling), but we will not cover them as they are out of the report's scope [11].

2.2 Sampling Based Planners

A significant drawback of using resolution complete (the algorithm is guaranteed to find a path) and resolution optimal (the algorithm will find the shortest path if one is available) graph search planners such as A*, is that they are only suitable for small problem sizes [27].

The algorithms described in this chapter employ a sampling technique to explore the unknown environment rapidly, thus scaling well with large problem sizes.

2.2.1 Rapidly-exploring Random Tree (RRT)

The Rapidly-exploring Random Tree (RRT) [8, 9, 3, 10] is a randomised data structure solution to the pathfinding problem which shares some desirable properties with probabilistic road-maps (PRM) [4], but, unlike PRMs, it is able to handle problems that have non-holonomic constraints (a holonomic robot is a robot which is able to move instantaneously in all available degrees of freedom).

The algorithm starts by creating a graph G with a single vertex containing the agent position. Then, we proceed to incrementally add new vertices until we get to the goal region (the region in which we can assume that we have reached the goal or will be able to safely reach it). At each iteration, we sample a new point on the map which represents a valid position (not intersecting any obstacle, or out of bounds) $x_{rand} \notin Os$. We find the nearest vertex x_{near} from x_{rand} by searching the graph (e.g. graph pruning). We choose the next vertex to be inserted x_{new} based on a predefined max_dist variable. If we are too far from x_{near} ($d(x_{near}, x_{rand}) > max\_dist$) we choose a point on the line segment defined by x_{near} and x_{rand} that is max_dist far away from x_{near}. If we are too close to x_{near} ($d(x_{near}, x_{rand}) \leq max\_dist$) we simply choose x_{rand} (See Figure 2.14). After we have chosen x_{new}, we make sure that the path from x_{near} to x_{new} is

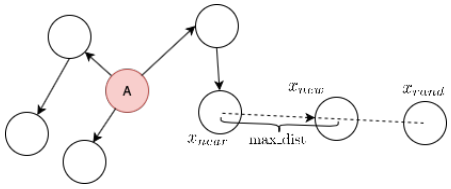
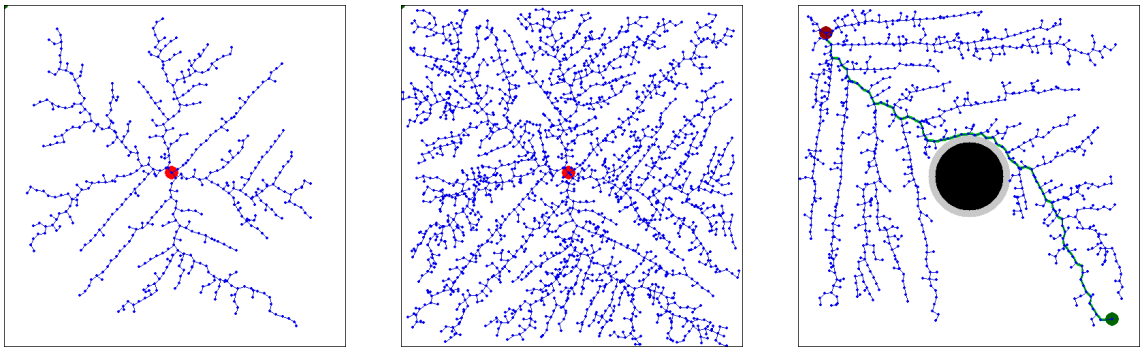


Figure 2.14: RRT x_{new} decision algorithm, if x_{rand} is too far from x_{near} we interpolate x_{new} between the line segment defined by x_{near} and x_{rand} such that x_{new} is max_dist away from x_{near}



(a) 500 iterations (b) 2000 iterations (c) Full RRT

Figure 2.15: RRT run with 500 iterations (left) and with 2000 iterations (middle) with max_dist 10. The whole algorithm can be seen in the right figure. Blue dots represent vertices and blue lines represent edges

collision free, and if it is we add x_{new} as a new vertex to the graph G and (x_{near}, x_{new}) as a new edge (See Figure 2.15, See Algorithm 6).

It is worth mentioning that there exist some variations which use a dynamic max_dist, usually picking a random number from the interval $[0, max)$ at each new iteration. Furthermore, the algorithm can be extended to different metric spaces by changing the distance function d.

Algorithm 6 Rapidly-exploring Random Tree (RRT)

```
1:  procedure RRT(M: (A, Os, G))
2:      max_dist ← maximum distance allowed between tree children and sample
3:      G ← initalize graph with A as vertex
4:      while True do
5:          x_rand ← Sample()
6:          x_near ← GetNearestVertex(x_rand)
7:          x_new ← GetNewVertex(x_near, x_rand, max_dist)
8:
9:          if CollisionFree(x_near, x_new) then
10:             add x_new as a new vertex to G
11:             add (x_near, x_new) as a new edge to G
12:
13:         if x_new is in G region then
14:             follow trace from root G to x_new
15:             return
```

Space complexity is $\mathcal{O}(|V| + |E|)$, where $|V|$ is the number of nodes and $|E|$ is the number of edges in graph G. Time complexity is given by $\mathcal{O}(i(\mathcal{O}(Sample) + \mathcal{O}(GetNearestVertex) + \mathcal{O}(GetNewVertex) + \mathcal{O}(CollisionFree)))$, where i is the number of iterations and $\mathcal{O}(x)$ is the time complexity of method x. i can be bounded by the number of nodes from graph G ($\mathcal{O}(|V|)$) (at each step we add a new node to the graph). $\mathcal{O}(Sample)$ depends on the choice of sampling method (we use uniform random sampling which is $\mathcal{O}(1)$). $\mathcal{O}(GetNearestVertex)$ has DFS time and space complexity which is $\mathcal{O}(b^d)$ and $\mathcal{O}(bd)$, where b is branching factor and d is depth (if pruning is

used b becomes $\hat{b}$). $\mathcal{O}(GetNewVertex)$ is $\mathcal{O}(1)$ as it is a simple logical decision. $\mathcal{O}(CollisionFree)$ depends on the collision detection system. There are collision systems that have $\mathcal{O}(n)$ time complexity (for all entities). Because we need to check the path between x_{near} and x_{new} we still have $\mathcal{O}(n)$ time complexity. Therefore, the final time complexity is given by $\mathcal{O}(|V|(\mathcal{O}(b^d) + \mathcal{O}(n)))$.

A major advantage of using this method is that it is quite fast and memory efficient since it is run on a subset of the grid (the samples). Moreover, the algorithm can be coupled with the algorithms described in the Section 2.3 (Interpolating Curve Planners) by transforming the graph edges into curves, which offers support for non-holonomic robots.

A major disadvantage is that, although the algorithm is probabilistic complete (as more samples are drawn the more likely is to find a path to the goal), it does not find an optimal path and the solution is usually quite jerky.

To overcome this issues, we introduce the RRT* [10, 27] algorithm which has been proven to be asymptotic optimal (as the number of iterations gets larger, the more we approach to the optimal solution). RRT* does this by incrementally modifying the structure of the graph. When a new node is affixed to the graph, the algorithm might choose to rewire the connections in the graph by considering the new node as a replacement parent for the other existing nodes, if the resulting change yields a better solution.

A major drawback of using RRT* is that, in order to achieve an optimal solution, the number of iterations has to be quite large and therefore, it becomes quickly expensive in higher dimensions. The Informed RRT* [10] algorithm attempts to solve this issue by adopting an ellipsoid heuristic approach. The algorithm shares the same logic as RRT* and it only improves the performance of finding an optimal path once a solution is found. This is achieved by sampling from the ellipsoidal heuristic. Thus, the number of iterations is reduced, and the optimal search is focused on a smaller region.

2.3 Interpolating Curve Planners

The algorithms that lie into this section are used in the local planning part of the pathfinding problem. They attempt to find a trajectory that fits a given global description of the path (such as way-points) by taking into account multiple parameters such as feasibility, comfort, vehicle dynamics and efficiency. Interpolation is used to increase the number of data points between the way-points in order to smooth out the trajectory and create easily traversable paths for non-holonomic robots [2].

Lines and Circles. Primitives such as lines and circles can be used to describe the local path between two way-points. Figure 2.16(a) represents the shortest path to execute a 3-step 180°turn for a car. Because it can only fit geometric primitives, the algorithm is simple to implement and fast, but it is also quite limited (preferential parameters such as curvature angle and continuation are completely ignored).

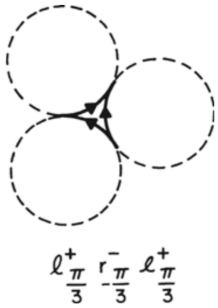
Clothoid Curves. The implementation of this algorithm is based on Fresnel integrals, and the resulting curves offer a smoother transition between different curvatures (such as between a straight line and a curve) than Lines and Circles (See Figure 2.16(b)). The algorithm accounts for multiple constraints such as dynamic and physical vehicle limitations (e.g. steering wheel), thus making it more robust for non-holonomic robots such as a car. Moreover, the algorithm was used in the design of highways and railways.

Polynomial Curves. This type of curves are mainly used to satisfy different preferential parameters such as angle and curvature when drawing the trajectory between two way-points. The main advantage of using this method is that the preferential parameters determine the coefficients of the polynomial, and thus, it is much more flexible. Figure 2.16(c) represents an example of using polynomial curves to change lanes.

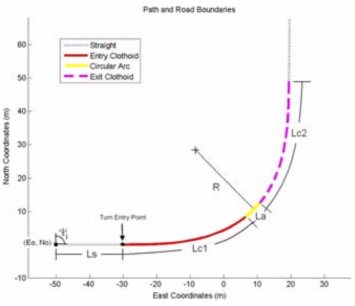
Bézier Curves. This algorithm produces curves based on control points. The control points placement defines the curvature at the beginning and the end of the curve. The significant advantage of this algorithm is that it has a low computational cost since the shape of the curve is defined by the control points. Therefore, the algorithm has been extensively used in different drawing software applications, technical drawing (they can also be drawn by hand) and trajectory design. Moreover, the curve can be used to approximate clothoid curves. Figure 2.16(d) represents an example of using 3rd and 4th degree Béziers to find the best curvature estimate based on the current situation.

Spline Curves. The spline curve is sub-divided into multiple parametric patches that can be defined as polynomial curves, clothoid curves and b-splines (Bézier curves). Splines have a high degree of smoothness at each patch joint and can be extended into higher dimensions. Figure 2.16(e) represents an example of a b-spline with a changing knot (the junction between two sub-segments).

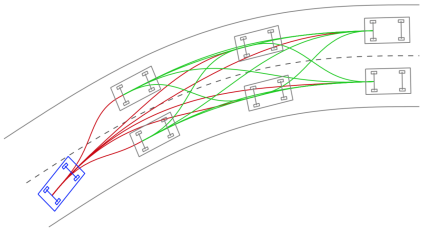
It is worth mentioning that the global description of the path demands to include the collision-free areas in addition to the way-points or provide a collision detection system to be used when drawing the curves to ensure that no collisions are encountered while performing any of the above algorithms.



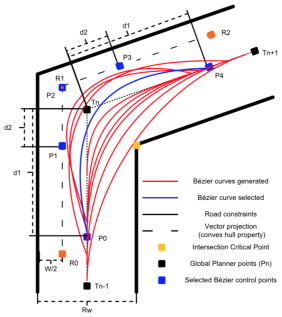
(a) Lines and Circles [2]



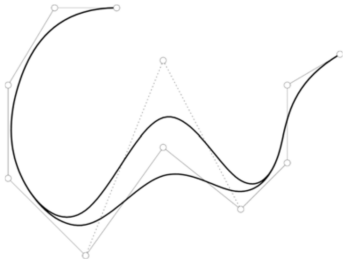
(b) Clothoid [2]



(c) Polynomial [2]



(d) Bézier [2]



(e) Spline [2]

Figure 2.16: Examples of the most important interpolating curve planners

2.4 Numerical Optimization Approaches

This category of algorithms describes the path planning problem as a cost function, which is then minimised by using function approximation techniques and machine learning methods.

2.4.1 Potential Field Method

The main concept of the Potential Field Method (PFM) [2, 28, 29] algorithm is to fill the map with a potential field f in which the robot is attracted towards the goal and repulsed from obstacles. The potential field f is composed of two functions: $f = a + r$ where a is the attraction function and r is the repulsion function. After the potential function is computed, an optimisation technique of finding the minimum of f, such as gradient descent, is applied (See Figure 2.17). The Wave-front Planner 2.1.1 is an example of a potential function $f = a$ where the repulsion function is missing.

The space complexity is $\mathcal{O}(nm)$ where n is the width and m is the height of the map. The time complexity is $\mathcal{O}(\mathcal{O}(f) + \mathcal{O}(gradient\_descent))$. $\mathcal{O}(f) = \mathcal{O}(\mathcal{O}(a) + \mathcal{O}(r))$ depends on the choice of a and r. Assuming same a as in the Wave-front Planner, $\mathcal{O}(a) = \mathcal{O}(b^d)$, where b is the branching factor and d is the depth. Assuming that we use a simple r which inflates the obstacle, the time complexity of $\mathcal{O}(r)$ is $\mathcal{O}(xo)$, where x is the inflation rate and o is the average obstacle size. $\mathcal{O}(gradient\_descent)$ is more involved if we use real gradient descent optimisation as it is based on the convergence rate. If we were to use the same gradient descent method from the Wave-front Planner, then $\mathcal{O}(gradient\_descent) = \mathcal{O}(d)$, where d is the depth of the solution. The final time complexity is $\mathcal{O}(b^d + xo + d)$.

This solution is appealing because it is mathematically elegant and simple. It behaves well on complex environments which contain narrow passages, and it keeps a safe distance from the obstacles based on different factors such as relative velocity. Moreover, the algorithm can be extended to highly dynamic environments in which the obstacles and the goal are continually moving. Lastly, we can use vehicle parameters such as velocity and torque, to define a and r. Thus, we can avoid collisions based on the velocity of the vehicle (i.e. when the vehicle has a high enough velocity to not being able to exit a collision trajectory) [28].

One of the major drawbacks of using this method is that depending on the choice of f, local minimum problems might arise. Usually, these problems require particular solutions and rigorous analysis to ensure that there is no situation in which the agent will be trapped forever.

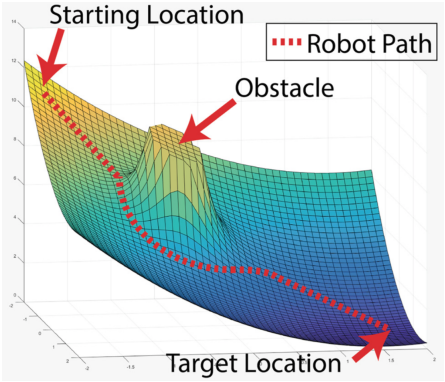


Figure 2.17: Potential field $f = a + r$, where a is the attraction function and r is the repulsion function [30]

2.4.2 LSTM

Currently, the most important works that have considered LSTMs in this field are [15, 16, 1].

Nicola et al. attempt to use an LSTM network to create an online global path planner by generating a low-resolution high-level path from start to goal [15]. The network was trained on randomly filled generated mazes with different corridor sizes with A* 2.1.2 as ground truth and tested on maps from the popular game Dragon Age. The primary motivation of using the proposed method from [15] is that, unlike A* which is offline (directly produces the final output), this method is an online search strategy that takes into account the fact that the environment is usually unknown or too expensive to store. Thus, at each time step, the network is queried for the next action that it should take given the local information of the map and previous choice of action. One of the significant drawbacks is that the rate of success of the algorithm gets lower as the map becomes more complicated (longer corridors, more complex objects).

Lee et al. make use of the LSTM architecture to extend the state of the art Value Iteration Networks (VIN) [16]. VINs are similar to the Value Iteration algorithm described in applied on a 2D grid world, but without making use of pre-defined MDP parameters such as the transition probabilities and rewards. The paper has pointed out that, despite being so successful, there are still issues with the current design of VIN such as the design of the transition model and the large number of recursive calls taken to achieve a reasonable estimate of the optimal path to the goal. The proposed solution replaces the recurrent VIN update rule with an LSTM kernel. [16] has shown that the performance of the new solution has better or at least as good performance as the original VIN.

Inoue et al. aim to use a Convolutional Auto-encoder (CAE) combined with an LSTM to plan a path under a changing environment, by disregarding the unknown environment constraint (the map is fully discovered) [1]. The two network sections are trained individually. Firstly, the CAE is trained on randomly generated maps with obstacles of different sizes. After that, the decoder is discarded, and the encoder is used to encode the same maps in order to extract the main features. Secondly, the LSTM is trained on the encoded maps by using RRT as ground truth. Unlike [15], the solution is offline as it produces a full path (which might collide with obstacles), but it can be easily converted to an online solution in order to handle collision detection and prevention. A significant difference between this solution and [15] is that it has a higher overall success rate, which might arise since the CAE learns the input features as opposed to being hand crafted as in [15].

Chapter 3

Background

Machine Learning (ML) is a programming approach for creating Artificial Intelligence (AI) systems in non-trivial environments by making use of experiences and a substantial amount of data [31, 14, 32, 33].

One of the most trivial Machine Learning examples is classifying handwritten digits by making use of the MNIST database [34]. It is conspicuous that Machine Learning is the only viable option in developing this kind of applications as it is tough (if not impossible) to come up with hard-coded rules for each class case [14].

Notation. We will follow the same mathematic conventions used in [14] (See Table 3.1).

x	A scalar
$\boldsymbol{x}$	A vector
$\boldsymbol{M}$	A matrix
$\boldsymbol{M}^T$	A transposed matrix
x_i	Element i from vector $\boldsymbol{x}$, starting from 0
$\boldsymbol{X_i}/\boldsymbol{x_i}$	Row i of matrix $\boldsymbol{X}$
$X_{i,j}$	The element with row i and column j from matrix $\boldsymbol{X}$
$\boldsymbol{x} * \boldsymbol{y}$	An element-wise vector product
$x^{(l)}/\boldsymbol{x^{(l)}}/\boldsymbol{X^{(l)}}$	Depends on the context of l

Table 3.1: Notations [14]

The aim of Machine Learning methods is to find a pattern in the features of data $\boldsymbol{X} \in \mathbb{R}^{n \times m}$, where n is the number of data examples and m is the number of features. Machine Learning can be classified into two major categories: supervised and unsupervised, but we are going to focus on supervised ML. Supervised ML operates on data that has been labelled with $\boldsymbol{y} \in \mathbb{R}^n$, where each y_i corresponds to a row in $\boldsymbol{X}$, $f(\boldsymbol{x_i}) = y_i$, $f \colon \mathbb{R}^m \to \mathbb{R}$. Our goal is to find a function $h \colon \mathbb{R}^m \to \mathbb{R}$ which approximates f, $h(\boldsymbol{x_i}) \simeq f(\boldsymbol{x_i})$.

3.1 Neural Networks

(Feed-forward) Neural Networks (NN) [31, 14, 32, 33] are a type of supervised Machine Learning inspired by real-world biological neurons. Each NN architecture is composed of artificial neurons.

3.1.1 Artifical Neuron

An artificial neuron [31, 32, 14] (See Figure 3.1) has $m+1$ inputs $x_{i\in\{0,\dots,m\}}$ and $m+1$ weights $w_{i\in\{0,\dots,m\}}$, where input 0 is a bias (we set $x_0 = 1$ so that w_0 becomes bias), with one output y. The neuron can be "activated" using an activation function f such as sigmoid ($\sigma(x) = \frac{1}{1+e^x}$), tanh ($tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$) or rectified linear unit (ReLU) ($f(x) = \begin{cases} 0, x < 0 \\ x, x \geq 0 \end{cases}$). The big picture is that the weights help the neuron learn a combination of inputs while the bias behaves like a learnable threshold. The idea behind the activation function is that we can define if a neuron is active or non-active based on the output value of f (e.g. for sigmoid $f\colon \mathbb{R} \to [0,1]$, if $f < 0.5$ neuron is non-active).

$$y = f(z) = f(\sum_{i=0}^{m} x_i \cdot w_i) = f(\boldsymbol{x}^T \boldsymbol{w}), x_0 = 1 \text{ and } w_0 \text{ bias}$$

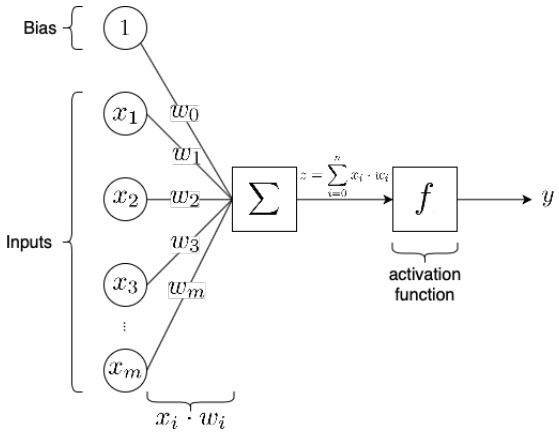


Figure 3.1: Artificial neuron representation

3.1.2 Neural Network Architecture

We can then combine multiple artificial neurons to create a layered architecture (See Figure 3.2). There are 3 types of layers: the input layer, the output layer and the hidden layer. The input and output layers are mandatory and should match the number of neurons with the size of the input features and respectively output. The number of hidden layers, as well as, their number of neurons are part of the NN architecture, and they are referred to as the depth and respectively the width of the network [31, 32, 14].

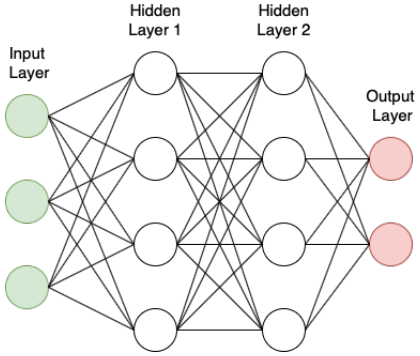


Figure 3.2: Neural network architecture with 2 hidden layers

3.1.3 Forward-propagation

Each layer can be then written as $\boldsymbol{a^{(l)}} = f(\boldsymbol{z^{(l)}}) = f(\boldsymbol{W^{(l)}}^T \boldsymbol{a^{(l-1)}})$, where $\boldsymbol{a^{(l-1)}} \in \mathbb{R}^{m+1}$ is the input to layer l, $\boldsymbol{a^{(l)}} \in \mathbb{R}^d$ is the output of layer l, $\boldsymbol{W^{(l)}} \in \mathbb{R}^{(m+1)\times d}$ are the weights associated with layer l, $a_0^{(l-1)} = 1$ and $\forall i. W_{i,0}^{(l)}$ is bias. Therefore, each row r in $\boldsymbol{W^{(l)}}$ corresponds to the weights associated with the r neuron in layer l. $l = 0$ corresponds to the input layer and $l = L$ is the output layer. Now, we can write the whole architecture as following:

$$
\begin{aligned}
\boldsymbol{y} &= \boldsymbol{a_L} \\
\boldsymbol{a_0} &= \boldsymbol{x} \\
\boldsymbol{a^{(l)}} &= f(\boldsymbol{W^{(l)}}^T \boldsymbol{a^{(l-1)}}) \\
\boldsymbol{y} &= f(\boldsymbol{W^{(L)}}^T f(\boldsymbol{W^{(L-1)}}^T f(...f(\boldsymbol{W^{(0)}}^T \boldsymbol{x})))
\end{aligned}
$$

For now we can only (forward) propagate our input through the network and receive a random output, but we do not learn from our data $\boldsymbol{x}$ [31, 32, 14].

3.1.4 Back-propagation

Back-propagation [31, 32, 14] is a method in which we perform a backward pass through the network in order to adjust the weights of the neurons. The changing amount depends on how well we currently approximate the output against $\boldsymbol{y}$. If we have a bad approximation we adapt the weights in order to get closer to the true output.

We can develop a cost function J that uses a loss function L which will tell us how close are we to the actual solution. As an example, the least square error is a loss function used in regression $L(\boldsymbol{y^{(l)}}, \boldsymbol{a^{(l)}}) = \frac{1}{n}\sum_{i=0}^{n}(y_i^{(l)} - a_i^{(l)})^2$, $J = L$. Our objective is to minimise the cost function J, as we would like to be as close as possible to the actual solution, while updating the weights (learn from our mistakes).

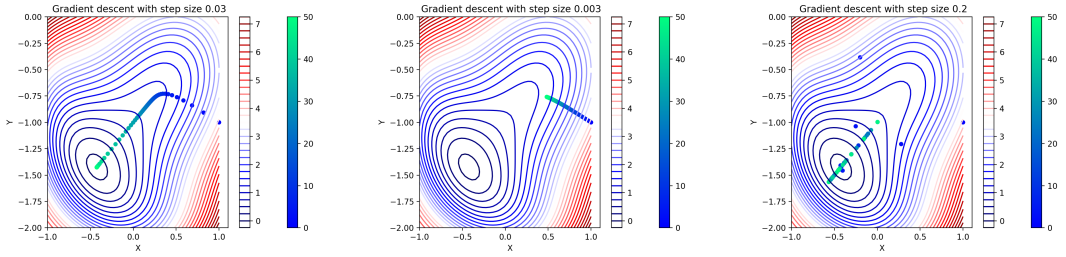
In order to update the weights correctly, we are going to use gradient descent (See Figure 3.3), which is an iterative first-order method for finding a local minimum (or maximum as both problems are equivalent) or global minimum if the problem is convex. Therefore the update rule for our weights becomes:

$$
\boldsymbol{W^{(l)}} \leftarrow \boldsymbol{W^{(l)}} - \alpha \frac{\partial J^{(l)}}{\partial \boldsymbol{W^{(l)}}}
$$

for each layer l, where α is the learning rate. The weights are updated starting from the output layer $l = L$ towards the input layer $l = 0$ as when we update the layer $l = k$, we need the output of layer $l = k + 1$ in order to compute $J^{(l)}$.

3.1.5 Learning Rate

The learning rate [31] α controls how much we change the weights. We have to be careful when setting the learning rate as if it is too low we might not reach the minimum (if we do not have enough iterations) or if it is too high, we are going to oscillate and might not find the minimum or even diverge (See Figure: 3.3).



(a) Learning rate is just right (b) Learning rate is too small (c) Learning rate is too high

Figure 3.3: Gradient descent with a reasonable, too small or too high learning rate. When the learning rate is reasonable we reach a minimum point, when the learning rate is too small we do not reach a minimum if the number of iterations is not high enough, and when the learning rate is too high we start to oscillate and might even diverge

There are techniques that deal with this issue such as momentum or exact linear search optimisations for the learning rate: Golden Section or Newton-Raphson methods [35]:

$$\alpha^{(j)} \in \arg\min_{\alpha \geq 0} \boldsymbol{W^{(l)}} - \alpha^{(j)} \frac{\partial J^{(l)}}{\partial \boldsymbol{W^{(l)}}}, \text{ for each iteration j}$$

3.1.6 Training

Putting it altogether, in order to train the Neural Network, for each training example $\boldsymbol{x_i}$ we:

- execute a forward propagation to get $\forall l \in \{0, ..., L\}.\boldsymbol{a^{(l)}}$
- back-propagate to update the weights $\forall l \in \{0, ..., L\}.\boldsymbol{W^{(l)}}$ following the gradient descent formula

However, before training the model, we have to process the data so that we have a mechanism of improving (by tuning the hyper-parameters) and evaluating the model. There are multiple ways of splitting the data, but we are going to focus on the most important ones: holdout and k-fold cross validation [31, 32, 14].

The holdout method is mostly used when there is a lot of training data. The training data is split into three subsets: training, validation and testing (usually 60% training, 20% validation, 20% testing). The model is then trained on the training data and evaluated on the validation data in order to improve the performance of the network by tuning the model hyper-parameters. The process of passing the entire training and validation data through the model is known as an epoch. The process can then be repeated n epochs until no further improvement is noticed or over-fitting occurs. When the model is tuned, we run a final evaluation on the testing data to report the final results of our model. We do not tune the model on the testing data as this would contradict our generalisation assumption and will make the evaluation redundant.

In the case of having low amounts of training data, we can opt for the k-fold cross validation method. The idea behind this method is that we are going to partition our data set into two groups: training and testing k times. The partitioning is achieved as seen in Figure 3.4. The training process is similar to the holdout method, but the evaluation results are averaged over the splits. It is worth mentioning that there are many variants to this method such as: leave n out, leave one out, k-fold cross validation with validation and test set and many more.

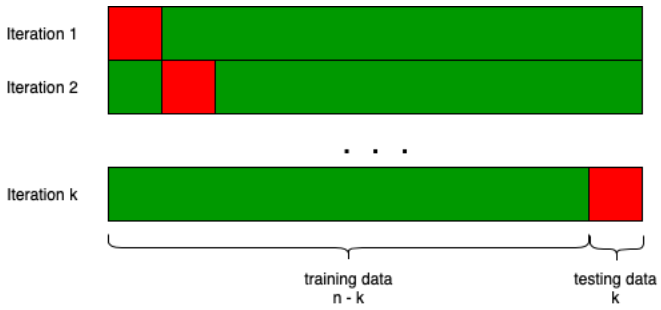


Figure 3.4: k-fold cross validation. Training data is represented with green and testing data is represented with red

3.1.7 Evaluation

At each evaluation step in our training pipeline, we can report a series of useful statistical measures such as loss over model complexity, precision, accuracy, recall, F_1 measure, confusion matrices and others [31, 32, 14].

3.1.8 Over-fitting

Over-fitting [31, 32, 14] occurs when the model's predictions are almost perfectly matching the training data targets, meaning that it cannot generalise for unseen data. The opposite phenomenon is under-fitting and happens when the model is under-trained and cannot make any useful predictions. Formally, when the model under-performs, it has high bias, and when the model over-performs, it has high variance (See Figure 3.5).

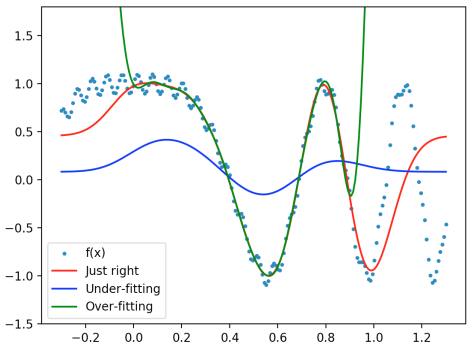


Figure 3.5: Over-fitting example. Blue dots represent the actual function we are trying to approximate, the red function is just right, the green function is over-fitting, the blue function is under-fitting. The model was trained on the interval $[0, 0.9]$

Over-fitting starts at the point where the cost function starts increasing on the testing data, even though it keeps decreasing on the training data (See Figure 3.6).

3.1.9 Regularisation

To overcome the over-fitting problem we can use different approaches such as: early stopping (manually stop the training process when over-fitting is spotted), dropout (assign a probability that a neuron will "fire"), adding noise to data, L-norm regularisation (add a penalty term to the loss function; e.g. L1, L2, max norm) [31, 32, 14].

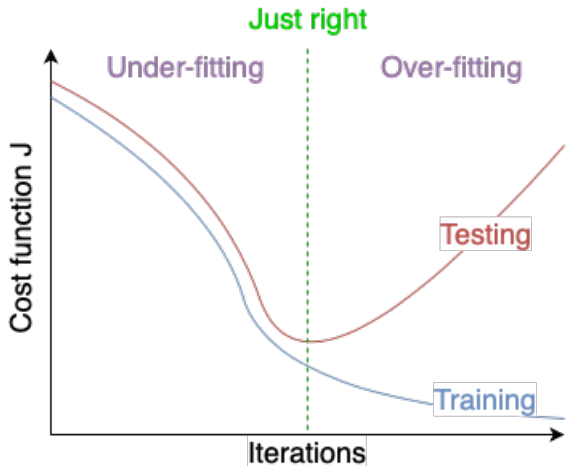


Figure 3.6: Over-fitting threshold

L1 regularisation is used to "get rid" of weights by effectively setting them to 0. Intuitively, this means that we are going to only keep meaningful features.

$$J(\boldsymbol{W}) = L(\boldsymbol{y}, \boldsymbol{a}) + \lambda \sum_{w} |w|, \text{ where } \lambda \text{ is the regularisation factor}$$

L2 regularisation is used to "punish" large weights, thus favouring small weights. This means that our h will not oscillate that much due to high weights. It should be noted that L2 is more sensible to outliers than L1.

$$J(\boldsymbol{W}) = L(\boldsymbol{y}, \boldsymbol{a}) + \lambda \sum_{w} w^2, \text{ where } \lambda \text{ is the regularisation factor}$$

3.2 Recurrent Neural Networks

Neural Networks are great for solving regression, logistic and classifying problems, but they do not handle well time-series problems. One of the reasons why is that time-series problems depend on the previous state and due to the acyclic graph design of the NN architecture, this information cannot be propagated through the network, and it is lost.

Recurrent Neural Networks (RNN) [31, 36] are tackling this sort of issue by introducing a backwards edge from the last hidden layer to the first hidden layer in the NN architecture that carries the previous state information. Thus, if we unfold the RNN architecture, we can notice that each previous outcome is connected to the current one (See Figure 3.7).

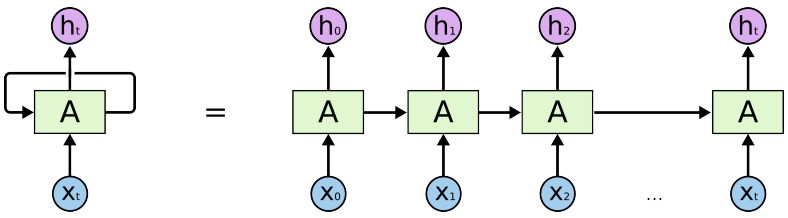


Figure 3.7: Recurrent Neural Network [36]

One of the major issues with RNNs is that they do not train well. They use back-propagation through time, which is a similar training process to the NN one, but it achieved over the length

of the time series. This procedure suffers from the vanishing or exploding gradient problem. The vanishing gradient problem states that if the gradient of the weights is less than 1 at time t, than as $t \to 0$ so is $\nabla w^{(t)} \to 0$. In the case of the weights being matrices, we use the L2 norm $\left\|\nabla \boldsymbol{W}^{(t)}\right\| = \sqrt{\lambda}$, where λ is the largest eigenvalue of $\boldsymbol{W}^{(t)}$. The exploding gradient problem states that if the gradient of the weights is greater than 1 at time t, than as $t \to 0$, $\nabla w^{(t)} \to \infty$ (analogue for $\boldsymbol{W}^{(t)}$). Therefore, only the first layers from the back are trained properly if the time-series length is even remotely big. A common solution (but not preferred) to this problem is to change the architecture by adding extra backward edges between hidden layers so that we can propagate the gradient more efficiently.

3.3 Long Short-Term Memory (LSTM)

The Long Short-Term Memory (LSTM) [13, 36] is a type of Recurrent Neural Network that solves the vanishing and exploding gradient problem by adding internal gates to the architecture to control the gradient flow during training.

As seen in Figure 3.8, the LSTM cell contains four types of internal gates that control the flow of information, and three types of input: $\boldsymbol{x}^{(t)}$ (the actual input at time t), $\boldsymbol{c}^{(t)}$ (cell state at time t, which we do not expose outside the internal architecture) and $\boldsymbol{h}^{(t)}$ (hidden state at time t).

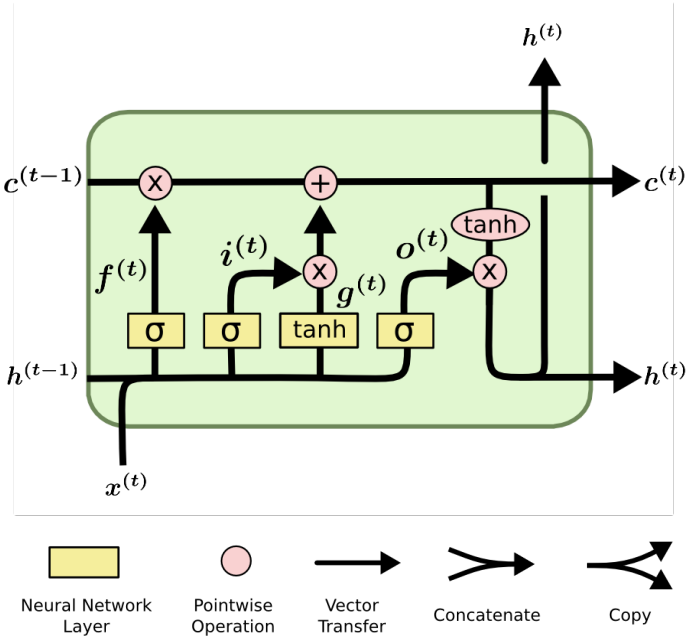


Figure 3.8: LSTM cell [36]

The **forget** gate $\boldsymbol{f}^{(t)} = \sigma(\boldsymbol{W}^{(f)} \cdot [\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}] + \boldsymbol{b}^{(f)})$, where σ is the sigmoid function, $\boldsymbol{W}^{(f)}$ are the weights and $\boldsymbol{b}^{(f)}$ is the bias associated with the **forget** gate, controls the amount of information that we would like to forget as $\boldsymbol{f}^{(t)} \colon \mathbb{R}^n \to [0, 1]$, so when $\boldsymbol{f}^{(t)}$ tends to 0 we forget more about the previous states.

The **input** gate $\boldsymbol{i}^{(t)} = \sigma(\boldsymbol{W}^{(i)} \cdot [\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}] + \boldsymbol{b}^{(i)})$ controls how much information we would like to keep from the input. As in the case of the **forget** gate, when $\boldsymbol{i}^{(t)}$ tends to 0 we discard more information.

The **gate** gate $\boldsymbol{g}^{(t)} = tanh(\boldsymbol{W}^{(g)} \cdot [\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}] + \boldsymbol{b}^{(g)}$ decides the importance of the data. Because $\boldsymbol{g}^{(t)} \colon \mathbb{R}^n \to [-1, 1]$, if $\boldsymbol{g}^{(t)}$ tends to 1 we consider the input to be more important.

The next cell state is given by $\boldsymbol{c}^{(t)} = \boldsymbol{f}^{(t)} * \boldsymbol{c}^{(t-1)} + \boldsymbol{i}^{(t)} * \boldsymbol{g}^{(t)}$. The cell state combines the output

of all internal gates and contains the information that should be carried to the next state.

The **output** gate $\boldsymbol{o^{(t)}} = \sigma(\boldsymbol{W^{(o)}} \cdot [\boldsymbol{h^{(t-1)}}, \boldsymbol{x^{(t)}}] + \boldsymbol{b^{(o)}})$ controls how much information we show to the user. As in the case of the **forget** and **input** gates, if $\boldsymbol{o^{(t)}}$ tends to 0 we output less information.

The actual output is given by the hidden state $\boldsymbol{h^{(t)}} = \boldsymbol{o^{(t)}} * tanh(\boldsymbol{c^{(t)}})$.

The particular design of this architecture solves the vanishing and exploding gradient problems by making use of the internal cell state $\boldsymbol{c^{(t)}}$. During training, the gradients flow more efficiently along the line defined between $\boldsymbol{c^{(t-1)}}$ and $\boldsymbol{c^{(t)}}$. Moreover, it can be further extended by adding more LSTM layers by connecting the output at each time step from layer $n-1$ to the input of layer n to create a stacked architecture that has a similar effect to a Deep Neural Network.

It should be mentioned that there exist various LSTM architectures and extensions that attempt to improve the overall performance of the network such as: Gated Recurrent Units (GRUs), LSTMs with peephole connections and many more. Some of the extended architectures have reduced computational cost due to a lower complexity design (e.g. the GRU has a lower number of internal gates compared to the LSTM architecture), but, they still achieve similar or insignificantly better prediction performance than LSTMs [31, 32, 14, 13, 36].

Chapter 4

PathBench

PathBench is a motion planning platform used to develop, assess, compare and visualise the performance and behaviour of the discussed algorithms. The platform is split into four main components: **Simulator**, **Generator**, **Trainer** and **Analyzer** joined by the infrastructure section. Additionally, the platform provides a *ROS* real-time extension for interacting with a real-world robot through PathBench (See Figure 4.1). The full architecture can be seen in Figure 4.2.

Notation. We will use *italic font* for libraries, **bold font** for classes and `typewriter font` for code snippets/file names/functions/variables. For types, we use the *python* type hinting system (e.g. a list of integers is defined as `List[int]`).

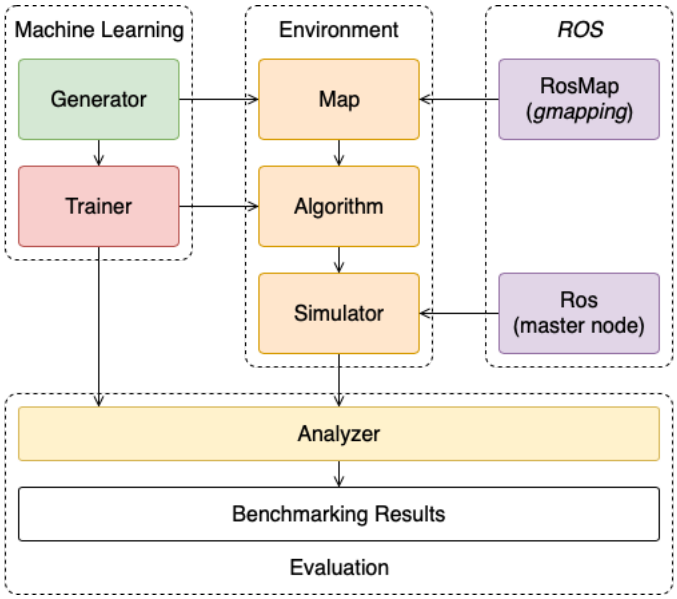


Figure 4.1: PathBench structure high-overview. Arrows represent information flow/usage ($A \xleftarrow{gets/uses} B$). The Machine Learning section is responsible for training dataset generation and model training. The Environment section controls the interaction between the agent and the map, and supplies graphical visualisation. The *ROS* section provides support for real-time interaction with a real physical robot. The Evaluation section provides benchmarking methods for algorithm assessment

Infrastructure. This component is responsible for linking all other components and provide general service libraries and utilities (for a comprehensive explanation of the infrastructure subcomponents, please refer to Appendix A.1).

Simulator. This section is responsible for environment interactions and algorithm visualisation. It provides custom collision detection systems and a graphics framework for rendering the internal state of the algorithms.

Generator. This section is responsible for generating and labelling the training data used to train the Machine Learning models.

Trainer. This section is a class wrapper over the third party Machine Learning libraries. It provides a generic training pipeline based on the holdout method and standardised access to the training data.

Analyzer. The final section manages the statistical measures used in the practical assessment of the algorithms. Custom metrics can be defined as well as graphical displays for visual interpretations.

ROS Real-time Extension. The extension provides real-time support for visualisation, coordination and interaction with a physical robot. The extension is split into two main components: **RosMap** (integrates the *gmapping ROS* package (SLAM scan) into a dedicated internal map environment) and **Ros (master node)**. We explain the **Ros (master node)** architecture and functionality in Section 6.6 (Path Planning on Real-world Robot) in which we evaluate the performance of the proposed solution on a real robot.

4.1 Comparison with other motion planner platforms

Currently, there are a variety of standardised libraries which contain at least two of the mentioned sections (**Simulator** and **Analyzer**) such as: *ROS* [17], *OMPL* [18], *MoveIt* [19] (has benchmarking capabilities [20]).

ROS. The Robot Operating System (ROS) is a platform which contains various simulation environments (including 2D and 3D) for different types of robots: ground robots with different degrees of freedom constraints, flying robots (drones) and manipulator robots (arm robots which interact with different objects). It represents the standard library in robotics for simulation and development.

OMPL. The Open Motion Planning Library (OMPL) is a standalone library which focuses on motion planning exclusively. It is more lightweight than ROS and has reduced capabilities (there is no collision detection). The library of available path planners is limited to sampling-based planners such as RRT or PRM (probabilistic road maps), but there is a variety of optimised implementation for each type of planner.

MoveIt. Combines both ROS and OMPL to create a high-level implementation for cleaner and faster development of new algorithms. It has more capabilities than ROS and OMPL and includes custom benchmarking techniques [20].

PathBench (our platform). Our implementation offers a more abstract overview of the environment and is extremely lightweight compared to the mentioned libraries. We include both a simulation environment (**Simulator**) and benchmarking techniques (**Analyser**), but we also include a generator for creating synthetic datasets for Machine Learning applications (**Generator**) and a ML training pipeline (**Trainer**) for generic ML models. The **Analyser** is quite involved and extensible, but the **Simulator** has limited capabilities in both rendering and environment interaction compared to the other libraries. The main motivation of using our platform and not the existing standardised libraries is that the platform was build to be used in an ideal research environment. Since, we use ML methods, the focus was set on the path generation and not the interaction between the environment and the agent. However, we do provide a clean API interface for the algorithms which makes them portable to the standardised libraries. Moreover, we provide a *ROS* real-time extension which converts the internal map move action into network messages

(velocity control commands) using the *ROS* publisher-subscriber APIs (See Table 4.1 for platform comparison).

Platform	Visualisation	Benchmarking	Machine Learning Support	Development Efficiency	Robot Variety	Environment Complexity and Interaction
ROS	✓	✗	✗	**Reduced**	**High**	**High**
OMPL	✓	✗	✗	**High**	**Reduced**	**Reduced**
MoveIt	✓	✓	✗	**Moderate**	**High**	**High**
PathBench	✓	✓	✓	**High**	**Reduced**	**Reduced**

Table 4.1: Platform capabilities comparison chart

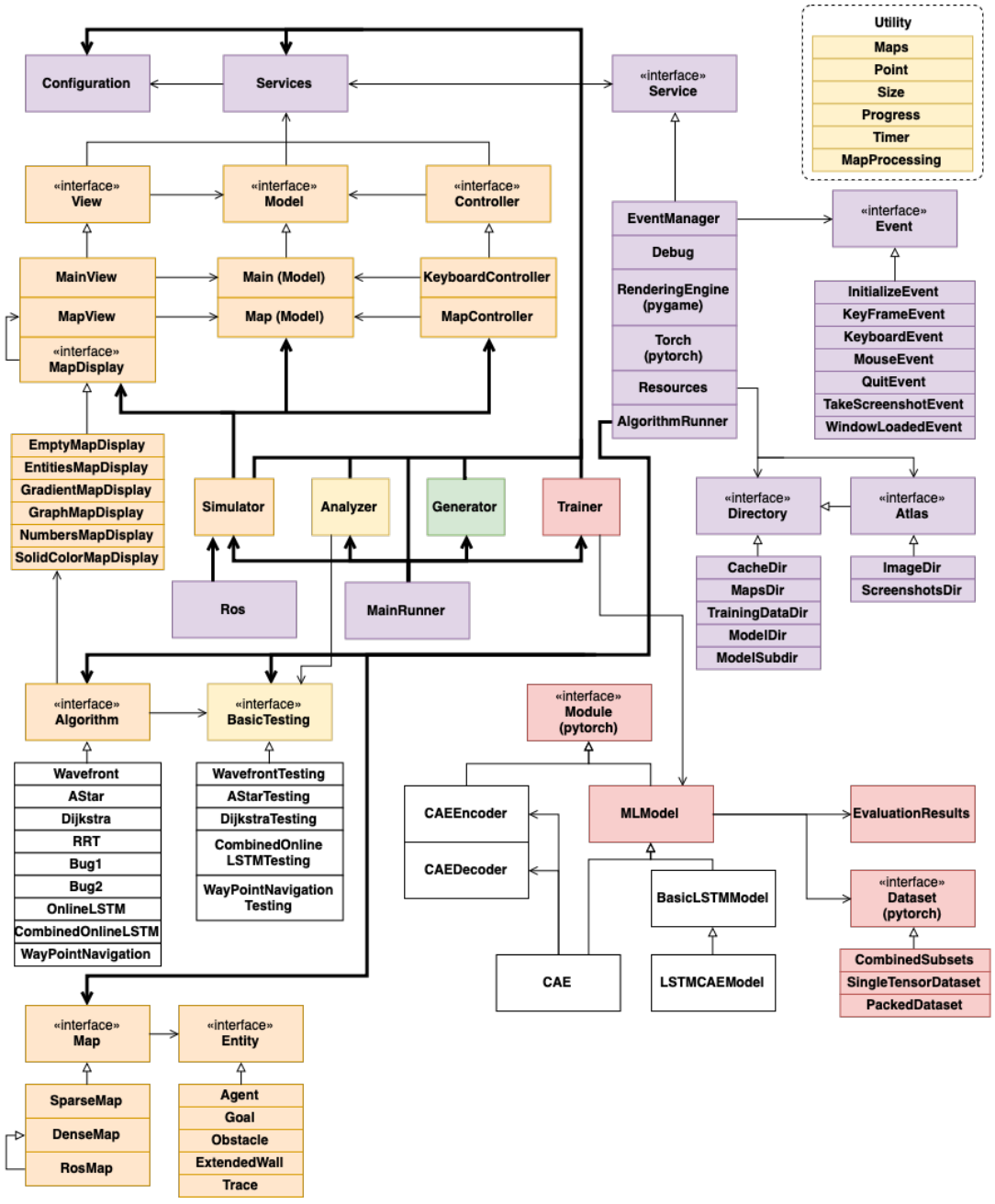


Figure 4.2: Full platform architecture overview. Arrows with full head represent dependency ($A \xrightarrow{depends\,on/uses} B$) and arrows with hollow head represent inheritance ($A \xrightarrow{inherits\,from} B$). Colours are mapped as follows: purple is infrastructure, orange is simulator, green is generator, red is trainer, yellow is utility/analyser and white is extension

4.2 Implementation

Before coding the platform, we have investigated a series of libraries and programming languages and we have decided to write it in *python ver. 3.7.3*, use *pytorch* [37] for machine learning and *pygame* [38] for rendering.

The choice of programming language was straightforward as *python* is the standard in ML and research applications. Moreover, a lot of open source ML libraries are available such as *tensorflow* [39] and *pytorch* which are used both in production software and researching.

For the ML library, we have chosen *pytorch* over *tensorflow*, because *pytorch* was developed with the intent of it being used in research. *tensorflow* is a mature ML library which has extensive community support, and it is used by major companies in production software, but, unlike *pytorch*, it is quite hard to debug, due to the design of the computational graphs. In *tensorflow*, you have to compile the model and use special session variables, while *pytorch* offers the possibility of dynamically changing the computational graphs, which allows the user to debug more easily. This feature is most useful when dealing with RNNs with variable size inputs (in our case the LSTM) [40].

The simulator was build using the rendering library *pygame*. The other choices included *pyglet* [41] and *Unity 3D* [42]. *pyglet* is an advanced rendering engine which is based on *OpenGL* [43], but due to the fact that we do not have graphics intensive requirements, a lightweight library such as *pygame* is a better option. Moreover, *pygame* provides useful rendering helper functions which do not require prior knowledge about *OpenGL*, thus making development faster.

Unity 3D was considered as an alternative to *pygame* as it provides an additional physics engine which has collision detection and ray casting. Both features were needed in later development stages and had to be manually implemented. The main downside to choosing *Unity 3D* was that we could not use *pytorch* anymore, because *Unity 3D* uses *C#* as the core programming language. There were multiple solutions to this issue: use *IronPython*[1] [44], use the *C#* ML wrappers or create a ML server. However, none of them had a good trade-off to make the switch. Moreover, all solutions required different communication protocols which could have severely affected the performance of the path planners.

4.3 Simulator

The **Simulator** is both a visualiser tool and an engine for developing **Algorithm**s (See Figure 4.3). It supports animations and custom **MapDisplay** components which render the **Algorithm**'s internal data. Table A.4 from Appendix A provides a list of user commands which control the simulator during run-time.

A **Map** contains different entities such as the **Agent**, **Goal** and **Obstacle**s, and provides a clean interface that defines the movement and interaction between them. Therefore, a **Map** can be extended to support various environments. The downside to this features is that each map has to implement its own physics engine or use a third party one (such as the *pymunk* physics engine [45] or *OpenAI Gym* [46]). The current implementation supports three 2D maps: **DenseMap**, **SparseMap** and **RosMap**.

The **DenseMap** stores entities in a grid format in order to reduce the time complexity of the APIs (i.e. collision detection, ray casting and line drawing). Collision detection is $\mathcal{O}(1)$, and most operations such as ray casting and line drawing are trivial to implement. When the agent has a radius attached to it, the obstacles are inflated by creating **ExtendedWall** objects around the obstacle boundary (the method is similar to the repulsion function from Potential Fields; time

[1] *IronPython* is a library which provides a *python2* session wrapper that can be directly used in *C#* code

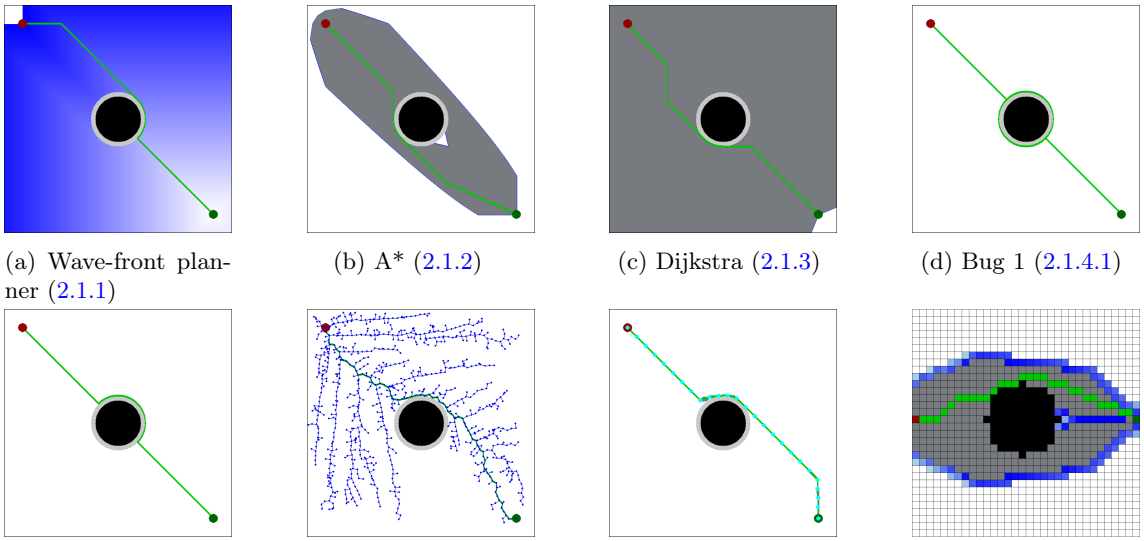
complexity is $\mathcal{O}(xo)$, where x is the inflation rate and o is the average obstacle size). The agent can overlap the **ExtendedWall** obstacles as long as the **ExtendedWall** obstacles do not contain its centre.

Unlike the **DenseMap**, the **SparseMap** stores all entities in a list similar to *Unity 3D*. It does not need obstacle inflation, and it has fast collision detection as only circles are used for each entity (circle collision is $\mathcal{O}(1)$), but it does not implement a pairwise checking algorithm (such as pairwise pruning or sweep and prune) and, thus, the complexity of the whole collision detection system becomes $\mathcal{O}(n^2)$ as all pairs of entities are checked. Thus, the **SparseMap** is mostly used to create user-defined maps which are then converted to a **DenseMap**.

The **RosMap** extends the **DenseMap** to integrate the *gmapping ROS* package (SLAM scan) by converting the SLAM output image into an internal map environment. Because we extend the **DenseMap** component, we inherit the collision detection system and all additional functionality from it. The **RosMap** environment has support for live updates, meaning that algorithms can query an updated view by running a SLAM scan. The map uses simple callback functions to make SLAM update requests and convert movement actions into network messages using the *ROS* publisher-subscriber communication system.

For all maps, movement is allowed in eight directions: horizontal, vertical and diagonal (movement cost is based on the Euclidean distance: horizontal/vertical 1 and diagonal $\sqrt{2}$). It should be noted that, because we are simulating the map environment, we need the full map information for **SparseMap** and **DenseMap**. However, because we provide a **Map** interface, we can define custom maps (such as **RosMap**) that restrict the agent information access (e.g. in the real world, the robot usually does not has access to the full information of the environment).

Additionally, the **Simulator** provides animations that are achieved through key frames and synchronisation primitives (the process is described in Appendix A.1).



(a) Wave-front planner (2.1.1)

(b) A* (2.1.2)

(c) Dijkstra (2.1.3)

(d) Bug 1 (2.1.4.1)

(e) Bug 2 (2.1.4.2)

(f) RRT (2.2.1)

(g) Global Way-point LSTM Planner (5.4)

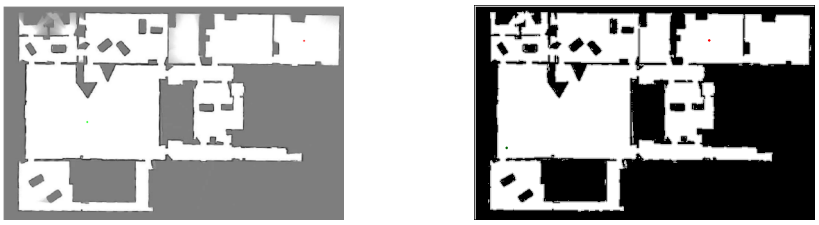
(h) A* (2.1.2) on a grid display

Figure 4.3: Different planners run on the same map (except Sub-figure (h)). The red entity is the agent, the green entity is the goal, light green entities are traces, black/light grey entities are obstacles and everything else is custom display information (e.g. in Sub-figure (b) dark grey represents search space)

4.4 Generator

The **Generator** can execute five actions: conversion from image to map, map generation, map labelling, map augmentation and map modification.

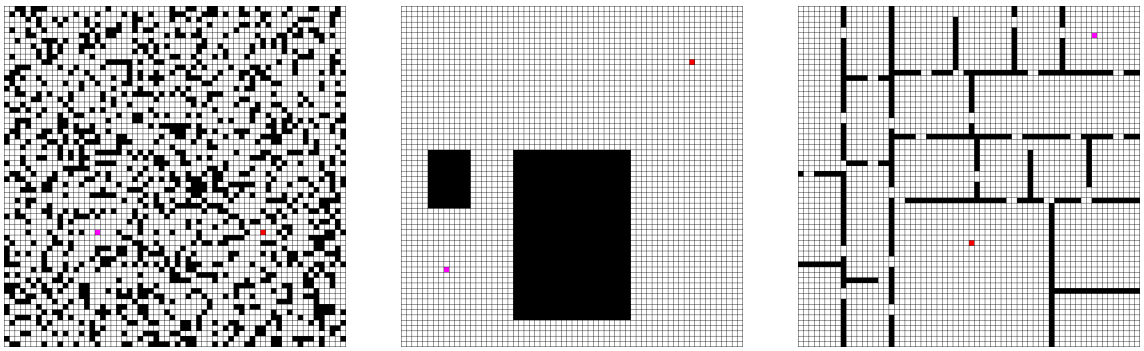
Conversion. An image can be converted into an internal **Map** and saved into the maps directory from **Resources**. The image can be a software drawn image or Simultaneous Localization and Mapping (SLAM) image [21] output from a real robot, as long as it follows the conventions imposed by the image converter: an agent represented by a true red circle has to be present, a goal represented by a true green circle has to be present and the obstacles need to be in the grey-black colour range (See Figure 4.4).



(a) Original SLAM Image (b) Converted **Map**

Figure 4.4: Example of the conversion process from the **Generator**

Generation. The generation procedure accepts as input, different hyper-parameters such as the type of generated map, number of generated maps, obstacle fill rate range, number of obstacle range, minimum room size range and maximum room size range, which define the structure of the maps. When a range is given as input, the generator picks a random number between the range and feeds it into the associated map type generator. Currently, the generator can produce three types of maps: uniform random fill map (See Algorithm 7), block map (See Algorithm 8) and house map (See Algorithm 9) (See Figure 4.5) and it can easily be extended to support different synthetic maps such as mazes and cave generation using cellular automata. All generated maps are placed into a new **Atlas** directory which is a custom directory that saves files using the index number. It keeps track of the next available index, and thus, when a new file is saved, it is "appended" to it. **Atlas** directories are used for easier indexing operations such as index loading and index saving (the file system service is described in Appendix A.1).



(a) Uniform random fill (64x64 dimension, [0.1, 0.3] obstacle fill rate range)

(b) Block map (64x64 dimension, [0.1, 0.3] obstacle fill rate range, [1, 6] number of obstacles range)

(c) House (64x64 dimension, [8, 15] minimum room size range, [35, 45] maximum room size range)

Figure 4.5: The current generated maps are: (a) Uniform random fill (10000 samples), (b) Block map (10000 samples), (c) House atlas (10000 samples). We will use magenta colour for the goal for all generated maps as the dark green goal is quite hard to spot

Algorithm 7 Uniform random fill generator

procedure GENERATE-UNIFORM-RANDOM-FILL-MAP(*dimension*, *obstacle_fill_rate*)
    Initialise empty *map* of dimension *dimension*
    $fill \leftarrow obstacle\_fill\_rate * dimension.width * dimension.height$
    $nr\_of\_obstacles \leftarrow 0$

    while $nr\_of\_obstacles < fill$ **do**
        obstacle_position ← random position within *dimension*
        if *obstacle_position* is free on *map* **then**
            place unity obstacle at *obstacle_position* on *map*
            increment $nr\_of\_obstacles$

    place agent and goal at random free positions on *map*
    return *map*

Algorithm 8 Block map generator

procedure GENERATE-BLOCK-MAP(*dimension*, *obstacle_fill_rate*, $nr\_of\_obstacles$)
    Initialise empty *map* of dimension *dimension*
    $fill \leftarrow obstacle\_fill\_rate * dimension.width * dimension.height$

    for i in $[0, nr\_of\_obstacles)$ **do**
        $next\_obst\_fill$ ← random value from $fill$
        while can't place block **do**
            $first\_side$ ← random value from $next\_obst\_fill$
            $second\_side \leftarrow next\_obst\_fill \;/\; first\_side$
            try to place block of dimension ($first\_side$, $second\_side$) on random position (blocks may overlap)

        $fill \leftarrow fill - next\_obst\_fill$

    place agent and goal at random free positions on *map*
    return *map*

Algorithm 9 House generator

procedure SUBDIVIDE($top\_left\_corner, dimension$)
    if can't split anymore due to min_room_size **then**
        place a room at top_left_corner with dimension *dimension*

    if $dimension \leq max\_room\_size$ **then**
        50% change to place a room at top_left_corner with dimension *dimension*

    random pick vertical or horizontal
    random split vertical/horizontal into 2 blocks: $first\_block$, $second\_block$

    Subdivide($first\_block\_top\_left\_corner, first\_block\_dimension$)
    Subdivide($second\_block\_top\_left\_corner, second\_block\_dimension$)
procedure GENERATE-HOUSE-MAP(*dimension*, min_room_size, max_room_size)
    Initialise empty *map* of dimension *dimension*

    Subdivide($(-1, -1), dimension + 1$)
    for each *room* **do**
        get *room* walls and add doors with probability 25% (max 1 door per wall)

    place agent and goal at random free positions on *map*
    return *map*

Labelling. The labelling procedure takes a map **Atlas** and converts it into training data by picking only the specified features and labels. The training data is then saved as a `.pickle` file with name format as `training_{atlas name}_{number of samples}`. The structure of the training data is based on normal *python* objects (`List[Dict[str, Any]]`) for quick inspection and analysis. Features/labels are picked by using the **MapProcessing** component (See Table 4.2 for feature reference). A* is used as ground truth for feature/label generation. All features/labels can be saved as a variable sequence (needed for LSTM) or single global input (needed for auto-encoder).

Feature/Label Key	Description
agent_position	The current agent position
direction_to_goal	The current direction from the agent to the goal
direction_to_goal_normalized	The normalised direction from the agent to the goal
distance_to_goal	The current Euclidean distance from the agent to the goal
distance_to_goal_normalized(n)	The current normalised Euclidean distance from the agent to the goal (normalisation is done by clamping the output between [0, n))
raycast_8	The current distance from agent to obstacles on all eight directions (vertical, horizontal, diagonal)
raycast_8_normalized(n)	The current normalised distance from agent to obstacles on all eight directions (vertical, horizontal, diagonal) (normalisation is done by clamping the output between [0, n))
agent_goal_angle	The current angle defined by the line segment from the agent to the goal (code $torch.atan2(v.y, v.x)$, where v is the line segment)
valid_moves	The current 0-1 tensor which describes in which direction (vertical, horizontal, diagonal) is agent movement available (1 is available)
global_map	The current whole map obstacles given as a normalised 0-1 tensor (1 is obstacle)
local_map	Like global_map, but it has 9×9 dimension and agent is centred
next_position	The next move direction that the agent should take
next_position_index	The next action that the agent should take given as an index from 0 to 7

Table 4.2: **Generator** list of features and labels. A* is used as ground truth for label annotation

Augmentation. The augmentation procedure takes an existing training data file and augments it with the specified extra features and labels. It is used to remove the need for re-generating a whole training set.

Modification. A custom lambda function which takes as input a **Map** and returns another **Map** can be defined to modify the underlining structure of the map (e.g. modify the agent position, the goal position, create doors, etc.).

4.5 Trainer

The training pipeline is composed of: data pre-processing, data splitting, training, evaluation, results display and pipeline end. All ML models must inherit from the **MLModel** class. The model is passed through the pipeline together with a configuration file (`Dict[str, Any]`) which describes the training process (See Table 4.3 for general configuration hyper-parameters). Each model can define extensions for all pipeline sections and extra configuration parameters.

Data Pre-processing. Data is loaded from the specified training sets, and only the features and labels used throughout the model are picked from the training set and converted to a *pytorch*

Key	Pipeline Section	Type	Description
data_features	Data Pre-processing	`List[str]`	Which sequential features should be picked from training data
data_labels	Data Pre-processing	`List[str]`	Which sequential labels should be picked from training data
data_single_features	Data Pre-processing	`List[str]`	Which single features should be picked from training data
data_single_labels	Data Pre-processing	`List[str]`	Which single labels should be picked from training data
training_data	Data Pre-processing	`List[str]`	Which training data files should be loaded and feature/label picked
validation_ratio	Data Splitting	`float`	How much validation data should be reserved from data
test_ratio	Data splitting	`float`	How much evaluation data should be reserved from data
epochs	Training	`int`	How many times should data be passed during training
batch_size	Training	`int`	Defines the batch size for each epoch
loss	Training	`Callable[[Tensor, Tensor], Tensor]`	Describes the loss function: Arguments(model output, label), Returns(loss value)
optimizer	Training	`Callable[[MLModel], Optimizer]`	Describes the optimizer that should be used: Arguments(model itself), Returns(optimizer)
save_name	Pipeline End	`str`	The model save name

Table 4.3: Training pipeline basic configuration (more algorithm-specific training configurations are provided in Chapter 6 (Evaluation))

Dataset (in total there can be four datasets: one feature sequence, one single feature tensor, one label sequence and one single label tensor). Sequential data is wrapped into a **PackedDataset** which sorts the input in reverse order of its sequence length (max length first, min length last). The data is packed into a *pytorch* **PackedSequence** object using the `pack_sequence` function from *pytorch*. The `pack_sequence` function only accepts sorted sequences (i.e. the input should be an upper triangular matrix) in order to increase the speed of an LSTM forward pass. The **PackedDataset** class saves the sequence itself, the lengths and the sorting permutation. If both sequence and single features are available the single feature tensor is sorted as well according to the permutation used in **PackedDataset** and wrapped into a **TensorDataset**. Both datasets are returned as a **CombinedSubsets** object. The resulting **Dataset** is cached, because data pre-processing takes a long time and consumes a lot of memory (for 30000 samples, over 16 GB of RAM are needed).

Data Splitting. The pre-processed data is shuffled and split into three categories: training, validation and testing (usually 60%, 20%, 20%) according to the Holdout method. The **CombinedSubsets** object is used to couple the feature dataset and label dataset of the same category into a single dataset. Then, all data is wrapped into its **DataLoader** object with the same batch size as the training configuration (usually 50).

Training. The training process puts the model into training mode and takes the training **DataLoader** and validation **DataLoader** and feeds them through the model n times, where n is the number of specified epochs. The training mode allows the gradients to be updated and at each new epoch, the optimiser sets all gradients to 0. Each model has to extend a special `batch_start` hook function which is called on each new batch. The `batch_start` function is responsible for passing the data through the network and returning the loss result. The trainer takes the loss result and applies a backward pass by calling the `.backward()` method from the loss. Afterwards, the optimiser is stepped, and the weights of the model are updated. The statistics, such as the loss over time, for the training and validation sets are logged by two **EvaluationResults** objects (one for training and one for validation) which are returned to the pipeline. The **EvaluationResults** class contains several hook functions which are called through the training process at their appropriate times: `start`, `epoch_start`, `epoch_finish`, `batch_start`, `batch_finish`, `finish`. At each epoch end, the **EvaluationResults** object prints the latest results.

Evaluation. The evaluation process puts the model into evaluation mode and has a similar structure to the training process. The evaluation mode does not allow gradients to update. The testing dataset is passed only once through the model and an **EvaluationResults** object containing the final model statistics is returned to the pipeline.

Results Display. This procedure displays the final results from the three **EvaluationResults** objects (training, validation, testing) and final statistics such as the model loss are printed. The training and validation loss logs are displayed as a *matplotlib* [47] figure. This method can be easily extended to provide more insight into the network architecture (e.g. the **CAE** model displays a plot which contains the original image, the reconstructed version, the latent space snapshot and the resulting feature maps).

Pipeline End. At the end, the model is saved by serialising the model `.state_dict()`, the model configuration, the plots from results display process and the full printing log into a **ModelSudir** under **ModelDir**. The save name is formated according to the following convention: `{config save_name}_{config training_data}_model`.

4.6 Analyser

The **Analyser** is used to assess and compare the performance of the path planners. This is achieved by making use of the **BasicTesting** component. When a new session is run through the **AlgorithmRunner**, if a **BasicTesting** component is attached to it, the session records a series of statistical measures depending on the type of testing (See Table 4.4). The **BasicTesting** component is also linked to the simulator to enable visualisation testing. The key frame feature and synchronisation variable are tied to the **BasicTesting** component, which allows the user to enhance each key frame and define custom behaviour. Each **Algortihm** instance can create debugging views called **MapDisplay**s which can render custom information on the screen such as the the internal state of the **Algortihm** (e.g. Search Space, Total Fringe) (See Table 4.5).

Instead of manually running a **Simulator** instance to assess an **Algorithm**, the **Analyser** has an extensive algorithmic analysis procedure split into two parts: simple analysis and complex analysis. We also provide a training dataset analysis routine for inspecting the generated maps.

Simple Analysis. n (usually 10) map samples are picked from each generated map type, and m algorithms are assessed on them. The results are averaged and printed.

Complex Analysis. n maps are selected (generated or hand-made), and all m algorithms are run on each map x (usually 50) times with random agent and goal positions. As in the simple analysis stage, the results are averaged and reported. In the end, all $n \times x$ results are averaged and reported.

Training Dataset Analysis. A training set analyser procedure is provided to inspect the training

Name	Supported Algorithms	Description
Map	`All`	The actual map
Trace	`All`	The actual trace points
Map Obstacle Ratio	`All`	The percentage of obstacles from the map
Original Distance	`All`	The Euclidean distance from agent to goal at the beginning of the algorithm
Algorithm Type	`All`	The type of the algorithm that was run
Success Rate	`All`	The rate of success of finding a path from the agent to the goal
Steps	`All`	The total steps (movements) taken to reach the goal
Distance	`All`	The total distance taken to reach the goal. Steps is different from Distance as the diagonal movement cost is 1, but the diagonal movement cost is $\sqrt{2}$
Time	`All`	The total time taken to reach the goal
Distance Left	`All`	In case of failure what is the Euclidean distance left from the agent to the goal
Search Space	`A*`, `Dijkstra`	The search space that was used to find the path (visited set without priority queue)
Total Fringe	`A*`, `Dijkstra`	The left priority queue size, after the goal was found
Total Search	`Wave-front`, `A*`, `Dijkstra`	Total Search = Search Space + Total Fringe

Table 4.4: **Analyser** general statistic measures (more algorithm-specific metrics are provided in Chapter 6 (Evaluation))

Display Name	Supported Algorithms	Description
Map	`All`	Displays the map in two modes: grid, normal
Entities	`All`	Displays the map entities: clear tiles (white), agent (red), goal (dark green), obstacles (black), extensions (light grey), trace (light green)
Step Grid (gradient)	`Wave-front`	Displays the step grid (white-dark blue gradient, min is white, max is dark blue)
Step Grid (numbers)	`Wave-front`	Displays the actual numbers from the step grid (simulator has to be launched in grid display mode)
Search Space and Total Fringe	`A*`, `Dijkstra`	Displays the visited set (dark grey) and the priority queue (fringe) (light blue-dark blue gradient, the darker the blue, the higher the priority)
Graph	`RRT`	Displays the graph edges (blue lines) and graph nodes (blue circles)

Table 4.5: **Algorithm** information displays (more algorithm-specific information displays are provided in Chapter 6 (Evaluation))

datasets by using the basic metrics defined in Table 4.4 (e.g. Original Distance, Success Rate, Map Obstacle Ratio).

All printing from the three sections is saved in log files in the **Resources** directory. In order to view and interpret the results in a friendlier format, the results are tabulated (a latex table generator helper function is used to transfer the results from the log to the report).

Chapter 5

Methods

In this section we are going to present the proposed solutions. We have implemented an existing solution: Online LSTM Planner (replica of [15]) and developed three proposed solutions: CAE Online LSTM Planner, LSTM Bagging Planner and Global Way-point LSTM Planner. For each solution, we are going to theoretically prove the worst case (average case) space and time complexity, state the optimality conditions, state a few advantages and disadvantages and showcase a few algorithm runs on the synthetically generated maps (i.e. the training datasets). It should be noted that the detailed training information and exhaustive experiments will be presented in Chapter 6 (Evaluation).

Online LSTM Planner. The planner attempts to replicate the solution from [15]. The algorithm essentially uses an LSTM network to retrieve the next action that the agent should take given the current location data (localisation and local surroundings information). The planner will be later used in the final proposed solution and provides a strong frame of reference for the empirical experiments.

CAE Online LSTM Planner. The algorithm is a hybrid solution between [15] and [1] which attempts to fix some issues that are present in the Online LSTM Planner (e.g. the algorithm does not know how to navigate between complex obstacles and long corridors). This is done by augmenting the input from the LSTM network with the compressed global image snapshot (when we are dealing with partial knowledge environments, we are still going to use the global image snapshot, but we are going to include the unknown environment as well). The global snapshot is compressed using a Convolutional Auto-encoder (CAE).

LSTM Bagging Planner. It is a solution inspired by ML ensemble methods which combines the previous algorithms (Online LSTM Planner and CAE Online LSTM Planner) into a unique best-of-all kind of algorithm. The planner attempts to boost the performance of the previous algorithms by picking the best solution depending on the layout of the environment.

Global Way-point LSTM Planner. It represents the final proposed solution. The planner uses one local kernel and one global kernel. The global kernel is responsible for suggesting a series of way-points which will guide the agent through the environment and the local kernel is responsible for the actual manoeuvring between way-points. The planner uses the LSTM Bagging Planner as the global kernel by transforming it into a way-point suggestion algorithm. This is achieved by bounding the number of iterations of the LSTM Bagging Planner. Any classic solution can be used as the local planner, but we have decided to use A* as it represents the base algorithmic frame of reference against all other proposed solutions in Chapter 6 (Evaluation).

5.1 Online LSTM Planner

The Online LSTM Planner algorithm is a close replica of [15], with some architectural and logic changes. The planner uses an LSTM network to query the next agent action (next movement) at each time step.

5.1.1 LSTM Architecture

The LSTM architecture takes four types of inputs: the normalised distance between the agent to obstacles on all eight directions (raycast_8_normalised(50)), the normalised direction to the goal (direction_to_goal_normalised), the angle defined by the direction to the goal (agent_goal_angle) (not required to be normalised as it is already bounded by definition), the normalised distance to the goal (distance_to_goal_normalised(100)) (See Figure 5.1) and returns the next agent action/-movement index (next_position_index) (See Table 4.2 for reference). No data pre-processing is done as the input is already normalised and we use batch normalisation layers.

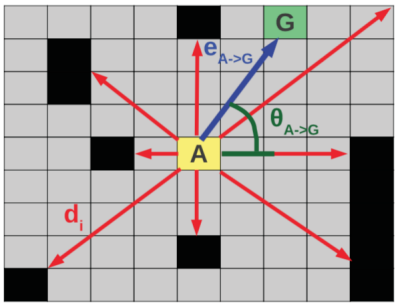


Figure 5.1: Input visualisation [15]. d_i is raycast_8_normalised(50), $\theta_{A\to G}$ is agent_goal_angle and $e_{A\to G}$ is both direction_to_goal_normalised and distance_to_goal_normalised(100)

The network uses the cross entropy loss function which combines both log softmax and negative log likelihood into a single function:

$$L(\boldsymbol{x}, y) = -\log \frac{e^{x_y}}{\sum_j e^{x_j}}, \text{ where } \boldsymbol{x} \text{ is the prediction for all classes and y is the actual class}$$

The model contains a hidden state and cell state which are initialised at each new batch with a 0 tensor of size $2 \times lstm\_layers \times batch\_size \times lstm\_output\_size$. The architecture has the following structure: one batch normalisation layer, two LSTM layers, one batch normalisation layer and one linear layer (See Figure 5.2).

The architecture of the network is almost identical to the one from [15], but we do not feed the previous agent action as the paper does due to performance reasons. The data has to be constantly packed and unpacked before it is fed through the LSTM layers. When the data has to be passed through an LSTM layer, it is packed, and when it has to be passed through a linear layer, it has to be unpacked. If the previous action was added as an input, then the data had to be packed and unpacked for each sequence step from the batch, which is a severe performance downgrade (packing and unpacking is thoroughly described in Appendix B). Moreover, because we use an LSTM network, the previous action information lies into the hidden and cell state already.

Batch Normalisation Layer. Unlike the paper, we use batch normalisation layers to speed up the learning process by reducing the covariance shift (the hidden unit values shift). Moreover, the network can generalise better for unseen examples if they belong to the same distribution as the training data (e.g. if we train the model on a dataset composed of black cats, the network will not identify coloured cats, but if we use batch normalisation, since coloured cats belong to the same distribution as black cats (both of them are cats and share the same physical appearance, but have different colour) the network will recognise unseen coloured cats as well) [48].

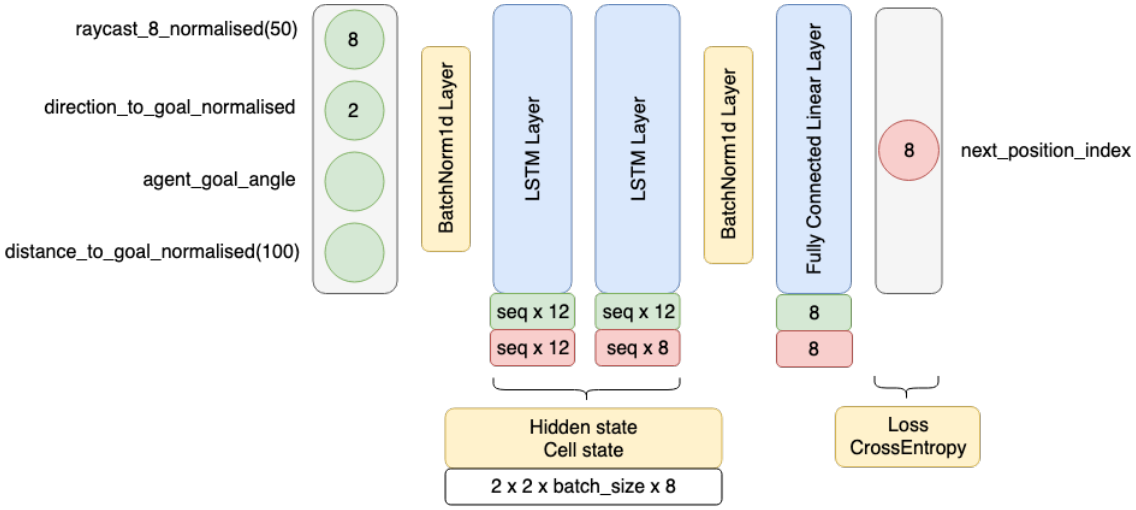


Figure 5.2: LSTM architecture overview

LSTM Layer. This is the core layer of the network. The layer accepts packed/unpacked sequence data as input, but packed data yields a range of advantages including a training performance boost and input masking which reduces the probability of a preferential action (the exact details are provided in Appendix B). We are using two layers as we have to learn non-linear data.

Linear Layer. This is a standard layer from a neural network which contains n weights and one bias. The layer is added to the end of the network to convert the output of the LSTM to an action.

The algorithm itself is trivial as we only need to extract the mentioned features from the map by using the **MapProcessing** utility class, feed one tensor at a time and execute the given action. The algorithm takes two additional inputs: $model\_name$ and max_it (with default value ∞). The $model\_name$ specifies which model has to be loaded (because the model save name is based on the training set name, we can choose which model we want to use based on the dataset on which it was trained). The max_it argument states the maximum number of iterations after which the algorithm exits, even if it has not found a path. Before running the main loop, the model has to initialise the hidden and cell state with a 0 tensor of shape $2 \times 2 \times 1 \times 8$ ($batch\_size$ is 1). Sometimes the network will start to oscillate between two points if it cannot find a path (even if one exists). In order to avoid running the algorithm forever, we have implemented a fail-safe mechanism. If a location is visited more than 5 times the algorithm is considered stuck, and it aborts the execution (See Algorithm 10).

Algorithm 10 Online LSTM Planner

procedure ONLINE-LSTM-PLANNER(M: (A, Os, G), $model\_name$, $max\_it = \infty$)
    $model \leftarrow$ load model with save name $model\_name$
    Initialise $model$ hidden and cell state with a 0 tensor of shape $2 \times 2 \times 1 \times 8$
    $history\_frequency \leftarrow \{:\}$

    for i in $[0, max\_it)$ **do**
        if G was reached **then**
            return
        $features \leftarrow$ extract feature tensor from M by using **MapProcessing** utility class
        $next\_action \leftarrow model.forward$(features)
        if $next\_action$ is valid **then**
            Move agent A according to $next\_action$

        $history\_frequency[A] \leftarrow 1$ (if no value) or $history\_frequency[A] + 1$
        if history_frequency[A] > 5 **then**
            return

5.1.2 Complexity Analysis

Theorem 5.1: 2-dimensional Complexity

Worst case time complexity (2D environments; See Theorem 5.3 for proof):

$$\mathcal{O}(OnlineLSTM) = \mathcal{O}(xo + \min(max\_it, d))$$

Worst case space complexity (2D environments; See Theorem 5.4 for proof):

$$\mathcal{O}(OnlineLSTM) = \mathcal{O}(1)$$

*where xo is the inflation pre-process from **DenseMap** (this is also present in A* when it is run on a **DenseMap**), x is the inflation rate, o is the average obstacle size and d is the solution depth.*

Theorem 5.2: D-dimensional Complexity

Higher dimensional worst case time complexity:

$$\mathcal{O}(OnlineLSTM) = \mathcal{O}(xo + \min(max\_it, d)\, 3^D)$$

Higher dimensional worst case space complexity:

$$\mathcal{O}(OnlineLSTM) = \mathcal{O}(3^D)$$

*where xo is the inflation pre-process from **DenseMap** (this is also present in A* when it is run on a **DenseMap**), x is the inflation rate, o is the average obstacle size, d is the solution depth and D is the dimension size.*

Proof

In higher dimensions, raycasting is increased to the number of edges in the dimension. Thus, in a 3D world we need 26 raycasts instead of 8. $\mathcal{O}(raycast\_8\_normalised(50))$ becomes $\mathcal{O}(raycast(D, 50)) = \mathcal{O}(3^D \times 50) = \mathcal{O}(3^D)$. Thus, the worst case time complexity becomes $\mathcal{O}(OnlineLSTM) = \mathcal{O}(xo + \min(max\_it, d)\, 3^D)$ and the worst case space complexity becomes $\mathcal{O}(OnlineLSTM) = \mathcal{O}(3^D)$.

Theorem 5.3: General 2-dimensional Worst Case Time Complexity

The general worst case time complexity (2D environments) is:

$$\begin{aligned}\mathcal{O}(OnlineLSTM) = \mathcal{O}(xo + \min(max\_it, d)\,(&\mathcal{O}(collision\_detection)+\\ &\mathcal{O}(movement\_action)+\\ &\mathcal{O}(feature\_extraction)+\\ &\mathcal{O}(network\_pass)))\end{aligned}$$

*where xo is the inflation pre-process from **DenseMap** (this is also present in A* when it is run on a **DenseMap**), x is the inflation rate, o is the average obstacle size and d is the solution depth.*

Proof

$\mathcal{O}(collision\_detection)$ *is* $\mathcal{O}(1)$ *as we are using the **DenseMap** component.*
$\mathcal{O}(movement\_action)$ *(moving the agent to an adjacent location) is* $\mathcal{O}(1)$.

$$\begin{aligned}\mathcal{O}(feature\_extraction) = \mathcal{O}(&\mathcal{O}(raycast\_8\_normalised(50))+\\ &\mathcal{O}(direction\_to\_goal\_normalised)+\\ &\mathcal{O}(agent\_goal\_angle)+\\ &\mathcal{O}(distance\_to\_goal\_normalised(100)))\end{aligned}$$

$\mathcal{O}(raycast\_8\_normalised(50))$ *is* $\mathcal{O}(8 \times 50) = \mathcal{O}(1)$ *(one raycast is* $\mathcal{O}(n)$*, in our case* $n = 50$ *and we compute 8 raycasts; it should be noted that this operation is a bit more expensive in practice, but it is bounded).*

$$\begin{aligned}\mathcal{O}(direction\_to\_goal\_normalised) &= \mathcal{O}(agent\_goal\_angle)\\ &= \mathcal{O}(distance\_to\_goal\_normalised(100))\\ &= \mathcal{O}(1)\end{aligned}$$

Therefore, $\mathcal{O}(feature\_extraction)$ *is* $\mathcal{O}(1)$.

$$\begin{aligned}\mathcal{O}(network\_pass) = \mathcal{O}(&\mathcal{O}(packing) + \mathcal{O}(unpacking)+\\ &2\mathcal{O}(batch\_norm) + 2\mathcal{O}(lstm) + \mathcal{O}(linear))\end{aligned}$$

$\mathcal{O}(packing)+\mathcal{O}(unpacking)$ *is* $\mathcal{O}(n \log n)$ *(in our case* $n = 1$ *so complexity is* $\mathcal{O}(1)$*).*

$\mathcal{O}(batch\_norm)$ *is* $\mathcal{O}(n)$ *where* n *is the size of the 1D tensor (in our case* $n = 12$ *and* 8*;* $\mathcal{O}(12) = \mathcal{O}(8) = \mathcal{O}(1)$*, so batch normalisation pass is* $\mathcal{O}(1)$*).*

$\mathcal{O}(linear)$ *is* $\mathcal{O}(batch\_size \times input\_size \times output\_size)$ *(matrix multiplication) (in our case,* $batch\_size = 1$*,* $input\_size = 8$ *and* $output\_size = 8$*, so* $\mathcal{O}(linear) = \mathcal{O}(8 \times 8) = \mathcal{O}(1)$*).* $\mathcal{O}(lstm)$ *is* $\mathcal{O}(sequence\_size \times \mathcal{O}(linear))$*, but in our case* $sequence\_size = 1$ *and therefore* $\mathcal{O}(lstm) = \mathcal{O}(1)$.

Lastly, $\mathcal{O}(network\_pass) = \mathcal{O}(1)$ *(it should be noted that even if time complexity is* $\mathcal{O}(1)$*, this operation takes a bit of time, but it is bounded).*

Theorem 5.4: General 2-dimensional Worst Case Space Complexity

The general worst case space complexity (2D environments) is:

$$\mathcal{O}(OnlineLSTM) = \mathcal{O}(\mathcal{O}(model\_size) + \mathcal{O}(map\_features) + \mathcal{O}(network\_pass) + \mathcal{O}(hidden\_cell\_state))$$

Proof

$\mathcal{O}(model\_size)$ is the loaded model architecture size which can be treated as $\mathcal{O}(1)$.

$\mathcal{O}(map\_features)$ is $\mathcal{O}(12) = \mathcal{O}(1)$ (we extract 12 features: 8 raycast_8_normalised(50), 2 direction_to_goal_normalised, 1 agent_goal_angle, 1 distance_to_goal_normalised(100)).

$\mathcal{O}(network\_pass)$ is the maximum size of the transformed input which is $\mathcal{O}(12) = \mathcal{O}(1)$.

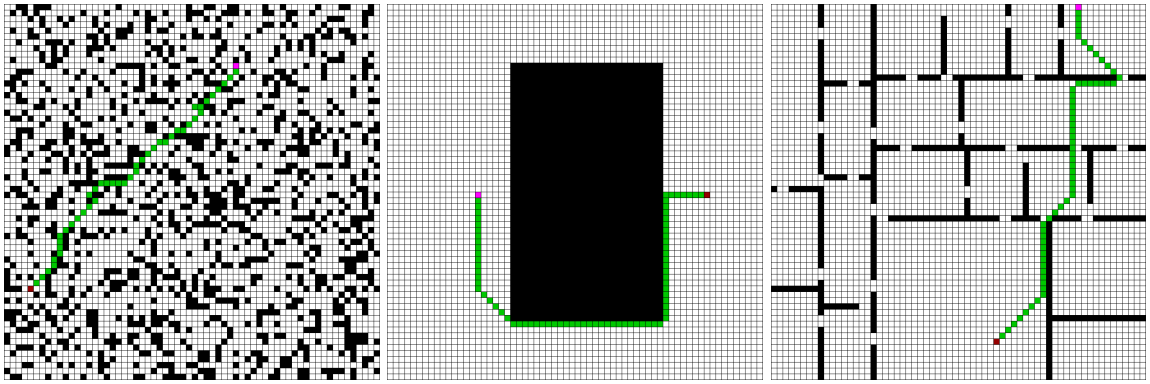
$\mathcal{O}(hidden\_cell\_state)$ is $\mathcal{O}(2 \times lstm\_layers \times batch\_size \times lstm\_output\_size)$ (in our case, $\mathcal{O}(hidden\_cell\_state) = \mathcal{O}(2 \times 2 \times 1 \times 8) = \mathcal{O}(1)$).

5.1.3 General Discussion

This solution does not usually find the optimal path, but it is relatively close to the performance of A*. When running the algorithm in simple environments (maps that contain several simple obstacles that resemble primitives (circles, squares) with/without clear sections (areas where the path is not obstructed by any obstacle)) the optimal performance is equal (most of the time) or insignificantly smaller than A*.

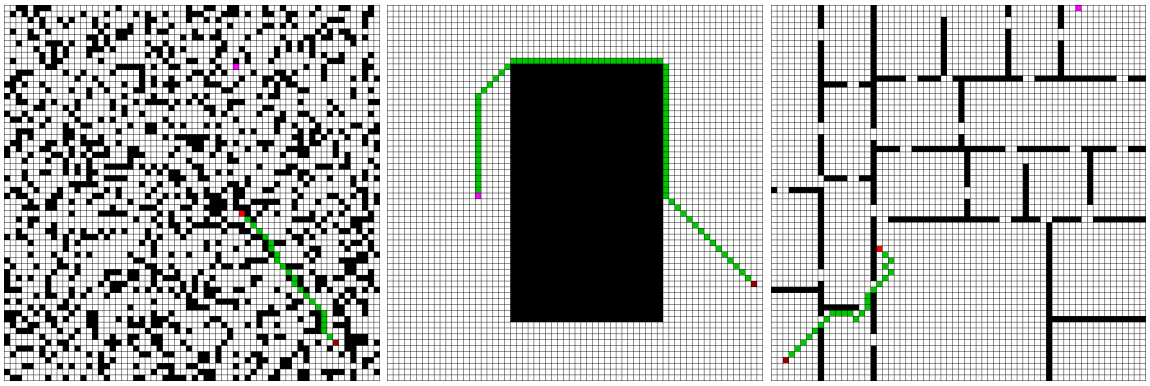
A major advantage over A* is that this method uses significantly less memory as no exploration is done ($\mathcal{O}(OnlineLSTM) = \mathcal{O}(1) < \mathcal{O}(A*) = \mathcal{O}(\hat{b}^d)$) (See Figure 5.3). When the map size is small (64×64) the algorithm is usually slower than A*, due to the feature extraction and network pass, but when the map size gets bigger (600×600 after empirical run, but could be smaller), as it usually is the case in the real world, the algorithm is faster than A* (and will be for larger maps as well). This is intuitively and theoretical correct as we can notice that if we inspect the complexities of both algorithms: $\mathcal{O}(OnlineLSTM) = \mathcal{O}(xo + \min(max\_it, d)) < \mathcal{O}(A*) = \mathcal{O}(xo + \hat{b}^d)$. The exponential time and space complexity increase in higher dimensions is reduced as well (time: $\mathcal{O}(OnlineLSTM) = \mathcal{O}(xo + \min(max\_it, d)\, 3^D) < \mathcal{O}(A*) = \mathcal{O}(xo + (3^D)^d)$, space: $\mathcal{O}(OnlineLSTM) = \mathcal{O}(3^D) < \mathcal{O}(A*) = \mathcal{O}((3^D)^d)$). It should be noted that A* still reduces the time and space complexity even in higher dimensions ($\mathcal{O}(\hat{b}^d) < \mathcal{O}((3^D)^d)$, but it highly depends on the heuristic choice. Lastly, the algorithm is online (supports external updates to the internal state), meaning that it supports dynamic and partial knowledge environments, unlike A*.

By following the paper implementation, we have inherited some issues. The major drawback is that the algorithm is quite greedy (due to the nature of the A* heuristic) and it does not know how to go around big obstacles and long corridors. Therefore, when the environment contains complex obstacles, the algorithm might not find a path to the goal (See Figure 5.4). Moreover, it is quite hard to infer the obstacle shape from the model input and adding more hard-coded features to describe the shape of the obstacle (e.g. the length of the obstacle boundary, the bounding box size of the obstacle and so on) is not feasible and increases the computational cost.



(a) Goal found (b) Goal found (c) Goal found

Figure 5.3: Successful Online LSTM Planner trained on all 30000 generated maps paths



(a) Goal not found (b) Path too long (c) Goal not found due to long corridors

Figure 5.4: Failed Online LSTM Planner trained on all 30000 generated maps paths

5.2 CAE Online LSTM Planner

The Convolutional Auto-encoder (CAE) Online LSTM Planner is a hybrid architecture based on the Online LSTM Planner and [1]. The algorithm attempts to solve the big obstacle navigation issue mentioned in the previous section by supplying extra features that describe the environment to the model (the global image snapshot). This is done by using a Convolutional Auto-encoder (CAE) to encode the global map features and feed the encoder output to the LSTM model (See Figure 5.5).

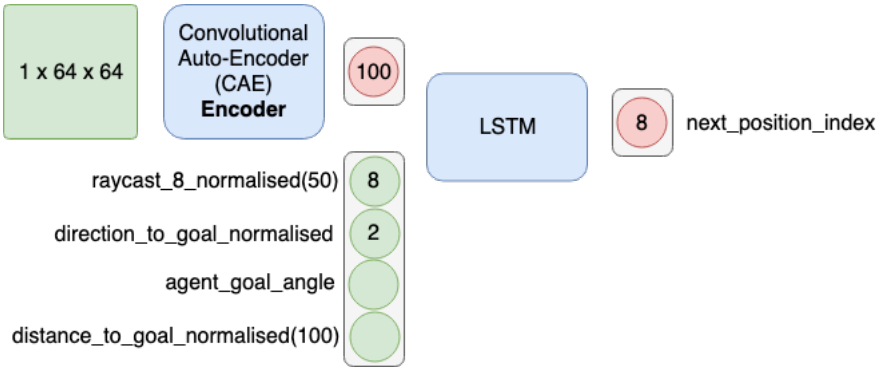


Figure 5.5: CAE and LSTM network architecture overviews

5.2.1 CAE Architecture

In order to avoid obstacle hard-coded features, we can make use of a CAE [49, 31] or Principal Component Analysis (PCA) [50, 51] model to extract the most representative features from the global map. Thus, we save computational performance and we do not need to manually examine each obstacle. The CAE was chosen over PCA as the training pipeline had to be slightly modified to feed the batched input to the PCA by following a similar approach to the **IncrementalPCA** model from *sklearn*. Moreover, if batches were removed, we would have probably run out of memory as the training set is quite large (500 MB for 10000 64×64 uniform random fill maps). Lastly, [1] has achieved great success rate results by using a CAE.

We have chosen a Convolutional Auto-encoder (AE) instead of a normal Linear AE because we would like to learn reproducible patterns from the input image. By creating convolution layers we learn different sets of localised patterns in the image [49, 31].

The CAE is trained on the map training datasets, but it can also be trained on different standard datasets such as MNIST [34] (implemented), CIFAR-10 [52] (can be extended). The CAE includes a specialised results display procedure. Depending on the training dataset, it displays two plots with row size equal to the number of different classes of maps and each row contains three images: the original image, the reconstructed version and the latent space viewed as a 2D grey scale image (the latent space is reshaped). The reason why we display two plots is that we would like to inspect how the latent space varies on maps that are within the same class. It also applies the same procedure if the MNIST dataset is used instead, but it only displays one plot with the first three samples. Furthermore, we plot the feature maps (each convolution output) for the three types of maps.

The CAE model architecture contains two sections: the Encoder and the Decoder (See Figure 5.6). The CAE uses the L1 loss function:

$$L(\boldsymbol{x}, \boldsymbol{y}) = \|\boldsymbol{x} - \boldsymbol{y}\|_1 \text{, where } \boldsymbol{x} \text{ is the prediction and } \boldsymbol{y} \text{ is the target}$$

The scope of the CAE is to compress the given image into a small sized vector (the latent space) which contains the most significant features of the image and then reconstruct it back. We pre-process the data by normalising the image with mean 0.5 and standard deviation 0.5. Thus, the input is in the range [-1, 1]. We provide a flag that enables or disables skip connections between a convolution and the mirror de-convolution. By using skip-connection, we make use of the data that was lost during compression to reconstruct the image and thus, the performance improves (the Encoder learns more information as well).

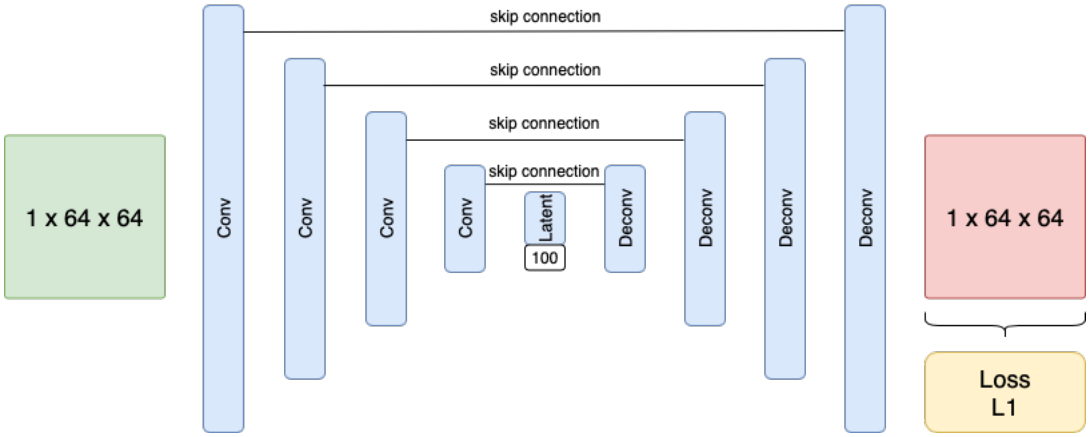


Figure 5.6: CAE model architecture overview

The CAE Encoder contains four convolutional layers and one linear layer as seen in Figure 5.7. Each convolution layer is composed of multiple layers placed in this following order: convolutional layer, batch normalisation layer, max pool and leaky ReLU as the activation function. The final layer of the encoder is a linear layer with another batch normalisation layer.

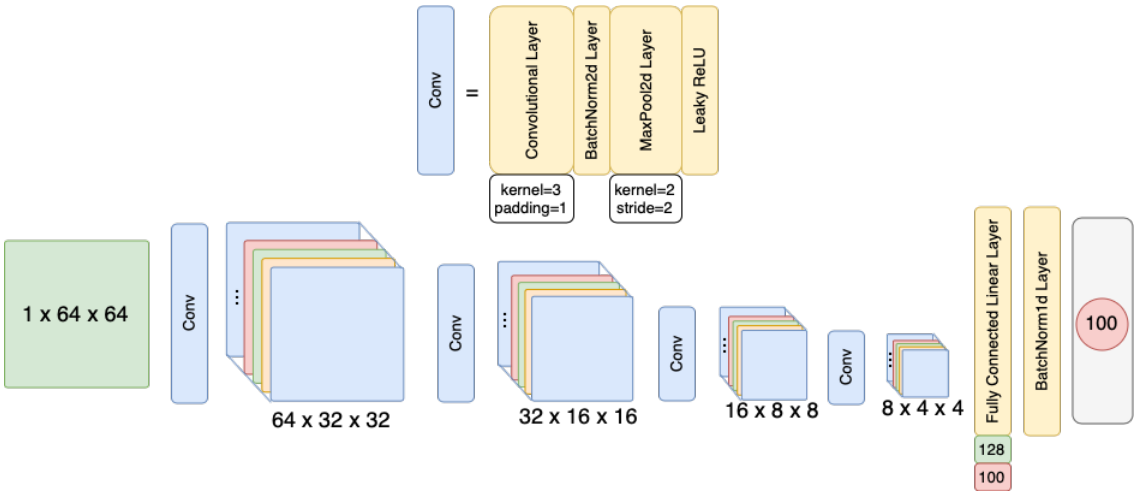


Figure 5.7: CAE Encoder architecture

Convolutional Layer. The convolutional layer uses a kernel window with learnable weights and bias, which is applied to the input image to extract the localised features. In our model, the kernel window has $(c) \times (i) \times 3 \times 3$ dimension where c is the number of output channels, and i is the number of input channels (the window is applied to all input channels). The padding parameter defines how much the image is padded with 0s (in our case, we have 1 padding for all convolutions). The stride parameter defines how much the kernel window is shifted along the image (in our case, we use the default value 1). By using a 3×3 kernel and 1 padding, we preserve the image dimensions and change the number of channels.

Batch Normalisation Layer. We use batch normalisation layers for the same reason described in the the Online LSTM Planner.

Max Pool. The max pool layer has the same arguments as the convolutional layer (kernel, padding, stride), but the kernel has no learnable parameters. By using a max pool layer with kernel size 2, padding 0 and stride 2, we reduce the image dimension by half. Max pooling is applied to all channels and thus, the number of output channels is identical to the number of input channels.

Leaky ReLU. The activation function breaks the linearities between two layers. Thus, the layers cannot be collapsed into a single layer. The Leaky ReLU function ($f(x) = \begin{cases} a \cdot x, x < 0 \\ x, \quad x \geq 0 \end{cases}$) is a modified version of the ReLU function which allows clamped negative values. The variable that defines the clamping (a) is a learnable parameter, and it is updated during back-propagation. After running a series of empirical evaluations on the training routine, we have decided to use Leaky ReLU (instead of ReLU) as the extra learnable parameter has improved the overall performance of the network.

Linear Layer. We include a linear layer at the end to convert the final image (by reshaping it along with all channels) into the latent vector.

The CAE Decoder contains one linear layer and four de-convolutional layers as seen in Figure 5.8. Each de-convolutional layer is composed of multiple layers placed in the following order: de-convolutional layer, batch normalisation layer (used for the same reason as described above) and ReLU activation function. The last de-convolutional layer has Tanh activation function as the input is normalised in the range [-1, 1] and the output of Tanh matches it.

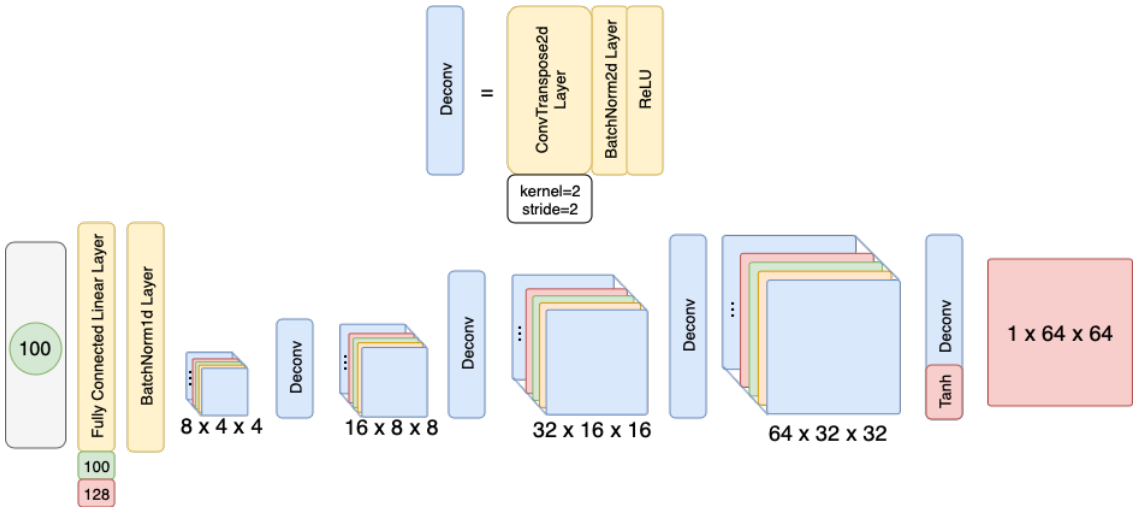


Figure 5.8: CAE Decoder architecture

Linear Layer. The initial linear layer converts the latent space back to the compressed image.

De-convolutional Layer. De-convolution is achieved by using the **ConvTranspose2d** module (kernel size 2, stride 2, padding 0) from *pytorch*. It represents the inverse operation of a convolution and, in our case, it acts as a layer that contains a convolutional layer with max pool that increases the image size by half (instead of reducing it).

ReLU. Again, after running empirical evaluations, we have noticed that ReLU achieves better performance. Moreover, by not allowing negative values, we force the encoder to learn and the network will not reconstruct the image only from the decoder (by having a 0 latent space).

5.2.2 LSTM Architecture

The LSTM network is identical to the LSTM architecture of the Online LSTM Planner and uses the same loss function (cross entropy loss). The only difference is that we feed the CAE Encoder output along with all inputs of the LSTM network (See Figure 5.9). We define a different model because we have to pre-process the input and join sequence data with single global images. Global images are replicated n_i times where n_i is the sequence length of sample i. Afterwards, the images are sorted according to the permutation that was used to sort the sequence input and joined together into a single tensor feature (per sample, per sequence).

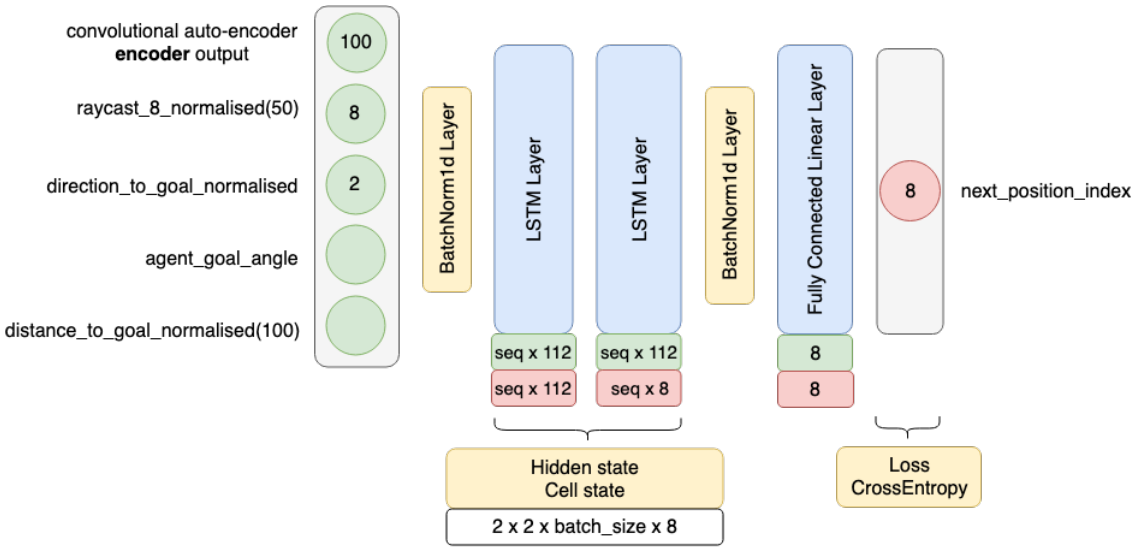


Figure 5.9: LSTM network architecture overview

The algorithm itself is identical to the Online LSTM Planner (See Algorithm 10), but we load the LSTM model associated with this planner instead. When we load the LSTM model, we cache the **encoded** global map snapshot scaled to 64×64 size (regardless if it is smaller or bigger). At each LSTM network pass, we concatenate the cached encoded map to the extracted features. It should be noted that the algorithm is theoretically online as it uses the same model as the Online LSTM Planner. However, because we cache the map in the beginning to increase the time performance, the algorithm might have inferior execution results when we discover more areas within the map. A simple solution to maintain the same performance as the Online LSTM Planner would be to take global map snapshots at each time step instead of caching the first result, while trading time efficiency. Nonetheless, we have made this choice due to the design of the final proposed solution, Global Way-point LSTM Planner.

5.2.3 Complexity Analysis

Theorem 5.5: 2-dimensional Complexity

Worst case time complexity (2D environments; See Theorem 5.7 for proof):

$$\mathcal{O}(CAEOnlineLSTM) = \mathcal{O}(xo + nm \log nm + \min(max\_it, d))$$

Worst case space complexity (2D environments; See Theorem 5.8 for proof):

$$\mathcal{O}(CAEOnlineLSTM) = \mathcal{O}(1)$$

*where xo is the inflation pre-process from **DenseMap** (this is also present in $A*$ when it is run on a **DenseMap**), x is the inflation rate, o is the average obstacle size, d is the solution depth and n, m are the global image snapshot dimensions.*

Theorem 5.6: D-dimensional Complexity

Higher dimensional worst case time complexity:

$$\mathcal{O}(CAEOnlineLSTM) = \mathcal{O}(xo + n^D \log n^D + 64^D \log 64^D + \min(max\_it, d)\, 3^D)$$

Higher dimensional worst case space complexity:

$$\mathcal{O}(CAEOnlineLSTM) = \mathcal{O}(64^D)$$

*where xo is the inflation pre-process from **DenseMap** (this is also present in $A*$ when it is run on a **DenseMap**), x is the inflation rate, o is the average obstacle size, d is the solution depth, D is the dimension size and n^D is the global image snapshot dimensions*

Proof

In higher dimensions, the global snapshot is increased to 64^D, where D is the dimension number and the raycasting increase is inherited from the Online LSTM algorithm. $\mathcal{O}(CAE\_pass)$ becomes $\mathcal{O}(64^D \log 64^D)$ and $\mathcal{O}(scaling)$ becomes $\mathcal{O}(n^D \log n^D)$, where n is the average dimension length of the original map. Therefore, the time complexity becomes $\mathcal{O}(CAEOnlineLSTM) = \mathcal{O}(xo + n^D \log n^D + 64^D \log 64^D + \min(max\_it, d)\, 3^D)$ (latent space is still 100) and the space complexity becomes $\mathcal{O}(CAEOnlineLSTM) = \mathcal{O}(64^D)$. Moreover, the architecture has to be changed to support higher dimensions by using higher dimension convolutions or reshaping the input. We can notice that is quickly becomes an infeasible solution. A solution would be to decrease the global image size (64^D) to something smaller, but since we are dealing with real robots we can stop at 3D dimensions (which is still feasible with the current architecture and 3D convolutions). Furthermore, we can even keep the global image size to 64×64 in higher dimensions (by projecting the environment onto a 64×64 plane or using PCA). This procedure incurs a performance loss (due to the loss of data in the projection), but achieves better real world applicability.

Theorem 5.7: General 2-dimensional Worst Case Time Complexity

The general worst case time complexity (2D environments) is:

$$\begin{aligned}\mathcal{O}(CAEOnlineLSTM) = \mathcal{O}(xo + \mathcal{O}(pre\_process)+\\ \min(max\_it, d)(\mathcal{O}(collision\_detection) + \mathcal{O}(movement\_action)+\\ \mathcal{O}(feature\_extraction) + \mathcal{O}(lstm\_network\_pass)))\end{aligned}$$

where xo is the inflation complexity and d is the solution depth.

Proof

$\mathcal{O}(collision\_detection) = \mathcal{O}(movement\_action) = \mathcal{O}(feature\_extraction) = \mathcal{O}(1)$ *as in OnlineLSTM (for $\mathcal{O}(feature\_extraction)$ we extract the same old features and concatenate them with the cahced encoded map).*

$$\begin{aligned}\mathcal{O}(pre\_process) &= \mathcal{O}(\mathcal{O}(scaling) + \mathcal{O}(CAE\_pass))\\ \mathcal{O}(CAE\_pass) &= \mathcal{O}(4\mathcal{O}(convolution) + 4\mathcal{O}(deconvolution) + 2\mathcal{O}(linear))\end{aligned}$$

Kernel based convolutions are realised by making use of the Fast Fourier Transform (FFT) method which is $\mathcal{O}(n \log n)$. Because we encode a 64×64 image (the input is bounded) we can consider this to be $\mathcal{O}(CAE\_pass) = \mathcal{O}(1)$ (in practice, this step is quite expensive, but the input is bounded and it is only run once). We can scale using a kernel window (similar to max pooling), in our case scaling is two dimensional and depends on the image size. Therefore the scaling complexity becomes $\mathcal{O}(scaling) = \mathcal{O}(nm \log nm)$, where n is the width and m is the height of the original map image, so $\mathcal{O}(pre\_process) = \mathcal{O}(nm \log nm)$ (the encoded map is cached; encoding is $\mathcal{O}(1)$).

$\mathcal{O}(lstm\_network\_pass)$ takes a 112 sized input (100 encoded latent space and 12 old features), but it is still $\mathcal{O}(1)$ (in practice this takes longer than the OnlineLSTM network pass, but it is bounded).

Theorem 5.8: General 2-dimensional Worst Case Space Complexity

The general worst case space complexity (2D environments) is:

$$\begin{aligned}\mathcal{O}(CAEOnlineLSTM) = \mathcal{O}(\mathcal{O}(model\_size) + \mathcal{O}(scaling) + \mathcal{O}(map\_features)+\\ \mathcal{O}(cae\_network\_pass) + \mathcal{O}(lstm\_network\_pass) + \mathcal{O}(hidden\_cell\_state))\end{aligned}$$

Proof

$\mathcal{O}(model\_size) = \mathcal{O}(map\_featrues) = \mathcal{O}(lstm\_network\_pass) = \mathcal{O}(hidden\_cell\_state) = \mathcal{O}(1)$ *as in OnlineLSTM (for $\mathcal{O}(map\_features)$ we have 112 features which is still $\mathcal{O}(1)$).*

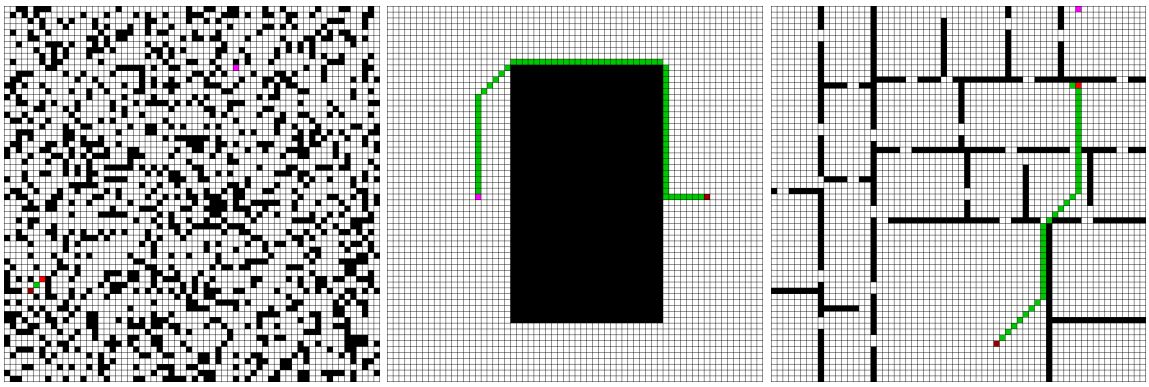
$\mathcal{O}(scaling)$ is bounded by the scaled image size which is 64×64. Therefore, $\mathcal{O}(scaling) = \mathcal{O}(1)$.

$\mathcal{O}(cae\_network\_pass)$ is bounded by the largest feature map in the network, which is $64 \times 32 \times 32$ and therefore $\mathcal{O}(cae\_network\_pass) = \mathcal{O}(1)$.

5.2.4 General Discussion

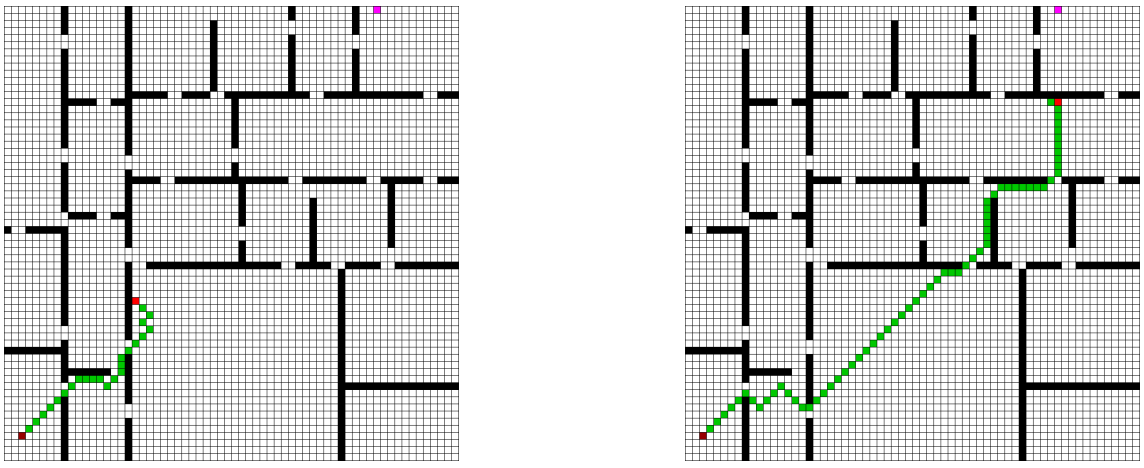
The CAE Online LSTM Planner shares the same advantages as the Online LSTM Planner. It is quite hard to argue if the implementation is better than A* in time and space complexity as the dimension increases, but it is better on 2D and 3D worlds. Lastly, it is a theoretically online solution, and therefore, it supports dynamic environments and partial knowledge. As mentioned above, in order to make it practically online, we have to feed a new global map snapshot at each time step. Because the extracted map is bounded, the worst case space complexity remains the same, but the worst case time complexity becomes $\mathcal{O}(CAEOnlineLSTM) = \mathcal{O}(xo + \min(max\_it, d)(nm \log nm))$ as we have to extract the global map snapshot and scale it at each time step.

The performance of the algorithm seems to be worse than the Online LSTM overall (See Figure 5.10), but it has a significant advantage. The algorithm is less greedy and attempts to go around obstacles more often, but it usually gets lost in the later stages (after a few iterations) (See Figure 5.11). This is quite important for the LSTM Bagging Planner from the next section.



(a) Goal not found (b) Goal found (c) Goal not found

Figure 5.10: CAE Online LSTM Planner trained on 10000 block maps paths. The map agent and goal position are identical to the one from figure 5.3



(a) Online LSTM Planner (b) CAE Online LSTM Planner

Figure 5.11: Online LSTM Planner vs CAE Online LSTM Planner

5.3 LSTM Bagging Planner

We currently have two solutions: Online LSTM Planner and CAE Online LSTM Planner. Both algorithms behave differently depending on the map layout. Moreover, we can notice the same behaviour variability when training the models on uncorrelated datasets. Thus, we introduce the LSTM Bagging Planner, which combines the performance of multiple models and picks the best behaving model depending on the map layout.

The idea was inspired by Ensemble Machine Learning methods [53]. Ensemble ML methods use multiple weak learners (other weak ML models such as Decision Stumps or Decision Trees) and, based on a majority voting consensus, outputs the classification choice. Ensemble ML is split into two categories: sequential and parallel. Sequential ensemble ML (e.g. AdaBoost) trains the weak learners sequentially on the same dataset. However, in order to make the learners independent of each other, the dataset is weighted. After training the first weak learner, if an example was wrongly classified, the weight associated with that particular sample will be increased (boosted) so that the next weak learner will classify the sample correctly (if an example was classified correctly, the weight is decreased). We are going to focus on the parallel ensemble methods which train all weak learners at the same time in parallel, but on different training datasets sampled from the original dataset. Thus, the weak learners are not correlated, and each one of them learns different features. Lastly, by having multiple uncorrelated weak learners, the voting procedure increases the accuracy of the predictions.

In our case, we use the Online LSTM and CAE Online LSTM Planners as our weak learners. Because the models can be trained on different generated training datasets (uniform random fill map, block map, house map) each weak learner learns how to behave in different environments (by extracting different features) and are uncorrelated.

The algorithm takes as input two arguments: $kernel\_names$ and max_it. $kernel\_names$ defines the names of the weak learners (the name is based on the model type and training dataset). max_it is used to initialise all weak learners ((CAE) Online LSTM Planner takes max_it as input) and has the same effect as the max_it from the (CAE) Online LSTM Planner. Weak learners (kernels) are loaded and executed in parallel on the map. If any/multiple kernels have found the goal we pick the one which has lower traversed length. Otherwise, we pick the kernel which has made the furthest progress (longest traversed length). If at any point we have multiple kernels that satisfy the best kernel conditions, we pick the one that occurs first in $kernel\_names$ (i.e. higher priority is given to the first kernels). Lastly, we follow the best kernel path (See Algorithm 11).

Algorithm 11 LSTM Bagging Planner

procedure LSTM-Bagging-Planner(M: (A, Os, G), $kernel\_names$, $max\_it = \infty$)
    $kernels \leftarrow$ load all $kernel\_names$ with maximum iterations max_it
    $results \leftarrow$ run all $kernels$ in parallel on M
    $best\_results \leftarrow []$

    for $result$ in $results$ **do**
        if $result$ has reached G **then**
            append $result$ to $best\_results$

    $best\_result \leftarrow \exists r1, \forall r2 \in best\_results.r1$ traversed length $\leq r2$ traversed length

    if $best\_result$ exists **then**
        execute $best\_result$ trace
        return

    $best\_result \leftarrow \exists r1, \forall r2 \in best\_results.r1$ traversed length $\geq r2$ traversed length

    execute $best\_result$ trace

5.3.1 Complexity Analysis

Theorem 5.9: 2-dimensional Complexity

Worst case time complexity (2D environments; See Theorem 5.10 for proof)):

$$\mathcal{O}(LSTMBagging) = \mathcal{O}(xo + nm \log nm + \min(max\_it, d))$$

Worst case space complexity (2D environments; See Theorem 5.11 for proof)):

$$\mathcal{O}(LSTMBagging) = \mathcal{O}(kernel\_number).$$

*where xo is the inflation pre-process from **DenseMap** (this is also present in A^* when it is run on a **DenseMap**), x is the inflation rate, o is the average obstacle size, d is the solution depth and n, m are the global image snapshot dimensions.*

Theorem 5.10: General 2-dimensional Worst Case Time Complexity

The general worst case time complexity (2D environments) is:

$$\begin{aligned}\mathcal{O}(LSTMBagging) = \mathcal{O}(kernel\_number\,(&\mathcal{O}(OnlineLSTM)+\\ &\mathcal{O}(CAEOnlineLSTM)))\end{aligned}$$

Proof

Because we run the kernels in parallel, the worst case time complexity becomes:

$$\begin{aligned}\mathcal{O}(LSTMBagging) &= \mathcal{O}(\mathcal{O}(OnlineLSTM) + \mathcal{O}(CAEOnlineLSTM))\\ &= \mathcal{O}(CAEOnlineLSTM)\\ &= \mathcal{O}(xo + nm \log nm + \min(max\_it, d))\end{aligned}$$

Theorem 5.11: General 2-dimensional Worst Case Space Complexity

The general worst case space complexity (2D environments) is:

$$\begin{aligned}\mathcal{O}(LSTMBagging) = \mathcal{O}(kernel\_number\,(&\mathcal{O}(OnlineLSTM)+\\ &\mathcal{O}(CAEOnlineLSTM)))\end{aligned}$$

Proof

$\mathcal{O}(\mathcal{O}(OnlineLSTM) + \mathcal{O}(CAEOnlineLSTM)) = \mathcal{O}(1)$.

Thus, the worst case space complexity becomes:

$$\mathcal{O}(LSTMBagging) = \mathcal{O}(kernel\_number).$$

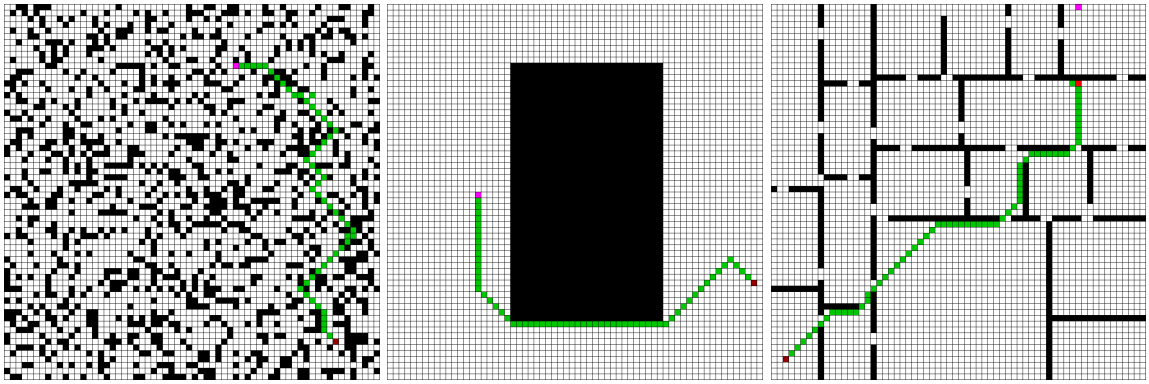
It should be noted that, in the actual implementation we clone the map $kernel\_number$ times and feed each clone to the associated kernel, but this is due to the design of the simulator and it can easily be changed to use the same initial map.

5.3.2 General Discussion

The algorithm inherits all properties from the Online LSTM Planner and CAE Online LSTM Planner. Therefore, the optimal path is relatively close to the A* performance, and it is even improved as we pick the shortest suggested path (if one is found).

The advantage of using this method is that the success rate of finding the goal is significantly increased compared to the Online LSTM and CAE Online LSTM Planners due to the reasons mentioned above (i.e. weak learners are uncorrelated). Moreover, it shares the same time complexity as a single kernel (See Figure 5.12) while inheriting the properties of all kernels.

The disadvantage is that, in practice, even if we run the kernels in parallel, there is a heavier resource load and the algorithm is usually slower than a single kernel. Lastly, there has been an implementation problem with *python* threads. *python* threads work differently than threads from other programming languages and usually do not improve the performance if the tasks are not interrupt driven (e.g. I/O tasks). Therefore, we have tried to replace threads with processes, but we could not do it as there was an issue when passing the data through the *pytroch* modules. The issue has not been fixed, and we are running the kernels sequentially, but it should be noted that switching to a parallel architecture is plausible (*pytroch* states that it supports multi-threading).



(a) Goal found. Best kernel: CAE Online LSTM (`house_10000`)

(b) Path is shorter. Best kernel: CAE Online LSTM (`uniform_random_fill_10000`, `block_map_10000`, `house_10000`)

(c) Goal not found, but is close to the goal. Best kernel: CAE Online LSTM (`uniform_random_fill_10000`, `block_map_10000`)

Figure 5.12: LSTM Bagging Planner with 10 kernels (kernel configuration is described in Table 5.1) paths. The maps are identical to the ones used in the failed Online LSTM Planner runs (See figure 5.4). The parenthesis value represents the training dataset on which the planner was trained on

Kernel	Training Data
CAE Online LSTM	`block_map_10000`
CAE Online LSTM	`uniform_random_fill_10000`
CAE Online LSTM	`house_10000`
CAE Online LSTM	`uniform_random_fill_10000_block_map_10000_house_10000`
CAE Online LSTM	`uniform_random_fill_10000_block_map_10000`
Online LSTM	`uniform_random_fill_10000`
Online LSTM	`block_map_10000`
Online LSTM	`house_10000`
Online LSTM	`uniform_random_fill_10000_block_map_10000`
Online LSTM	`uniform_random_fill_10000_block_map_10000_house_10000`

Table 5.1: LSTM Bagging Planner kernel configuration in priority order

5.4 Global Way-point LSTM Planner

After running empirical evaluations, we have noticed that the algorithms from previous sections get lost when the sequence size is too large. In order to overcome this issue, we introduce the Global Way-point LSTM Planner.

The Global Way-point LSTM Planner is a solution which makes use of two types of algorithms (kernels): local and global. The global kernel is responsible for producing a series of suggested global way-points and the local kernel is responsible for planning the trajectory between the way-points. For our scope, we are going to use one of the previous ML solutions (Online LSTM Planner, CAE Online LSTM Planner or LSTM Bagging Planner) as the global kernel. We can use any existing path planning solution for the local kernel, but we are going to use A* as it represents the main comparison frame of reference in Chapter 6 (Evaluation).

The algorithm accepts three inputs: the $local\_kernel$, $global\_kernel$ and gk_max_it. gk_max_it is used to initialise the $global\_kernel$ and is correlated with the distance between way-points. At each iteration, the $local\_kernel$ and $global\_kernel$ are reset. If the last way-point is not the goal itself, the local kernel is run one more time from the last way-point to the goal. As in the Online LSTM planner, we have included a fail-safe mechanism which breaks the main loop if a location is visited more than 5 times (See Algorithm 12).

Algorithm 12 Global Way-point LSTM Planner

```
1:  procedure GLOBAL-WAY-POINT-LSTM-PLANNER(M: (A, Os, G), local_kernel,
    global_kernel, gk_max_it)
2:
3:      Initialise local_kernel
4:      Initialise global_kernel with gk_max_it
5:      history_frequency ← {:}
6:
7:      while True do
8:          trace ← execute global_kernel
9:          way_point ← last trace position
10:         trace ← execute local_kernel from A to way_point
11:         follow trace
12:
13:         history_frequency[A] ← 1 (if no value) or history_frequency[A] + 1
14:
15:         if history_frequency[A] > 5 then
16:             break
17:
18:         if goal was reached then
19:             break
20:
21:     if goal was not reached then
22:         trace ← execute local_kernel
23:         follow trace
```

5.4.1 Complexity Analysis

Theorem 5.12: 2-dimensional Worst Case Complexity

The worst case time and space complexities (2D environments) are inherited from the local and global kernel:

$$\mathcal{O}(GlobalWaypointLSTM) = \mathcal{O}(\mathcal{O}(local\_kernel) + \mathcal{O}(global\_kernel))$$

It should be noted that the map is cloned when passed over to the kernels, but this is due to the simulator design and can easily be switched to passing a reference to the map instead.

Theorem 5.13: 2-dimensional Average Case Complexity

The average case time and space complexity (2D environments) is:

$$\mathcal{O}(GlobalWaypointLSTM) = \mathcal{O}(\frac{gk\_max\_it}{d}(\mathcal{O}(local\_kernel) + \mathcal{O}(global\_kernel)))$$

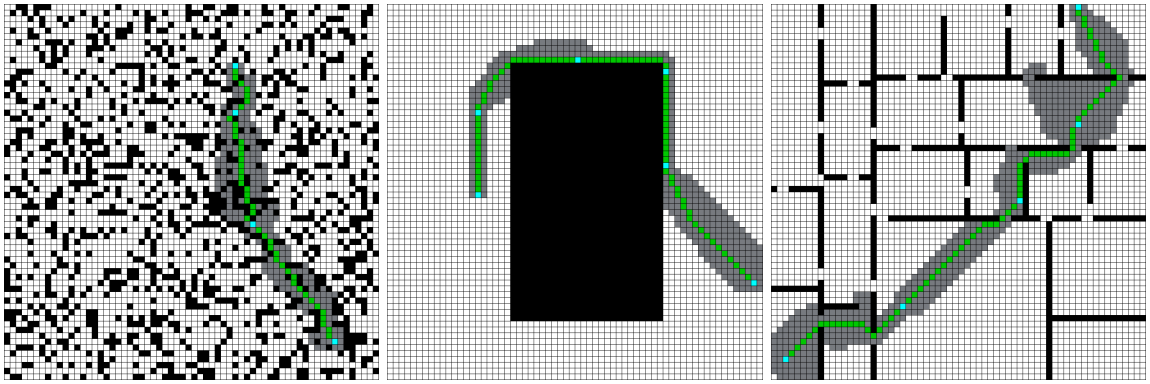
where d is the solution depth.

The complexity is equivalent to the average session complexity of the local and global kernel.

5.4.2 General Discussion

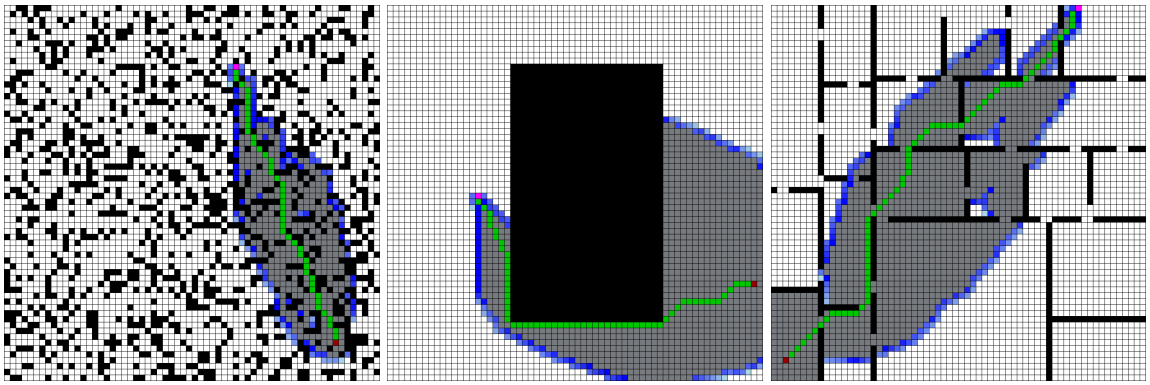
The major advantage of using this method is that we can transform the Online LSTM, CAE Online LSTM or LSTM Bagging Planners into way-point suggestion algorithms and thus, the sequence size is reduced, and the performance is improved. Moreover, we can use A* as our local kernel to ensure that we always find a path if we fail to place the last way-point on the goal. The local kernel can even be switched with a randomized kinodynamic solution (a planning solution which takes into account vehicle constraints such as velocity and torque) such as RRT. Because we use online global kernel solutions, we support partial information environments as the map can be queried again when we reach a new way-point. In order to support dynamic environments, we can switch the local kernel with an online planner that supports dynamic environments (we can even use Online LSTM, CAE Online LSTM or LSTM Bagging Planners themselves). Furthermore, the worst case time and space complexity of the algorithm is equal to A*: $\mathcal{O}(GlobalWaypointLSTM) = \mathcal{O}(\mathcal{O}(local\_kernel) + \mathcal{O}(global\_kernel)) = \mathcal{O}(\mathcal{O}(A*) + \mathcal{O}(global\_kernel)) = \mathcal{O}(A*)$. Lastly, the average time and space complexity is less than or equal to A*: $\mathcal{O}(GlobalWaypointLSTM) = \mathcal{O}(\frac{gk\_max\_it}{d}(\mathcal{O}(local\_kernel) + \mathcal{O}(global\_kernel))) = \mathcal{O}(\frac{gk\_max\_it}{d}(\mathcal{O}(A*) + \mathcal{O}(global\_kernel))) \leq \mathcal{O}(A*)$ (if the way-points are well distributed the performance is improved quite a lot) (See Figure 5.13 and Figure 5.14).

The major disadvantage to this approach is that the algorithm does not usually find the optimal path if the global kernel does not suggest optimal way-points, but because we are using the Online LSTM, CAE Online LSTM or LSTM Bagging Planners as the global kernel we receive a close to optimal path. Nonetheless, if we are using the A* local kernel, we have an optimal path between way-points.



(a) Goal found. Distance: 57.36 (b) Path is insignificantly shorter. Distance: 95.11 (c) Goal found. Distance: 99.30

Figure 5.13: Global Way-point LSTM Planner with local kernel A* and global kernel LSTM Bagging Planner (kernel configuration is described in Table 5.1). The maps are identical to the ones used in the failed Online LSTM runs (See figure 5.4). The produced path highly depends on the kernel priority. It should be noted that we display **all** memory used, but the average session memory is significantly smaller



(a) Path is a bit shorter. Distance: 54.04. A* memory uses more memory (b) Path is shorter. Distance: 68.38. A* uses more memory (c) Path is a bit shorter. Distance: 87.74. A* uses more memory

Figure 5.14: A* comparison to figure 5.13

Chapter 6

Evaluation

6.1 Methodology

In this chapter, we are going to present the empirical evaluation results. We will first inspect the synthetic generated training datasets, analyse the training procedure for the ML algorithms (Online LSTM Planner and CAE Online LSTM Planner), and lastly, we will run some experiments using the **Analyser** simple and complex evaluation procedures in order to review the performance of the proposed solution. We will analyse a variety of combinations of algorithms and training datasets described in Table 6.1.

Nr.	Planner	Training Data
0	A*	n/a
1	Online LSTM ([15])	`uniform_random_fill_10000`
2	Online LSTM	`block_map_10000`
3	Online LSTM	`house_10000`
4	Online LSTM	`uniform_random_fill_10000_block_map_10000`
5	Online LSTM	`uniform_random_fill_10000_block_map_10000_house_10000`
6	CAE Online LSTM	`uniform_random_fill_10000`
7	CAE Online LSTM ([1])	`block_map_10000`
8	CAE Online LSTM	`house_10000`
9	CAE Online LSTM	`uniform_random_fill_10000_block_map_10000`
10	CAE Online LSTM	`uniform_random_fill_10000_block_map_10000_house_10000`
11	LSTM Bagging	See Table 6.2
12	Global Way-point LSTM GK: CAE Online LSTM	`block_map_10000`
13	Global Way-point LSTM GK: Online LSTM	`uniform_random_fill_10000_block_map_10000_house_10000`
14	Global Way-point LSTM GK: LSTM Bagging (proposed solution)	See Table 6.2

Table 6.1: Evaluated algorithms and their respective training dataset. All algorithms are colour-coded: A* is light-grey, Online LSTM Planner is red (solution with same training dataset as [15] is darker red), CAE Online LSTM Planner is blue (solution with same training dataset as [1] is darker blue), LSTM Bagging Planner is orange, Global Way-point LSTM Planner is half cyan and half global kernel colour (e.g. Algorithm 14 has both cyan and orange colours as it uses the LSTM Bagging Planner as the GK)

In the future sections we will use a shorthand notation based on the algorithm index when talking about a particular solution, and we will reduce the number of inspected algorithms based on their performance. The proposed solution (Algorithm 14) has local kernel A* (Algorithm 0) and global kernel LSTM Bagging Planner (Algorithm 11). The kernel configuration for Algorithms 11 and 14 is described in Table 6.2. A* (Algorithm 0) is used as ground truth when training the ML models, and represents the comparison standard against all other algorithms. Algorithm 1 has the same

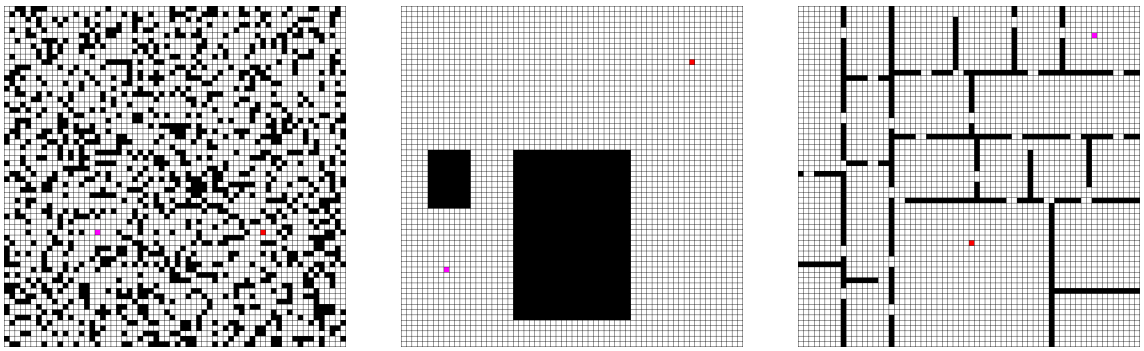
training dataset as [15], and Algorithm 7 has same training dataset as [1]. Algorithms 12 and 13 use Algorithms 7 and 5 (Algorithm 5 was chosen over 1 as it has better results) respectively as their global kernel. The reason behind evaluating Algorithms 12 and 13 is that we would like to inspect if the results of using the Global Way-point LSTM Planner with global kernel Online LSTM Planner and CAE Online LSTM Planner respectively achieve better results than the proposed solution (which uses the LSTM Bagging Planner as the global kernel).

Kernel	Training Data	Algorithm
CAE Online LSTM	`block_map_10000`	7
CAE Online LSTM	`uniform_random_fill_10000`	6
CAE Online LSTM	`house_10000`	8
CAE Online LSTM	`uniform_random_fill_10000_block_map_10000_house_10000`	9
CAE Online LSTM	`uniform_random_fill_10000_block_map_10000`	10
Online LSTM	`uniform_random_fill_10000`	1
Online LSTM	`block_map_10000`	2
Online LSTM	`house_10000`	3
Online LSTM	`uniform_random_fill_10000_block_map_10000`	4
Online LSTM	`uniform_random_fill_10000_block_map_10000_house_10000`	5

Table 6.2: LSTM Bagging Planner (Algorithms 11 and 14) kernel configuration in priority order

6.2 Synthetic Training Datasets Analysis

We have three types of generated synthetic maps: uniform random fill map, block map and house map. The map generation process is described in the Section 4.4 (Generator). The generated maps are described in Figure 6.1. The analysis of the training datasets can be found in Table 6.3.



(a) Uniform random fill map ([0.1, 0.3] obstacle fill rate range, 430 MB disk size)

(b) Block map ([0.1, 0.3] obstacle fill rate range, [1, 6] number of obstacles range, 400 MB disk size)

(c) House map ([8, 15] minimum room size range, [35, 45] maximum room size range, 230 MB disk size)

Figure 6.1: Synthetic generated maps (10000 samples of 64×64 dimension for each map)

Training Dataset Name	Path Available	Obstacles	Original Distance	Optimal Travel Distance	Disk Size
uniform_random_fill_10000	99.97%	19.95%	33.55	36.07	2GB
block_map_10000	99.99%	18.68%	34.35	38.92	2GB
house_10000	96.88%	8.25%	33.54	41.06	2GB

Table 6.3: Synthetic training datasets evaluation (all results are averaged)

6.3 Training Analysis

The training pipeline was run in the Google Colab environment (See Table 6.4).

Name	Value
GPU	GPU 0: NVIDIA Tesla T4 (2496 CUDA cores, Compute 3.7, 12GB(11.439GB Usable) GDDR5 VRAM)
CPU	Intel(R) Xeon(R) CPU @ 2.30GHz (No Turbo Boost, 1 Core, 2 Threads, 45 MB Cache)
RAM	~12.6 GB
Disk	~320 GB

Table 6.4: Google Colab machine specifications [54]

All training data was synthetically generated and we have a total of 30000 maps. It would have been better if we would have used a SLAM image dataset and convert it to internal maps as we would have had real-world data (e.g. the SUNCG dataset [55]). However, due to time restrictions, we had to generate the training data as it was faster. Moreover, we have control over the type of map and map parameters (obstacle fill rate, number of obstacles, min-max room size).

The training process is different from [15] and [1]. In [15], the authors use four types of maps: mazes with corridor having 4 steps width, mazes with corridor having 2 steps width, random filled for 25% map size and random filled for 40% of the map size. The number of maps is 5 for the training set and 3 for the validation set. Paths are generated using A* and are randomly selected from the available maps. The final training process results are: 0.3208 loss, 89.97% train accuracy and 88.60% validation accuracy. In [1], the authors state that a large amount of environmental images that contain randomly placed block obstacles (similar to our block map generated maps) has been used to train the CAE section, and 10 hard-codded maps with 10000 random paths for each map have been used to train the LSTM section. It should be noted that the paths are generated using RRT instead of A* and are modified by using a trajectory refinement method to extract high-quality paths. No training results have been provided.

Our training process uses three types of maps: uniform random fill map, block map and house map. We have a total of 30000 maps and separate datasets which use a subset of the maps based on the map type or the whole 30000 maps. We only generate a single path using A* for each map and we do not refine the trajectory. It should be noted that could have boosted the training dataset by sampling more paths from each map, but 30000 samples were enough (or a subset depending on the type of dataset). A major difference between our training procedure and the above training procedures is that we use a large number of environments for training the models and thus, we avoid over-fitting. Moreover, we can run experiments on a larger subset of maps and thus, we are more confident that our results are accurate and generalise well on different unseen environments.

6.3.1 Online LSTM Planner

The Online LSTM Planner has extra training configuration options described in Table 6.5.

Key	Pipeline Section	Type	Description
num_layers	Model Loading	`int`	Number of LSTM layers
lstm_input_size	Model Loading	`int`	LSTM input size
lstm_output_size	Model Loading	`int`	LSTM output size

Table 6.5: Online LSTM Planner extra training configuration options

The training parameters for the Online LSTM model are given in Table 6.6. All Online LSTM models use the same training configuration. We are going to evaluate each model training and report the following statistics: Training Loss (last training epoch loss), Validation Loss (last validation epoch loss), Evaluation Loss (test loss), Accuracy (evaluation accuracy), Precision (evaluation precision), Recall (evaluation recall), F1 (evaluation F1) and Confusion Matrix (confusion matrix of the predicted actions; actions are: 0 (go right), 1 (go top right), 2 (go top), 3 (go top left), 4 (go left), 5 (go bottom left), 6 (go bottom) and 7 (go bottom right) (See Figure 6.2)) (See Table 6.7). The *sklearn* library [56] has been used to produce all measurements and because we are using multi-class classification (we have 8 actions), we have used macro averaging for all statistics, which is a more "severe" measurement (outliers are equally punished and not weighted).

Key	Value
data_features	[distance_to_goal_normalized, raycast_8_normalized, direction_to_goal_normalized, agent_goal_angle]
data_labels	[next_position_index]
data_single_features	[]
data_single_labels	[]
epochs	100
loss	`CrossEntropyLoss`
optimizer	`lambda model: Adam(model.parameters(), lr=0.01)`
validation_ratio	0.2
test_ratio	0.2
save_name	`tile_by_tile`
training_data	See Table 6.1
batch_size	50
num_layers	2
lstm_input_size	12
lstm_output_size	8

Table 6.6: Online LSTM Planner training configuration

Model	Training Loss	Validation Loss	Evaluation Loss	Accuracy	Precision	Recall	F1	Confusion Matrix
1	0.033805	0.225089	0.141824	0.96	0.96	0.96	0.96	See Table 6.8
2	0.032614	0.105727	0.077589	0.98	0.97	0.97	0.97	See Table C.5
3	0.110707	0.430041	0.357634	0.91	0.91	0.91	0.91	See Table C.6
4	0.029944	0.090220	0.071301	0.97	0.97	0.97	0.97	See Table C.7
5	0.025989	0.114388	0.115875	0.92	0.92	0.92	0.92	See Table C.8

Table 6.7: Online LSTM Planner final training statistics

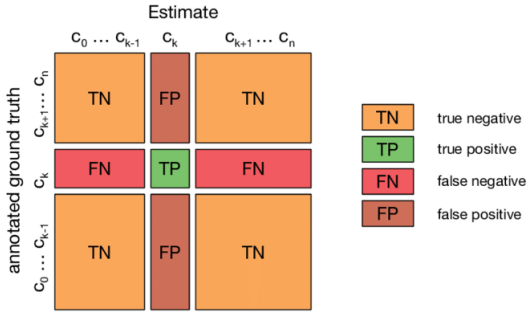
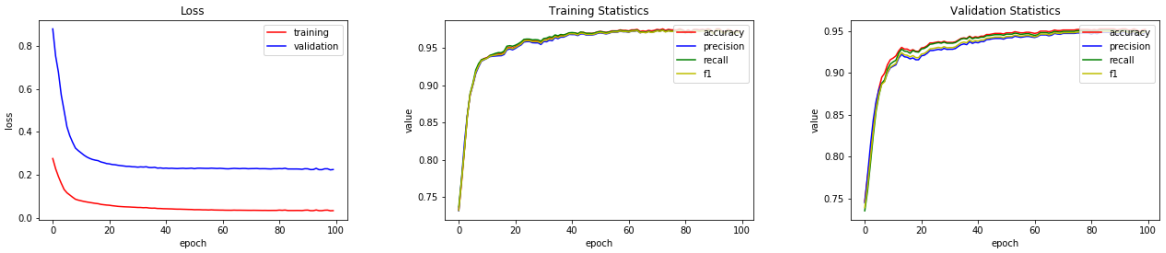


Figure 6.2: Confusion matrix visualisation [57]

		Predicted							
	Action	0	1	2	3	4	5	6	7
Actual	0	175	0	7	0	0	1	3	0
	1	4	207	2	0	0	0	0	0
	2	0	0	97	0	2	0	0	0
	3	0	0	0	202	1	0	0	0
	4	0	0	0	0	135	1	0	0
	5	0	0	1	0	0	227	5	1
	6	0	0	0	1	1	0	160	0
	7	7	0	1	0	0	3	5	164

Table 6.8: Confusion matrix for Algorithm 1



(a) Training/Validation Loss (b) Training Statistics (c) Validation Statistics

Figure 6.3: Training statistics for Algorithm 1

From the training results (See Figure 6.3) we can notice that the models have not experienced over-fitting. All models have higher accuracy than [15] (which has 89.97% training accuracy and 88.60% validation accuracy). Best trained models are Algorithms 2 and 4, but the other models are not far away. The confusion matrix (See Table 6.8) shows that we do not have a preferred action. For the full training statistics please refer to Appendix C.

6.3.2 CAE Online LSTM Planner

We will begin by assessing the CAE model training performance by inspecting the training loss graph, the feature maps and the latent space variation. After that, we will study the LSTM model training performance.

The CAE model includes extra training configuration options described in Table 6.9.

Key	Pipeline Section	Type	Description
use_mnist_instead	Data Pre-processing	`bool`	Describes if the MNIST dataset should be used instead (used for performance comparison)
mnist_size	Data Pre-processing	`Optional[int]`	Describes the size of the MNIST dataset (`None` for all)
with_skip_connections	Model Loading	`int`	Describes if the architecture should use skip connections
in_dim	Model Loading	`List[int]`	The size of the input image ([width, height])
latent_dim	Model Loading	`int`	The size of the latent vector

Table 6.9: CAE Online LSTM Planner: CAE model extra training configuration options

The default CAE Encoder used in the LSTM network from the CAE Online LSTM Planner is the CAE model trained on the same dataset as the LSTM network. The intuition behind this choice is that each CAE will learn separate features depending on the type of map. However, we are going to inspect only the CAE associated with the CAE Online LSTM Planner which has been trained on the same training dataset as [1] (Algorithm 7; block map). The training configuration for all CAEs is described in Table 6.10.

Key	Value
data_features	[]
data_labels	[]
data_single_features	[global_map]
data_single_labels	[global_map]
epochs	100
loss	`L1Loss`
optimizer	`lambda model: Adam(model.parameters(), lr=0.01)`
validation_ratio	0.2
test_ratio	0.2
save_name	`caelstm_section_cae`
training_data	[`uniform_random_fill_10000`, `block_map_10000`, `house_10000`]
batch_size	50
use_mnist_instead	`False`
mnist_size	`None`
with_skip_connections	`True`
in_dim	[64, 64]
latent_dim	100

Table 6.10: CAE Online LSTM Planner: CAE model training configuration

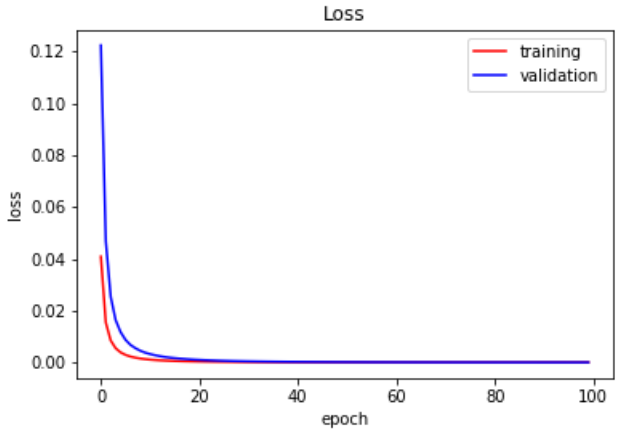


Figure 6.4: Training/Validation Loss for CAE model (Algorithm 7) (Train Loss: 0.000002, Validation Loss: 0.000005, Evaluation Loss: 0.000005)

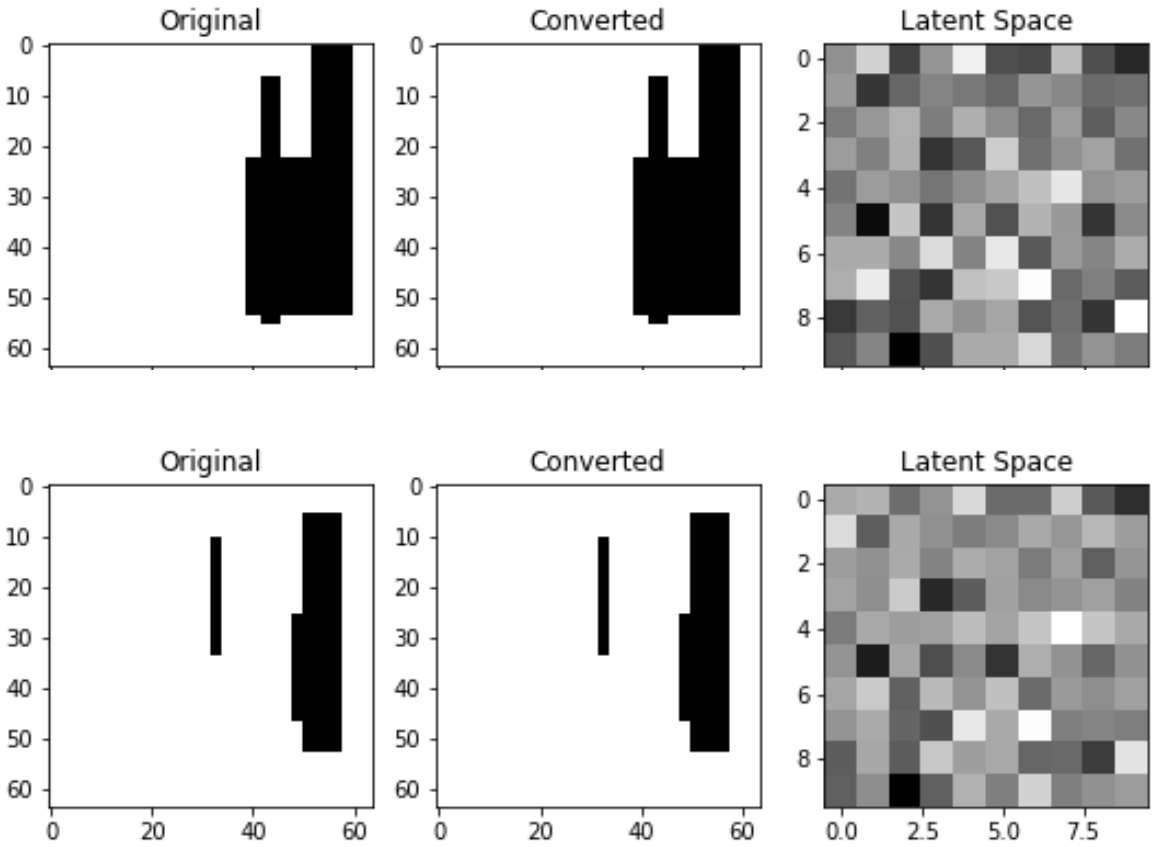


Figure 6.5: CAE model (Algorithm 7) network analysis

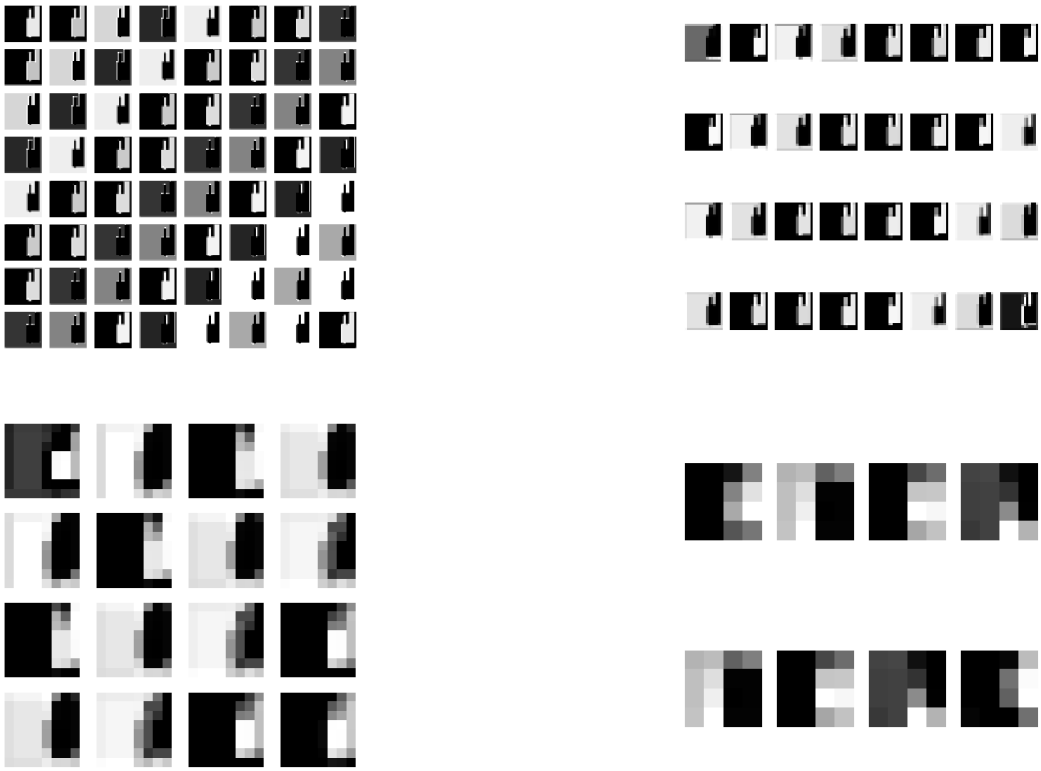


Figure 6.6: CAE model (Algorithm 7) first map from Figure 6.5 feature maps

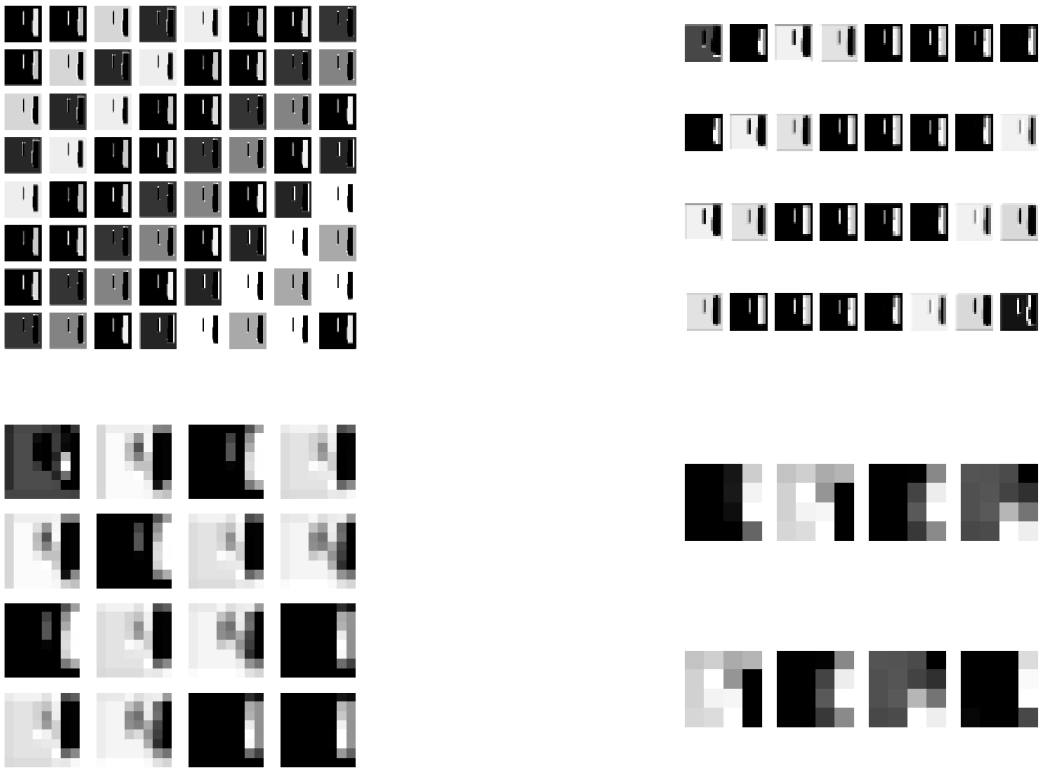


Figure 6.7: CAE model (Algorithm 7) second map from Figure 6.5 feature maps

The CAE has effectively 0 loss for all datasets (training, validation, evaluation) (See Figure 6.4). The converted representation is identical to the original image, due to the skip connections which help pass the lost compressed data over to the decoder (See Figure 6.5). However, if the losses are 0 it does not necessarily mean that the models have perfect performance. The model performance should be assessed based on the variability of the latent space and feature maps.

The scope of presenting the structure of the CAE network for different maps is to assess if the network actually learns the patterns in the data. This is achieved by inspecting how much the latent space varies between same type maps with different layouts.

After analysing the CAE network (See Figures 6.5, 6.6 and 6.7), we can notice that the block map CAE has reasonable latent space variability which is a sign of good generalisation and efficient feature extraction. Appendix C contains the full training analysis of the CAE Online LSTM Planner. By comparing the block map CAE against the uniform random fill and house CAEs, we have observed that the uniform random fill CAE has the worst performance due to the limited variability of the latent space. Intuitively, this happens because the uniform random fill map is very sparse and quite similar between other uniform random fill maps (even for the human eye is quite hard to spot the differences between this kind of maps), therefore less significant features can be extracted from the map. The other CAEs perform better because both types of maps (block and house maps) have a clear structure (block maps have block obstacles, house maps have rooms and doors), and thus, more significant features can be extracted from them.

We can also notice that the majority of feature maps have redundant features (e.g. the diagonal of all feature maps is almost identical) which is a sign that the CAE might be too deep for the current maps (See Figures 6.6 and 6.7). However, this does not affect our model performance as redundant data only increases the training process time.

The LSTM model includes extra training configuration options described in Table 6.11.

Key	Pipeline Section	Type	Description
custom_encoder	Model Loading	`Optional[str]`	Normally, the CAE with the same training dataset as this model is used, but custom_encoder specifies if another encoder should be used instead (when custom_encoder is `None`, it uses the default behaviour)

Table 6.11: CAE Online LSTM Planner: LSTM model extra training configuration options

All LSTM models share the same training configuration (See Table 6.12), with the exception of the epochs number for reasons described below. The LSTM section follows a similar pattern to the training analysis of the Online LSTM Planner and is summarised in Table 6.13.

Key	Value
data_features	[distance_to_goal_normalized, raycast_8_normalized, direction_to_goal_normalized, agent_goal_angle]
data_labels	[next_position_index]
data_single_features	[]
data_single_labels	[]
epochs	See Table C.9
loss	`CrossEntropyLoss`
optimizer	`lambda model: Adam(model.parameters(), lr=0.01)`
validation_ratio	0.2
test_ratio	0.2
save_name	`caelstm_section_lstm`
training_data	See Table 6.1
batch_size	50
custom_encoder	`None`
num_layers	2
lstm_input_size	112
lstm_output_size	8

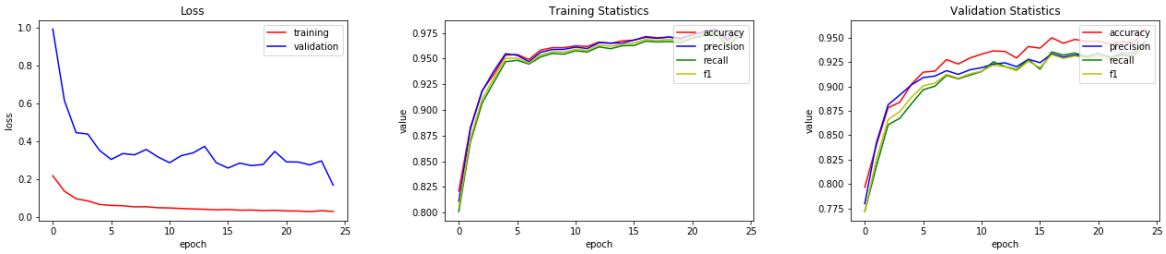
Table 6.12: CAE Online LSTM Planner: LSTM model training configuration

Model	Epochs	Training Loss	Validation Loss	Evaluation Loss	Accuracy	Precision	Recall	F1	CM
6	34	0.042641	0.153901	0.178480	0.96	0.96	0.95	0.95	See Table C.10
7	25	0.031109	0.169480	0.145604	0.96	0.96	0.95	0.95	See Table 6.14
8	45	0.146206	0.393990	0.610441	0.87	0.88	0.88	0.88	See Table C.12
9	37	0.019767	0.066242	0.097563	0.95	0.95	0.94	0.94	See Table C.13
10	50	0.033152	0.118695	0.096448	0.92	0.92	0.92	0.92	See Table C.14

Table 6.13: CAE Online LSTM Planner final training statistics (CM is short-hand for Confusion Matrix)

		Predicted							
	Action	0	1	2	3	4	5	6	7
Actual	0	294	1	0	0	4	0	0	0
	1	2	106	3	1	0	0	0	2
	2	0	0	235	0	0	0	11	0
	3	0	0	0	130	2	0	0	0
	4	5	0	0	1	186	0	0	0
	5	0	0	0	5	4	192	3	1
	6	0	0	3	0	0	1	396	2
	7	3	5	1	0	0	0	10	194

Table 6.14: Confusion matrix for Algorithm 7



(a) Training/Validation Loss (b) Training Statistics (c) Validation Statistics

Figure 6.8: Training statistics for Algorithm 7

We can notice from the training graph (See Figure 6.8) that the LSTM section model is prone to over-fitting and has to be manually early stopped to the best validation loss score. We can also observe that the majority of the CAE Online LSTM Planner training statistics (See Table 6.13) have lower validation loss than the ones from (See Table 6.7). However, most of the CAE Online LSTM Planner results have higher evaluation loss (which is also reflected in the accuracy metric) than the Online LSTM Planner results. This is a sign that the CAE Online LSTM Planner network has worse generalisation properties than the Online LSTM Planner network. But, the difference between them is really small and it does not has a significant impact on the overall performance of the network. The results from the confusion matrix (See Table 6.14) is identical to the results from the Online LSTM Planner: all actions are well distributed and there is no preferred action.

6.4 Experiments

In this section, we are going to run the **Analyser** pipeline which consists of two steps: simple and complex analysis. All experiments have been run on a MacBook Pro (Retina, 13-inch, Early 2015) (See Table 6.15).

Name	Value
Model	MacBook Pro (Retina, 13-inch, Early 2015)
GPU	Intel Iris Graphics 6100 1536 MB
CPU	2.7 GHz Intel Core i5
RAM	8 GB 1867 MHz DDR3
Disk	256 GB (SSD)

Table 6.15: Mac Book Pro specifications

The LSTM Bagging Planner defines custom statistic measures described in Table 6.16.

Name	Supported Algorithms	Description
Kernels	`LSTM Bagging Planner`	The name of all kernels in priority order
Pick Ratio	`LSTM Bagging Planner`	The "best behaving" kernel picked by the algorithm

Table 6.16: LSTM Bagging Planner statistic measure

The Global Way-point LSTM Planner defines custom display information (See Table 6.17) and custom statistic measures (See Table 6.18).

Display Name	Supported Algorithms	Description
Way-points	`Global Way-point LSTM Planner`	Displays the global kernel suggested way-points with cyan squares/circles (depending if the map is displayed as a grid or not)
Total Search Space	`Global Way-point LSTM Planner`	Displays the Total Search Space of the local kernel as dark-grey entities (local kernel has to be from A* family)

Table 6.17: Global Way-point LSTM Planner information displays

Name	Supported Algorithms	Description
GK Improvement	`Global Way-point LSTM Planner`	How much did the global kernel contributed to the final path. It is the local kernel travelled distance from the first way-point to the last way-point as a percentage from the total travelled distance
GK Steps	`Global Way-point LSTM Planner`	The total steps (movements) taken by the local kernel from the first way-point to the last way-point
GK Distance	`Global Way-point LSTM Planner`	The total local kernel travelled distance from the first way-point to the last way-point
GK Distance Left	`Global Way-point LSTM Planner`	The Euclidean distance between the last way-point and the goal
GK Calls	`Global Way-point LSTM Planner`	The number of global kernel calls
WP	`Global Way-point LSTM Planner`	The number of suggested way-points (we use this over the GK Calls as they are correlated)
WP In-Between Distance	`Global Way-point LSTM Planner`	The average Euclidean distance between way-points
Pick Ratio	`Global Way-point LSTM Planner`	The kernel pick percentages from the global kernel (global kernel has to be an LSTM Bagging Planner)
Search Space	`Global Way-point LSTM Planner`	The search space (visited set) that was used to find the path (without priority queue) (local kernel has to be from A* family)
Total Fringe	`Global Way-point LSTM Planner`	The left priority queue size, after the goal was found (local kernel has to be from A* family)
Total Search	`Global Way-point LSTM Planner`	Total Search = Search Space + Total Fringe (local kernel has to be from A* family)
Session Search Space	`Global Way-point LSTM Planner`	The average session search space (visited set) that was used to find the path (without priority queue) (local kernel has to be from A* family)
Session Fringe	`Global Way-point LSTM Planner`	The average session left priority queue size, after the goal was found (local kernel has to be from A* family)
Session Search	`Global Way-point LSTM Planner`	Session Search = Session Search Space + Session Fringe (local kernel has to be from A* family)

Table 6.18: Global Way-point LSTM Planner statistic measures

While analysing the algorithms, we will make use of the statistic measures defined in Tables 4.4, 6.16 and 6.18.

For each routine, we are going to produce four tables similar to Table 6.20. The first table describes the general performance of all algorithms compared to A* (Algorithm 0). It should be mentioned that the time statistics for the LSTM Bagging Planner (Algorithm 11) and the proposed solution (Algorithm 14) are significantly higher due to the parallelism issue (algorithms were run sequentially). The second table refers to the pick percentages of the internal kernels (See Table 6.2 for kernel reference) of the LSTM Bagging Planner (Algorithm 11) and the proposed solution (Algorithm 14) and is used to check the kernel pick distribution. The third table is associated with the Global Way-point LSTM Planner algorithms only and displays statistics which showcase the global kernel efficiency. For instance, if the WP In-Between Distance is higher, than we are more confident that the global kernel has proposed more efficient way-points. The final table displays the used memory space of the local kernel. We use four statistics: Total Search (the total local kernel memory (including fringe)), Total Fringe (the total local kernel fringe space), Session Search (the average local kernel session exploration (including fringe)) and Session Fringe (the average local kernel session fringe space).

The simple analysis stage is run on 30 maps (10 of each type) and the results are averaged (See Table 6.20). The complex analysis stage is run on 6 maps (3 generated and 3 hand crafted; See Figures 6.9, 6.10, 6.11, 6.12, 6.13 and 6.14). For each map we sample 50 random positions (agent and goal) and average the results of all 50 runs. At the end the overall results are reported in Table 6.27.

As a frame of reference the comprehensive results from [15] are shown in Table 6.19. The authors from [15] state that they have used 50 equally selected paths from 10 different maps for each environment type when evaluating their proposed solution. The authors from [1] state that they have achieved over 98% in 8 out of 10 environment maps. Moreover, the authors from [1] state that the generalisation property to unknown environments has been checked and confirmed. The results showed that the planner from [1] still has a high success rate in unknown maps, but path generation fails under environments which are significantly different from the trained environments. It should be mentioned that [1] associates a fail with a path that has found the goal, but overlaps obstacles. This is because their solution is offline and the network returns the full generated path, which consequently might collide with other obstacles. Therefore, the path generation fail under unknown environments does not theoretically contradict the robustness to unknown environments (since a path to the goal has been found). However, we should emphasise that our evaluation methods and results refer only to completely successful paths (i.e. collision-free trajectories). Lastly, the authors from [15] use the same collision-free evaluation constraint and therefore, we can relate and make comparisons against their results.

Planner	**Maps**	**Nr. of Paths**	**Nr. of Maps**	**Success Rate**	**Average Time (s)**	**Average Length**
A*	Dragon Age	50	10	100%	$3.37 \cdot 10^{-4}$	45.82
Online LSTM	Dragon Age	50	10	68% (I: -32%)	$1.69 \cdot 10^{-3}$	53.39 (I:-16.52%)
A*	Mazes with corridor 8 steps long	50	10	100%	$2.96 \cdot 10^{-4}$	50.90
Online LSTM	Mazes with corridor 8 steps long	50	10	26% (I: -74%)	$1.98 \cdot 10^{-3}$	61.71 (-21.24%)
A*	Random filled	50	10	100%	$2.84 \cdot 10^{-4}$	61.60
Online LSTM	Random filled	50	10	82% (I: -18%)	$2.04 \cdot 10^{-3}$	71.60 (I: -16.23%)

Table 6.19: [15] results (50 paths selected equally from 10 different maps for each environment type). The parenthesis value with prefix I: is the improvement rate against A*

Nr.	Success Rate	Distance	Time	Distance Left
0	93.33% (I: 0%)	39.55 (A*: 39.55) (I: 0%)	0.055s	2.36
1	46.67% (I: -49.99%)	35.09 (A*: 34.06) (I: -3.02%)	0.1342s	9.93
2	46.67% (I: -49.99%)	44.04 (A*: 36.77) (I: -19.77%)	0.1728s	12.52
3	60.0% (I: -35.71%)	45.44 (A*: 38.97) (I: -16.6%)	0.1328s	6.29
4	50.0% (I: -46.43%)	37.43 (A*: 35.07) (I: -6.73%)	0.1051s	9.45
5	70.0% (I: -25.0%)	43.88 (A*: 38.59) (I: -13.71%)	0.1166s	6.47
6	40.0% (I: -57.14%)	28.15 (A*: 27.14) (I: -3.72%)	0.1047s	14.24
7	43.33% (I: -53.57%)	37.96 (A*: 33.79) (I: -12.34%)	0.1399s	13.59
8	56.67% (I: -39.28%)	44.16 (A*: 39.03) (I: -13.14%)	0.1579s	10.52
9	56.67% (I: -39.28%)	42.19 (A*: 36.86) (I: -14.46%)	0.1513s	7.68
10	60.0% (I: -35.71%)	40.58 (A*: 36.29) (I: -11.82%)	0.1452s	7.38
11	83.33% (I: -10.71%)	43.22 (A*: 40.33) (I: -7.17%)	1.3548s	1.63
12	93.33% (I: 0%)	45.76 (A*: 39.55) (I: -15.7%)	0.407s	0.95
13	93.33% (I: 0%)	48.95 (A*: 39.55) (I: -23.77%)	0.3171s	0.62
14	93.33% (I: 0%)	44.94 (A*: 39.55) (I: -13.63%)	2.8274s	0.64

Nr.	Pick Ratio
11	[23.33, 26.67, 10.0, 3.33, 3.33, 6.67, 10.0, 10.0, 0.0, 6.67]%
14	[51.31, 21.02, 12.66, 3.43, 0.67, 0.0, 7.9, 1.67, 0.24, 1.11]%

Nr.	GK Improvement	GK Distance	GK Distance Left	WP	WP In-Between Distance
12	73.74%	30.97	11.08	5.03	10.12
13	86.51%	44.63	5.51	4.3	14.14
14	97.0%	49.23	1.3	4.1	15.91

Nr.	Total Search	Total Fringe	Session Search	Session Fringe
0	9.47%	2.86%	9.47%	2.86%
12	6.51% (I: 31.26%)	2.75% (I: 3.85%)	1.49% (I: 84.27%)	0.71% (I: 75.17%)
13	6.61% (I: 30.2%)	2.69% (I: 5.94%)	1.77% (I: 81.31%)	0.84% (I: 70.63%)
14	5.82% (I: 38.54%)	2.68% (I: 6.29%)	1.78% (I: 81.2%)	0.89% (I: 68.88%)

Table 6.20: **Analyser** simple analysis on 30 maps (10 uniform random fill maps, 10 block maps, 10 house maps). All experiments will have the same structure as this figure. The statistics are described in Tables 4.4, 6.16 and 6.18 and are all averaged. The parenthesis value containing the A*: prefix is the A* results that were run only on the filtered succeeded paths associated with the row run. The parenthesis value containing the I: prefix is the improvement ratio against A* (positive is better improvement and negative is degradation).

The simple analysis results (See Table 6.20) show that 2 maps (from 30) do not have a solution (given by A* success rate). We can notice that the best Online LSTM Planner was Algorithm 5 with a 70% (I: -25%) success rate and -13.71% distance improvement. The Online LSTM Planner which has been trained on the same type of map as [15] (Algorithm 1) has a poorer success rate of 46.67% (I: -49.99%), but higher distance improvement rate -3.02% (distance improvement is only computed on successful paths). By comparing these results with the results from [15] (See Table 6.19; Success Rate: 82% (I: -18%), Distance: (I: -16.23%)), we can observe that we have a general lower success rate, but higher distance improvement rate. The best CAE Online LSTM Planner is Algorithm 10 with 60.0% (-35.71%) success rate and -11.82% distance improvement. Algorithm 7 (which was trained on the same type of generated maps as [1]: block maps) has poorer success rate 43.33% (I: -53.57%) and lower distance improvement -12.34%. Overall, the results show that the Online LSTM and CAE Online LSTM have poorer success rate and insignificantly higher distance improvement rate than [15]). It should be noted that we use a higher number of environment maps (30) than [15] (10) and [1] (10) in order to ensure the generalisation property of our solutions. By using the LSTM Bagging Planner (Algorithm 11) we drastically increase the performance of the algorithms to 83.33% (I:-10.71%) (higher than [15]; 82% (I: -18%)) and decrease the distance improvement rate even further (-7.17% $<$ -16.23%). The Global Way-point LSTM

Planners (Algorithms 12, 13 and 14) have the same success rate as A* (due to the fact that we use A* as our local kernel) and lower overall distance rate improvement. The proposed solution (Algorithm 14) has a reasonable distance rate improvement of -13.63% compared to the other algorithms.

The pick ratio of the LSTM Bagging Planner (Algorithm 11) is well distributed compared to the proposed solution (Algorithm 14). The proposed solution (Algorithm 14) uses the kernel priority system described in Chapter 5 (Methods) (which can be noticed from the statistics; kernels lose pick percentage the further away they are from the head of the list). This shows that the priority of the kernels is essential to ensure the generation of a good path.

The third table shows that the proposed solution (Algorithm 14) has a really high GK Improvement rate compared to the other Global Way-point LSTM Planners (Algorithms 12 and 13), a high GK Distance (which is directly correlated to the GK Improvement), a great GK Distance Left, a lower WP count and a higher WP In-Between Distance. The GK Improvement rate and GK Distance show that the algorithm has suggested good way-points which accounted for the majority of the path journey. However, a higher GK Distance than the total travelled distance is a sign of oscillation which unnecessarily boosts the GK Improvement metric. However, because the difference between the GK Distance and the total Distance is small, the boost is reduced and almost insignificant. A low number of way-points with a higher way-point in-between distance is preferred over a larger number of way-points with a smaller way-point in-between distance due to the fact that we use the local kernel to optimise the path between the way-points. The GK Distance Left is really small which shows that the global kernel got lost near the goal. This metric is less useful when the environment resembles a maze as we might have to go around a long wall to get to the goal, but in our case, the environments are eligible for using the GK Distance Left metric.

The final table compares the used memory of the Global Way-point LSTM Planners against A*. We provide the total search and fringe space to get a general idea of the way-point efficiency, but only the session search and fringe space are taken into account as the memory is bounded by a single session. We can see that the search space is drastically reduced for all Global Way-point LSTM Planners compared to A*.

All algorithms have worse time performance than A*, but the LSTM Bagging Planner (Algorithm 11) and the proposed solution (Algorithm 14) have the highest times due to the implementation issues described in previous sections.

For the following complex analysis phase we are only going to mention the most noticeable changes (compared to the simple analysis) for the following algorithms: the Online LSTM Planner trained on the same type of maps as [15] (Algorithm 1), the CAE Online LSTM Planner trained on the same type of maps as [1] (Algorithm 7), the LSTM Bagging Planner (Algorithm 11) and the proposed solution (Algorithm 14). It should be noted that the generated maps have not been used in training and can be considered unknown as well.

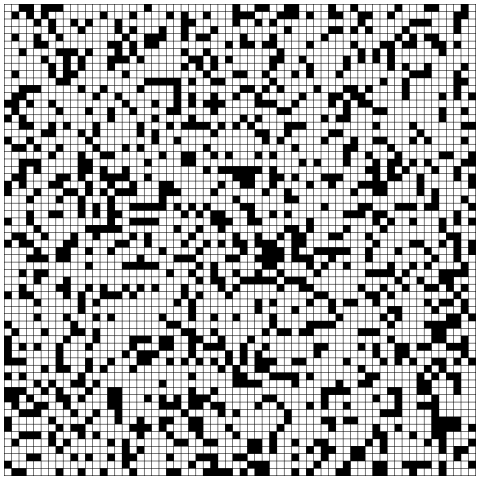


Figure 6.9: Uniform Random Fill Map

Nr.	Success Rate	Distance	Time	Distance Left
0	100.0% (I: 0%)	36.47 (A*: 36.47) (I: 0%)	0.0232s	0.0
1	44.0% (I: -56.0%)	26.64 (A*: 25.8) (I: -3.26%)	0.0578s	12.5
2	6.0% (I: -94.0%)	12.82 (A*: 11.71) (I: -9.48%)	0.0378s	24.12
3	24.0% (I: -76.0%)	28.72 (A*: 23.69) (I: -21.23%)	0.0928s	16.03
4	50.0% (I: -50.0%)	29.63 (A*: 28.64) (I: -3.46%)	0.0826s	9.33
5	44.0% (I: -56.0%)	27.09 (A*: 25.92) (I: -4.51%)	0.0753s	15.04
6	72.0% (I: -28.0%)	35.29 (A*: 33.45) (I: -5.5%)	0.1295s	8.02
7	8.0% (I: -92.0%)	18.0 (A*: 15.78) (I: -14.07%)	0.0748s	28.17
8	14.0% (I: -86.0%)	31.79 (A*: 26.27) (I: -21.01%)	0.1017s	18.97
9	42.0% (I: -58.0%)	26.87 (A*: 25.79) (I: -4.19%)	0.0922s	14.47
10	32.0% (I: -68.0%)	23.15 (A*: 22.21) (I: -4.23%)	0.0972s	14.94
11	74.0% (I: -26.0%)	35.8 (A*: 33.71) (I: -6.2%)	1.1013s	2.95
12	100.0% (I: 0%)	43.19 (A*: 36.47) (I: -18.43%)	0.5922s	0.0
13	100.0% (I: 0%)	38.23 (A*: 36.47) (I: -4.83%)	0.2458s	0.0
14	100.0% (I: 0%)	39.79 (A*: 36.47) (I: -9.1%)	2.4698s	0.0

Nr.	Pick Ratio
11	[4.0, 62.0, 8.0, 0.0, 4.0, 0.0, 0.0, 16.0, 4.0, 2.0]%
14	[17.0, 64.33, 6.67, 3.67, 2.0, 0.0, 1.0, 4.33, 1.0, 0.0]%

Nr.	GK Improvement	GK Distance	GK Distance Left	WP	WP In-Between Distance
12	37.98%	15.05	25.22	7.12	2.64
13	65.87%	22.21	13.64	4.74	8.13
14	97.61%	38.88	0.69	3.42	15.01

Nr.	Total Search	Total Fringe	Session Search	Session Fringe
0	7.38%	2.34%	7.38%	2.34%
12	6.56% (I: 11.11%)	2.34% (I: 0%)	1.02% (I: 86.18%)	0.4% (I: 82.91%)
13	5.24% (I: 29.0%)	2.03% (I: 13.25%)	1.17% (I: 84.15%)	0.5% (I: 78.63%)
14	4.69% (I: 36.45%)	2.07% (I: 11.54%)	1.43% (I: 80.62%)	0.68% (I: 70.94%)

Table 6.21: **Analyser** complex analysis on the uniform random fill map described in Figure 6.9 with 50 random agent/goal positions

- Algorithm 1 - same performance (the results are significantly worse than [15])
- Algorithm 7 - has really bad performance as it has been trained on block maps and the uniform random fill map has a significantly different structure
- Algorithm 11 - has worse success rate, same distance rate improvement and different picking pattern. The picking pattern is not well distributed and favours the algorithms which were trained on the uniform random fill map
- Algorithm 14 - has a good kernel improvement and better distance improvement rate

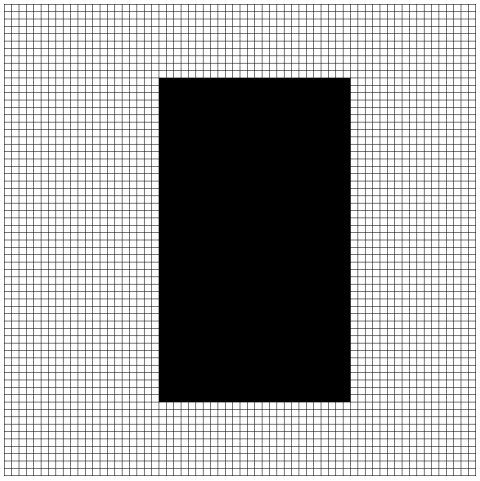


Figure 6.10: Block Map

Nr.	Success Rate	Distance	Time	Distance Left
0	100.0% (I: 0%)	42.22 (A*: 42.22) (I: 0%)	0.0502s	0.0
1	52.0% (I: -48.0%)	27.13 (A*: 27.07) (I: -0.22%)	0.0975s	14.45
2	98.0% (I: -2.0%)	48.12 (A*: 41.82) (I: -15.06%)	0.1539s	0.78
3	94.0% (I: -6.0%)	51.18 (A*: 41.05) (I: -24.68%)	0.1574s	2.34
4	100.0% (I: 0%)	46.34 (A*: 42.22) (I: -9.76%)	0.1331s	0.0
5	100.0% (I: 0%)	52.59 (A*: 42.22) (I: -24.56%)	0.1612s	0.0
6	42.0% (I: -58.0%)	22.62 (A*: 21.92) (I: -3.19%)	0.1171s	17.19
7	96.0% (I: -4.0%)	43.58 (A*: 41.64) (I: -4.66%)	0.1734s	1.58
8	82.0% (I: -18.0%)	42.86 (A*: 38.08) (I: -12.55%)	0.1667s	6.99
9	100.0% (I: 0%)	43.97 (A*: 42.22) (I: -4.14%)	0.1771s	0.0
10	100.0% (I: 0%)	53.48 (A*: 42.22) (I: -26.67%)	0.2055s	0.0
11	100.0% (I: 0%)	43.33 (A*: 42.22) (I: -2.63%)	1.8773s	0.0
12	100.0% (I: 0%)	46.17 (A*: 42.22) (I: -9.36%)	0.5092s	0.0
13	100.0% (I: 0%)	106.16 (A*: 42.22) (I: -151.44%)	0.4863s	0.0
14	100.0% (I: 0%)	49.25 (A*: 42.22) (I: -16.65%)	2.8345s	0.0

Nr.	Pick Ratio
11	[86.0, 0.0, 0.0, 4.0, 2.0, 0.0, 0.0, 0.0, 6.0, 2.0]%
14	[99.38, 0.62, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]%

Nr.	GK Improvement	GK Distance	GK Distance Left	WP	WP In-Between Distance
12	96.33%	44.11	1.58	3.7	15.73
13	91.14%	84.58	11.65	5.36	17.45
14	100.0%	49.25	0.0	3.78	16.41

Nr.	Total Search	Total Fringe	Session Search	Session Fringe
0	12.95%	3.02%	12.95%	3.02%
12	5.65% (I: 56.37%)	2.62% (I: 13.25%)	1.67% (I: 87.1%)	0.85% (I: 71.85%)
13	11.35% (I: 12.36%)	3.42% (I: -13.25%)	2.29% (I: 82.32%)	0.95% (I: 68.54%)
14	5.48% (I: 57.68%)	2.65% (I: 12.25%)	1.63% (I: 87.41%)	0.85% (I: 71.85%)

Table 6.22: **Analyser** complex analysis on the block map described in Figure 6.10 with 50 random agent/goal positions

- Algorithm 1 - better success rate due to the simplicity of the environment
- Algorithm 7 - has high success rate because it has been trained on the same type of environment, and good distance improvement rate (the results are similar to paper [1])
- Algorithm 11 - has same success rate as A* and really good distance improvement rate
- Algorithm 14 - GK Improvement is maximal, search space reduction is higher and the CAE Online LSTM Planner that was trained on the same type of maps is favoured

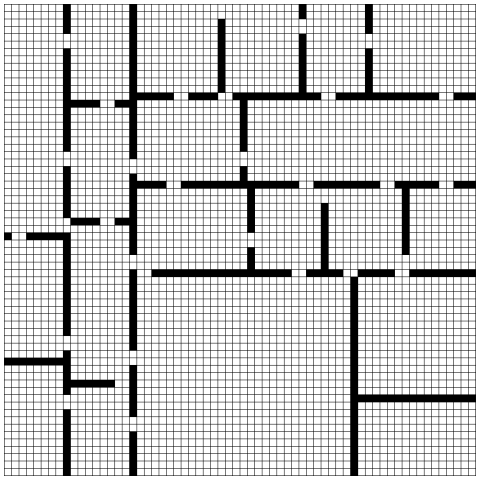


Figure 6.11: House Map

Nr.	Success Rate	Distance	Time	Distance Left
0	96.0% (I: 0%)	51.46 (A*: 51.46) (I: 0%)	0.0527s	2.44
1	6.0% (I: -93.75%)	18.89 (A*: 18.61) (I: -1.5%)	0.0394s	25.72
2	6.0% (I: -93.75%)	25.88 (A*: 24.38) (I: -6.15%)	0.0571s	19.22
3	72.0% (I: -25.0%)	56.14 (A*: 47.45) (I: -18.31%)	0.1198s	11.31
4	4.0% (I: -95.83%)	15.73 (A*: 15.73) (I: 0%)	0.0333s	17.32
5	40.0% (I: -58.33%)	53.35 (A*: 45.25) (I: -17.9%)	0.1218s	16.78
6	4.0% (I: -95.83%)	15.73 (A*: 15.73) (I: 0%)	0.068s	22.68
7	2.0% (I: -97.92%)	8.07 (A*: 8.07) (I: 0%)	0.0536s	29.76
8	46.0% (I: -52.08%)	56.98 (A*: 46.91) (I: -21.47%)	0.152s	19.01
9	4.0% (I: -95.83%)	15.73 (A*: 15.73) (I: 0%)	0.0685s	17.47
10	54.0% (I: -43.75%)	65.4 (A*: 55.07) (I: -18.76%)	0.236s	10.38
11	84.0% (I: -12.5%)	58.41 (A*: 50.28) (I: -16.17%)	1.3005s	4.14
12	96.0% (I: 0%)	57.91 (A*: 51.46) (I: -12.53%)	0.5204s	2.16
13	96.0% (I: 0%)	59.51 (A*: 51.46) (I: -15.64%)	0.2932s	2.2
14	96.0% (I: 0%)	58.51 (A*: 51.46) (I: -13.7%)	2.8492s	2.17

Nr.	Pick Ratio
11	[6.0, 2.0, 22.0, 24.0, 6.0, 0.0, 2.0, 38.0, 0.0, 0.0]%
14	[35.44, 20.73, 22.16, 3.17, 3.33, 1.0, 0.67, 13.17, 0.33, 0.0]%

Nr.	GK Improvement	GK Distance	GK Distance Left	WP	WP In-Between Distance
12	48.3%	26.29	25.91	7.54	3.65
13	71.9%	41.14	14.11	5.44	9.26
14	96.77%	58.79	3.61	4.22	16.36

Nr.	Total Search	Total Fringe	Session Search	Session Fringe
0	16.69%	4.94%	16.69%	4.94%
12	10.42% (I: 37.57%)	4.21% (I: 14.78%)	1.56% (I: 90.65%)	0.7% (I: 85.83%)
13	10.15% (I: 39.19%)	4.12% (I: 16.6%)	2.07% (I: 87.6%)	0.95% (I: 80.77%)
14	9.8% (I: 41.28%)	4.49% (I: 9.11%)	2.58% (I: 84.54%)	1.3% (I: 73.68%)

Table 6.23: **Analyser** complex analysis on the house map described in Figure 6.11 with 50 random agent/goal positions

- Algorithm 1 - performance is severely impacted
- Algorithm 7 - performance is severely impacted
- Algorithm 11 - good Success Rate, but worse distance improvement rate
- Algorithm 14 - great GK Improvement, same distance improvement rate and good search space reduction

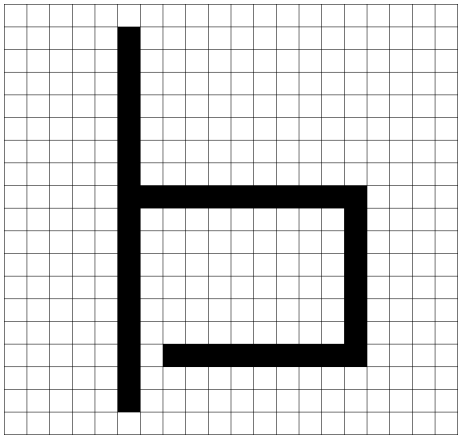


Figure 6.12: Hand Crafted Map 1

Nr.	Success Rate	Distance	Time	Distance Left
0	100.0% (I: 0%)	18.95 (A*: 18.95) (I: 0%)	0.0212s	0.0
1	22.0% (I: -78.0%)	6.65 (A*: 6.65) (I: 0%)	0.0256s	2.56
2	42.0% (I: -58.0%)	12.41 (A*: 11.54) (I: -7.54%)	0.0409s	2.72
3	38.0% (I: -62.0%)	15.13 (A*: 15.03) (I: -0.67%)	0.0526s	2.69
4	46.0% (I: -54.0%)	12.67 (A*: 12.13) (I: -4.45%)	0.0404s	1.08
5	60.0% (I: -40.0%)	16.97 (A*: 15.1) (I: -12.38%)	0.0458s	1.46
6	24.0% (I: -76.0%)	6.91 (A*: 6.91) (I: 0%)	0.0438s	3.24
7	44.0% (I: -56.0%)	12.12 (A*: 11.45) (I: -5.85%)	0.0693s	1.65
8	42.0% (I: -58.0%)	14.53 (A*: 13.44) (I: -8.11%)	0.0673s	4.39
9	14.0% (I: -86.0%)	5.71 (A*: 5.51) (I: -3.63%)	0.0476s	3.44
10	36.0% (I: -64.0%)	11.24 (A*: 9.75) (I: -15.28%)	0.0704s	1.7
11	72.0% (I: -28.0%)	18.62 (A*: 16.52) (I: -12.71%)	0.727s	0.99
12	100.0% (I: 0%)	23.74 (A*: 18.95) (I: -25.28%)	0.2857s	0.0
13	100.0% (I: 0%)	42.72 (A*: 18.95) (I: -125.44%)	0.2533s	0.0
14	100.0% (I: 0%)	46.86 (A*: 18.95) (I: -147.28%)	2.6125s	0.0

Nr.	Pick Ratio
11	[26.0, 2.0, 18.0, 4.0, 0.0, 0.0, 4.0, 20.0, 18.0, 8.0]%
14	[34.87, 2.0, 29.33, 5.76, 0.0, 0.0, 2.0, 13.68, 11.36, 1.0]%

Nr.	GK Improvement	GK Distance	GK Distance Left	WP	WP In-Between Distance
12	53.21%	8.42	1.65	4.24	4.7
13	88.32%	31.88	0.73	4.98	7.98
14	95.02%	41.05	0.65	5.2	8.93

Nr.	Total Search	Total Fringe	Session Search	Session Fringe
0	36.47%	7.18%	36.47%	7.18%
12	40.77% (I: -11.79%)	8.82% (I: -22.84%)	9.33% (I: 74.42%)	2.65% (I: 63.09%)
13	43.34% (I: -18.84%)	10.45% (I: -45.54%)	10.09% (I: 72.33%)	3.47% (I: 51.67%)
14	46.3% (I: -26.95%)	14.3% (I: -99.16%)	11.62% (I: 68.14%)	4.33% (I: 39.69%)

Table 6.24: **Analyser** complex analysis on the hand crafted map described in Figure 6.12 with 50 random agent/goal positions

- Algorithm 1 - performance is worse
- Algorithm 7 - performance is similar
- Algorithm 11 - Success Rate is a bit lower and distance improvement is worse as well
- Algorithm 14 - great GK Improvement, but severely impacted distance improvement rate (-147.28%). This is a sign that the global kernel goes around the whole obstacle until it finds the entrance. Other metrics are badly affected as well. Since Algorithm 11 has relatively similar performance, increasing the gk_max_it might improve the distance improvement rate

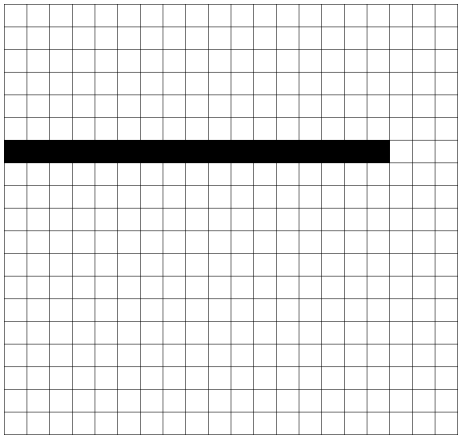


Figure 6.13: Hand Crafted Map 2

Nr.	Success Rate	Distance	Time	Distance Left
0	100.0% (I: 0%)	10.24 (A*: 10.24) (I: 0%)	0.0055s	0.0
1	78.0% (I: -22.0%)	7.5 (A*: 7.48) (I: -0.27%)	0.02s	2.04
2	80.0% (I: -20.0%)	8.05 (A*: 7.92) (I: -1.64%)	0.0206s	2.01
3	98.0% (I: -2.0%)	10.15 (A*: 10.09) (I: -0.59%)	0.0255s	0.16
4	78.0% (I: -22.0%)	7.48 (A*: 7.48) (I: 0%)	0.0196s	1.77
5	98.0% (I: -2.0%)	10.22 (A*: 10.17) (I: -0.49%)	0.0253s	0.32
6	78.0% (I: -22.0%)	7.89 (A*: 7.48) (I: -5.48%)	0.0429s	1.77
7	76.0% (I: -24.0%)	7.89 (A*: 7.36) (I: -7.2%)	0.0434s	1.97
8	96.0% (I: -4.0%)	10.16 (A*: 10.13) (I: -0.3%)	0.0481s	0.34
9	76.0% (I: -24.0%)	7.36 (A*: 7.36) (I: 0%)	0.0423s	1.92
10	100.0% (I: 0%)	10.31 (A*: 10.24) (I: -0.68%)	0.0485s	0.0
11	100.0% (I: 0%)	10.35 (A*: 10.24) (I: -1.07%)	0.4895s	0.0
12	100.0% (I: 0%)	10.43 (A*: 10.24) (I: -1.86%)	0.1472s	0.0
13	100.0% (I: 0%)	10.87 (A*: 10.24) (I: -6.15%)	0.0548s	0.0
14	100.0% (I: 0%)	10.24 (A*: 10.24) (I: 0%)	0.5487s	0.0

Nr.	Pick Ratio
11	[72.0, 6.0, 20.0, 2.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]%
14	[74.0, 8.0, 16.0, 2.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]%

Nr.	GK Improvement	GK Distance	GK Distance Left	WP	WP In-Between Distance
12	87.29%	8.0	1.97	3.02	5.64
13	100.0%	10.87	0.0	2.1	8.03
14	100.0%	10.24	0.0	2.06	8.12

Nr.	Total Search	Total Fringe	Session Search	Session Fringe
0	16.25%	8.17%	16.25%	8.17%
12	13.32% (I: 18.03%)	7.85% (I: 3.92%)	5.13% (I: 68.43%)	3.22% (I: 60.59%)
13	15.88% (I: 2.28%)	7.59% (I: 7.1%)	7.93% (I: 51.2%)	4.13% (I: 49.45%)
14	14.94% (I: 8.06%)	7.49% (I: 8.32%)	7.71% (I: 52.55%)	4.16% (I: 49.08%)

Table 6.25: **Analyser** complex analysis on the hand crafted map described in Figure 6.13 with 50 random agent/goal positions

- The performance of all algorithms is greatly boosted as the hand crafted map is quite simple in layout, but was used to test the ability of the algorithm to go around long walls which was an issue in paper [15]

- Algorithm 14 - has 100% GK Improvement and 0% distance improvement rate, which matches the A* performance exactly

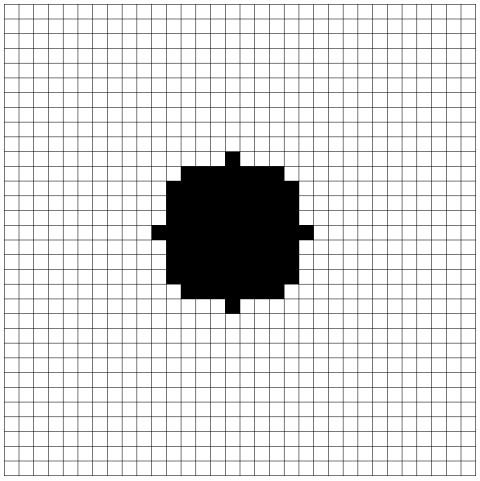


Figure 6.14: Hand Crafted Map 3

Nr.	Success Rate	Distance	Time	Distance Left
0	100.0% (I: 0%)	22.66 (A*: 22.66) (I: 0%)	0.0182s	0.0
1	76.0% (I: -24.0%)	20.19 (A*: 20.09) (I: -0.5%)	0.0587s	4.88
2	98.0% (I: -2.0%)	26.81 (A*: 22.59) (I: -18.68%)	0.0718s	0.42
3	74.0% (I: -26.0%)	23.72 (A*: 20.14) (I: -17.78%)	0.0697s	5.37
4	84.0% (I: -16.0%)	21.48 (A*: 21.21) (I: -1.27%)	0.0618s	3.35
5	96.0% (I: -4.0%)	23.64 (A*: 22.34) (I: -5.82%)	0.0562s	0.84
6	86.0% (I: -14.0%)	21.52 (A*: 21.25) (I: -1.27%)	0.0835s	2.94
7	76.0% (I: -24.0%)	23.38 (A*: 20.25) (I: -15.46%)	0.0807s	4.83
8	72.0% (I: -28.0%)	20.67 (A*: 19.71) (I: -4.87%)	0.0729s	6.11
9	90.0% (I: -10.0%)	22.5 (A*: 22.01) (I: -2.23%)	0.0762s	2.1
10	78.0% (I: -22.0%)	20.97 (A*: 20.49) (I: -2.34%)	0.0728s	4.23
11	100.0% (I: 0%)	23.27 (A*: 22.66) (I: -2.69%)	0.6844s	0.0
12	100.0% (I: 0%)	23.96 (A*: 22.66) (I: -5.74%)	0.164s	0.0
13	100.0% (I: 0%)	23.2 (A*: 22.66) (I: -2.38%)	0.1004s	0.0
14	100.0% (I: 0%)	22.66 (A*: 22.66) (I: 0%)	1.0798s	0.0

Nr.	Pick Ratio
11	[50.0, 38.0, 2.0, 0.0, 0.0, 2.0, 0.0, 4.0, 0.0, 4.0]%
14	[63.0, 31.0, 2.0, 2.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0]%

Nr.	GK Improvement	GK Distance	GK Distance Left	WP	WP In-Between Distance
12	81.99%	18.36	4.83	3.3	11.22
13	96.83%	22.14	0.84	2.6	14.37
14	100.0%	22.66	0.0	2.44	14.98

Nr.	Total Search	Total Fringe	Session Search	Session Fringe
0	19.46%	9.21%	19.46%	9.21%
12	14.93% (I: 23.28%)	7.75% (I: 15.85%)	5.43% (I: 72.1%)	3.12% (I: 66.12%)
13	15.24% (I: 21.69%)	7.87% (I: 14.55%)	6.41% (I: 67.06%)	3.63% (I: 60.59%)
14	14.57% (I: 25.13%)	7.53% (I: 18.24%)	6.35% (I: 67.37%)	3.59% (I: 61.02%)

Table 6.26: **Analyser** complex analysis on the hand crafted map described in Figure 6.14 with 50 random agent/goal positions

- The analysis has similar characteristics to the fifth complex analysis (See Table 6.25)

- Algorithm 14 - has 100% GK Improvement and 0% distance improvement rate, which matches the A* performance exactly

Nr.	Success Rate	Distance	Time	Distance Left
0	99.33% (I: 0%)	30.19 (A*: 30.19) (I: 0%)	0.0284s	0.41
1	46.33% (I: -53.36%)	17.85 (A*: 17.67) (I: -1.02%)	0.0519s	10.36
2	55.0% (I: -44.63%)	26.49 (A*: 23.17) (I: -14.33%)	0.0789s	8.21
3	66.67% (I: -32.88%)	32.17 (A*: 27.24) (I: -18.1%)	0.0883s	6.32
4	60.33% (I: -39.26%)	25.28 (A*: 23.87) (I: -5.91%)	0.0723s	5.48
5	73.0% (I: -26.51%)	29.39 (A*: 25.62) (I: -14.72%)	0.0797s	5.74
6	51.0% (I: -48.66%)	20.22 (A*: 19.51) (I: -3.64%)	0.0853s	9.31
7	50.33% (I: -49.33%)	24.02 (A*: 22.32) (I: -7.62%)	0.0988s	11.33
8	58.67% (I: -40.93%)	27.43 (A*: 24.44) (I: -12.23%)	0.0988s	9.3
9	54.33% (I: -45.3%)	25.32 (A*: 24.5) (I: -3.35%)	0.1s	6.57
10	66.67% (I: -32.88%)	31.73 (A*: 27.2) (I: -16.65%)	0.1237s	5.21
11	88.33% (I: -11.07%)	31.3 (A*: 29.09) (I: -7.6%)	1.0343s	1.35
12	99.33% (I: 0%)	34.07 (A*: 30.19) (I: -12.85%)	0.3688s	0.36
13	99.33% (I: 0%)	46.7 (A*: 30.19) (I: -54.69%)	0.2386s	0.37
14	99.33% (I: 0%)	37.74 (A*: 30.19) (I: -25.01%)	2.0605s	0.36

Nr.	Pick Ratio
11	[40.67, 18.33, 11.67, 5.67, 2.0, 0.33, 1.0, 13.0, 4.67, 2.67]%
14	[53.95, 21.11, 12.69, 2.77, 0.89, 0.33, 0.78, 5.2, 2.12, 0.17]%

Nr.	GK Improvement	GK Distance	GK Distance Left	WP	WP In-Between Distance
12	67.52%	20.04	10.19	4.82	7.26
13	85.68%	35.47	6.83	4.2	10.87
14	98.23%	36.81	0.82	3.52	13.3

Nr.	Total Search	Total Fringe	Session Search	Session Fringe
0	18.21%	5.82%	18.21%	5.82%
12	15.31% (I: 15.93%)	5.61% (I: 3.61%)	4.04% (I: 77.81%)	1.83% (I: 68.56%)
13	16.91% (I: 7.14%)	5.93% (I: -1.89%)	5.01% (I: 72.49%)	2.28% (I: 60.82%)
14	16.01% (I: 12.08%)	6.43% (I: -10.48%)	5.24% (I: 71.22%)	2.49% (I: 57.22%)

Table 6.27: **Analyser** complex analysis overall results

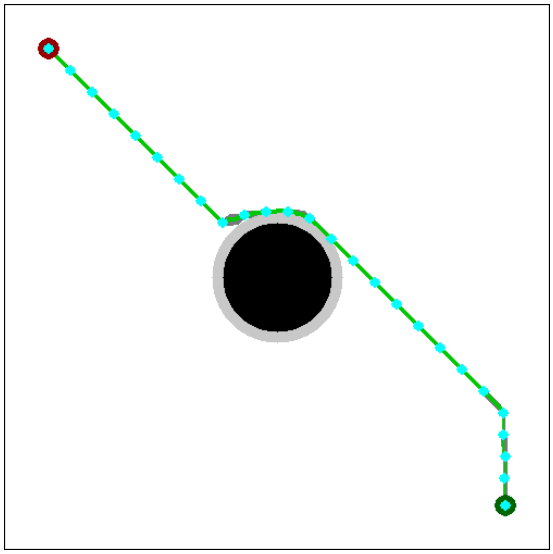
The overall results from the complex analysis have similar characteristics to the ones in the simple analysis (See Table 6.27). Some noticeable characteristics are:

- Algorithm 11 - has the same distance improvement rate, has a higher Success Rate and a stronger picking preference towards the block map CAE Online LSTM which can be argued by the fact that most of the maps have a similar structure to the block map

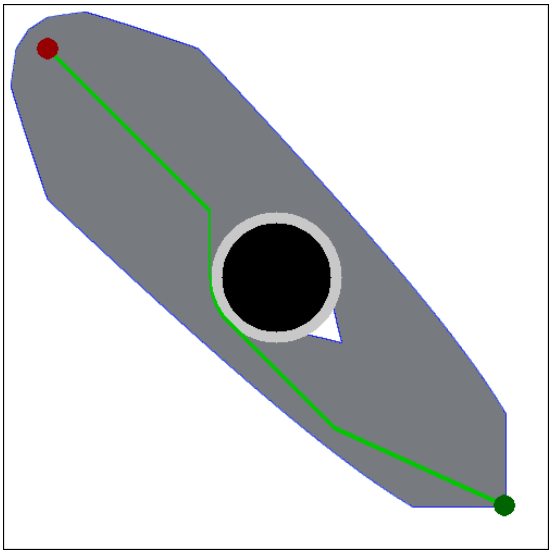
- Algorithm 14 - has affected distance improvement rate (-25.01% > -13.63%; which is still reasonable compared to the A* performance), has the same GK Improvement rate, almost no oscillation has been spotted, has the same picking behaviour, has the same way-point suggestion efficiency and the Session Search memory reduction is still significantly high compared to A*

It should be mentioned that we have run significantly more experiments in the overall complex analysis phase ($6 \times 50 = 300$; 300 paths from 6 maps) than [15] (50 paths from 10 maps) and [1] (10 paths from 10 maps). Thus, we can say that our experiments are valid. Moreover, because the results from the overall complex analysis phase have similar characteristics to the results from the simple analysis phase, we prove the robustness to unknown environments property (i.e. we show that the performance of the algorithm on trained maps is similar to the one on unknown maps).

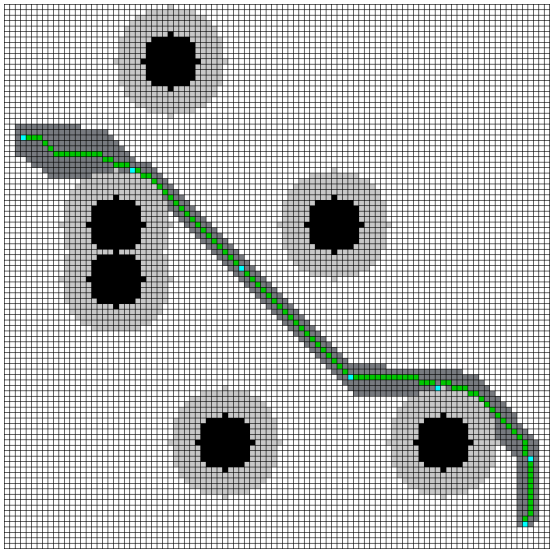
Lastly, Figure 6.15 contains more example runs of the Global Way-point LSTM Planner against A* which emphasise the memory/optimal distance trade-off.



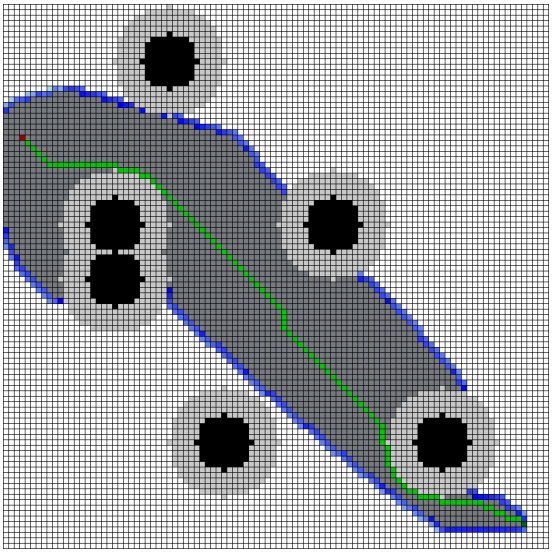
(a) Global Way-point LSTM Planner



(b) A*



(c) Global Way-point LSTM Planner



(d) A*

Figure 6.15: Global Way-point LSTM Planner and A* examples which showcase the memory/optimal distance trade-off

6.5 Path Planning on Real-world Maps

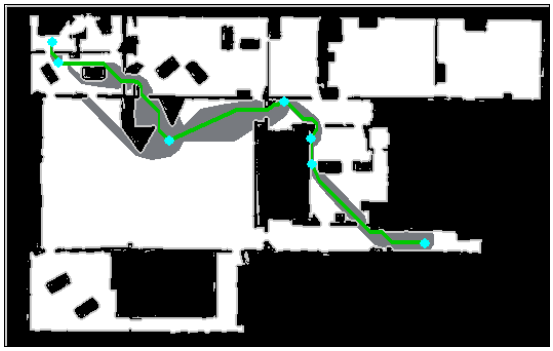
We have collected three real-world occupancy grid maps produced by SLAM sensors and converted them into internal maps. The maps have been used in the following works: [58], [59] and [60].

Figures 6.16, 6.17 and 6.18 highlight the performance of the Global Way-point LSTM Planner against A* on the real-world maps. Some runs achieve great results while others completely fail (in the sense that we fail to place the last global way-point on the goal). Furthermore, we can notice that the algorithm maintains the same behaviour across different environments which confirms the robustness to unknown environments property. This is intuitively correct, as we have used Machine Learning methods to find the path, and thus, we inherit the generalisation properties.

It should be noted that the run-times where significantly higher for the Global Way-point LSTM Planner due to the parallelising issue. We have also varied the global kernel max iterations to highlight the importance of choosing a proper argument. As a general rule, the number of iterations should be proportional to the size of the map.

(a) Global Way-point LSTM Planner (max iterations 80)

(b) A*



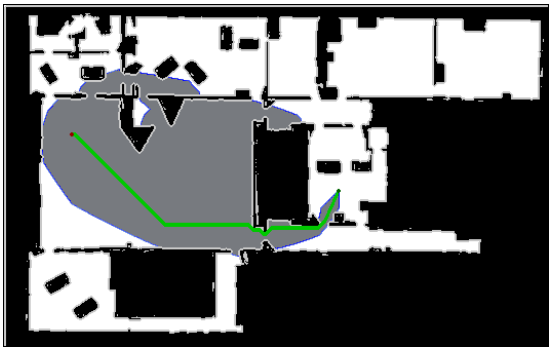
(c) Global Way-point LSTM Planner (max iterations 100)

(d) A*

(e) Global Way-point LSTM Planner (max iterations 100)

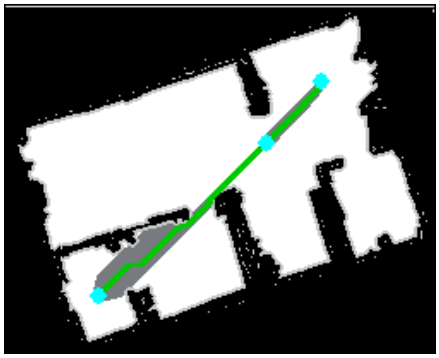
(f) A*

(g) Global Way-point LSTM Planner (max iterations 100)

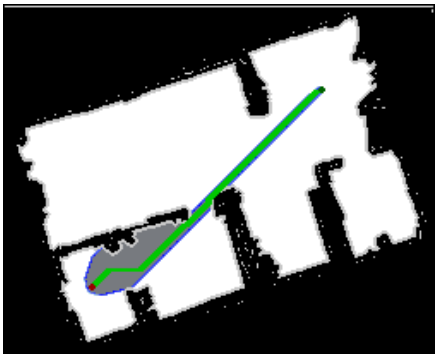


(h) A*

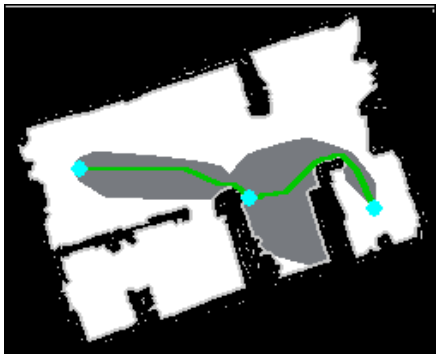
Figure 6.16: Global Way-point LSTM Planner vs A* runs on real-world occupancy grid maps [58]



(a) Global Way-point LSTM Planner (max iterations 100)

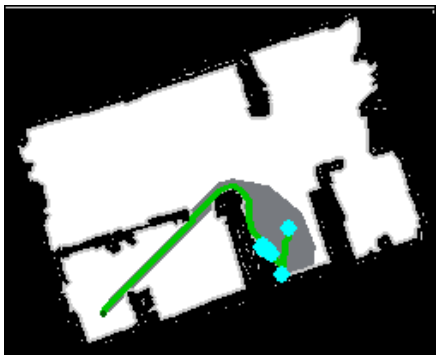


(b) A*

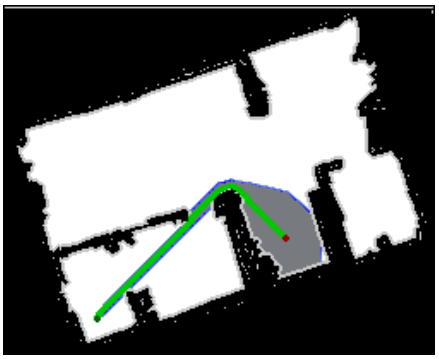


(c) Global Way-point LSTM Planner (max iterations 100)

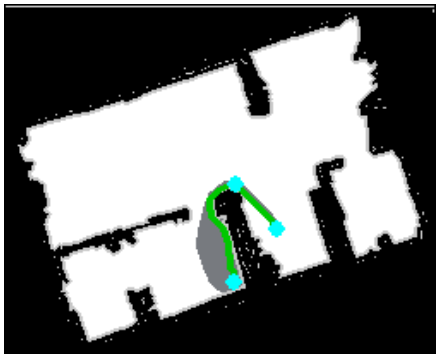
(d) A*



(e) Global Way-point LSTM Planner (max iterations 100)



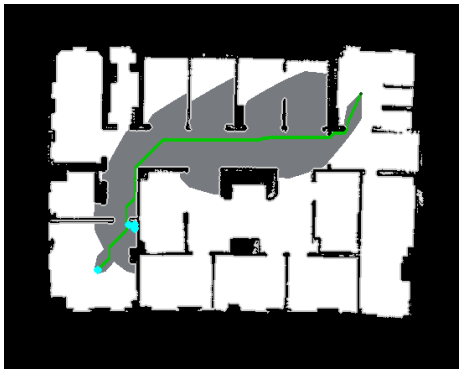
(f) A*



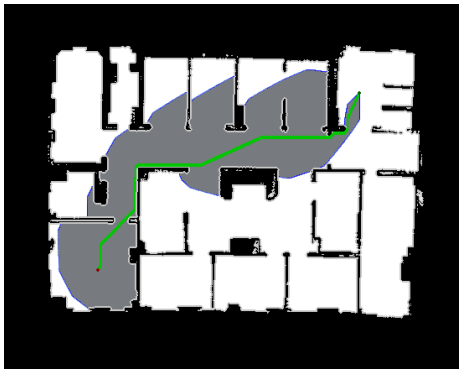
(g) Global Way-point LSTM Planner (max iterations 100)

(h) A*

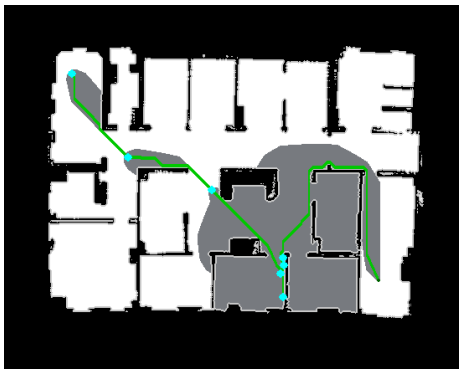
Figure 6.17: Global Way-point LSTM Planner vs A* runs on real-world occupancy grid maps [59]



(a) Global Way-point LSTM Planner (max iterations 100)

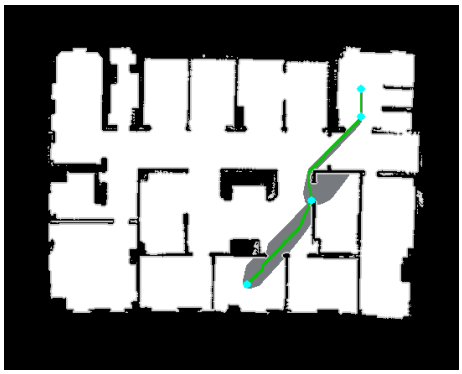


(b) A*

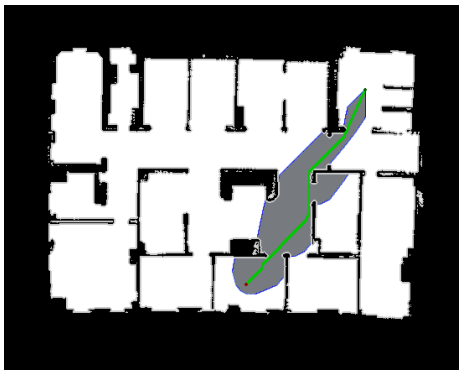


(c) Global Way-point LSTM Planner (max iterations 100)

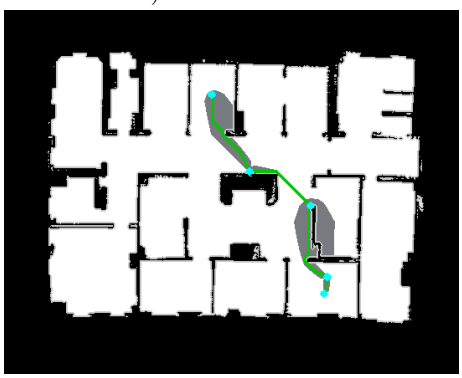
(d) A*



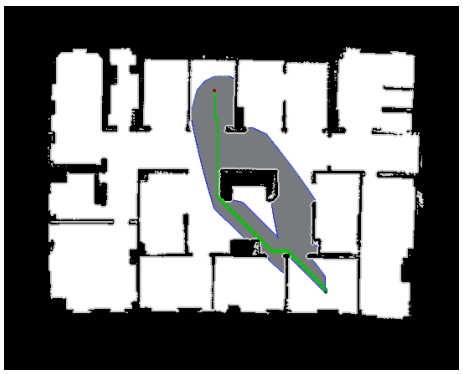
(e) Global Way-point LSTM Planner (max iterations 100)



(f) A*



(g) Global Way-point LSTM Planner (max iterations 100)

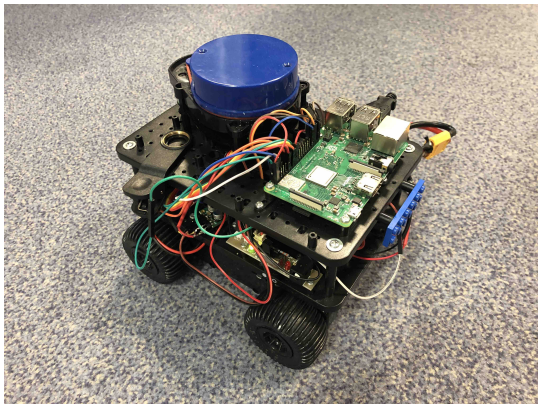


(h) A*

Figure 6.18: Global Way-point LSTM Planner vs A* runs on real-world occupancy grid maps [60]

6.6 Path Planning on Real-world Robot

The final evaluation will be run on a real-world robot at Imperial College London (See Figure 6.19). We will use a basic 4-wheeler robot with a YDLidar sensor attached at the top. The YDLidar sensor is a 360-degree two-dimensional distance measurement device which produces a SLAM output image scan [61]. The robot (Perceptbot) was build as a novel development platform for robotics by a group of students [62]. The robot has support for multiple hardware attachments (such as an external camera and an Intel Neural Compute Stick for Machine Learning), but we have only used the YDLidar sensor. The motherboard of the robot is a Raspberry Pi circuit board [63] which is running *Raspbian* [64], and makes use of the *ROS* library for motion controlling.



(a) Robot (Perceptbot) with YDLidar Sensor [62]

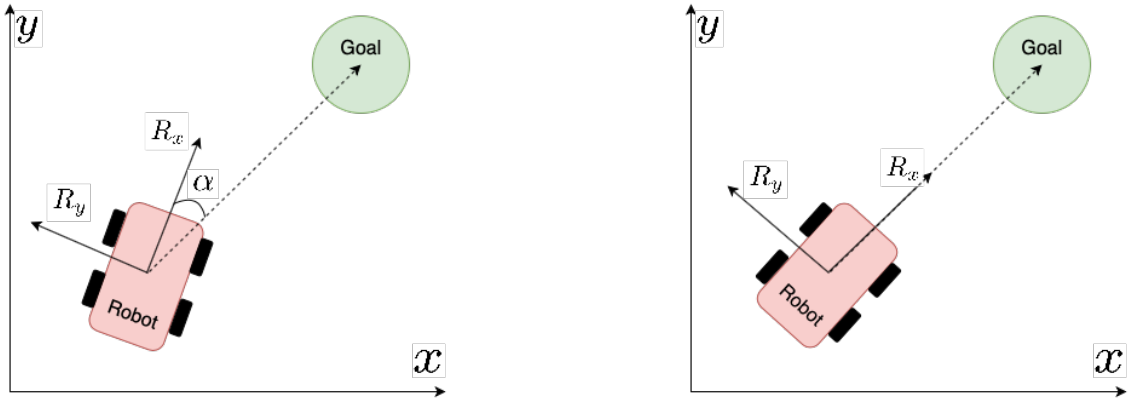
(b) Planned trajectory: start and goal positions

Figure 6.19: The robot (Perceptbot) (a) and the planned trajectory (b). The red circle represents the robot position (agent) and the green ✗ represents the desired destination (goal)

We have created a *ROS* master node (**Ros** component) which contains the Global Way-point LSTM Planner and a Motion Planner. The Motion Planner is responsible for physically moving the robot to the specified goal (or way-point), by querying simple velocity control commands using the *cmd_vel ROS* package, and for converting the real world coordinates into PathBench **Simulator** coordinates and vice-versa. The agent position and rotation is retrieved using the *robot_pose ROS package*. The YDLidar sensor is run using the *gmapping ROS* package with 0.15 meters grid cell size. Lastly, we have run the **Ros** master node and *gmapping* on a server which uses network packets (i.e. *ROS* publisher-subscriber APIs) to communicate with the robot. This was done because the performance of the *gmapping* package was severely impacted by the hardware limitations of the Raspberry Pi. The **Ros** master node could have been run on the Raspberry Pi itself, but we have configured it on the server for faster development and debugging. The *gmapping* SLAM scan output was integrated into PathBench by creating a custom map environment (**RosMap**), which has support for live updates.

The algorithm starts by converting all trace points generated by the Global Way-point LSTM Planner, including the ones generated by the local kernel, into way-points. The planning between two way-points is achieved by the Motion Planner with simple velocity commands. In the first phase, we rotate the robot so that the robot angle is equal to the angle of the direction to the next way-point. In the second phase, we move in a straight line until we reach the next way-point. If at any point we surpass a pre-defined angle threshold, we correct the robot pose by executing the first phase again, and continue the process. We only request a map update when we have reached the way-points suggested by the global kernel as the local kernel (A*) is offline (See Figure 6.20 and Algorithm 13).

Figure 6.21 showcases the performance of the real robot on the trajectory defined in Figure 6.19. We will also place the side by side view of the PathBench **Simulator** and the *ROS* simulator, Rviz. Table 6.28 contains the evaluation results reported by our platform.



(a) First Phase: Rotation

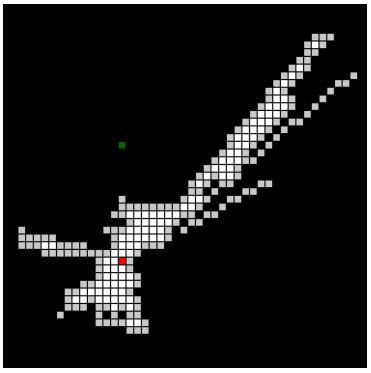
(b) Second Phase: Forward Movement

Figure 6.20: The illustrated two phases of the motion planning between two way-points. x and y are the world coordinate system and R_x and R_y are the robot frame coordinate system

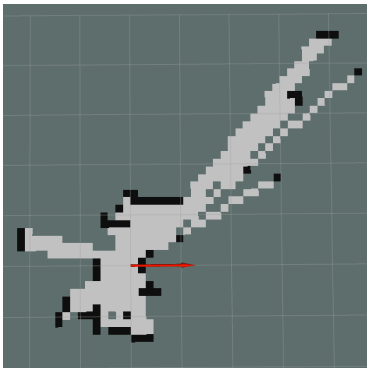
Algorithm 13 Robot-Planner

```
1:  procedure MOTION-PLANNER(wp, max_it, angle_threshold, goal_threshold)
2:      for i in [0, max_it) do
3:          agent_pos ← query agent position
4:          agent_angle ← query agent angle relative to the world coordinate system
5:          goal_dir ← wp - agent_pos
6:          goal_angle ← arctan2(goal_dir.y, goal_dir.x)
7:          α ← sign(goal_angle − agent_angle) (|goal_angle − agent_angle| % π)
8:
9:          if α ≥ angle_threshold then
10:             rotate with velocity proportional to α
11:             continue
12:
13:         if ‖goal_dir‖_2 ≥ dist_threshold then
14:             move forward with velocity proportional to ‖goal_dir‖_2
15:             continue
16:         else
17:             return
18:
19: procedure ROBOT-PLANNER(M : (A, Os, G))
20:     while goal is not reached do
21:         M ← query new SLAM scan
22:         way_points ← get Global Way-point LSTM Planner trace until next global way-point
23:
24:         for wp in way_points do
25:             Motion-Planner(wp, 1000, 0.1, 0.1)
26:
27:         if there are no way_points then
28:             goal was not found
29:             break
```

(a) Real-world: Start

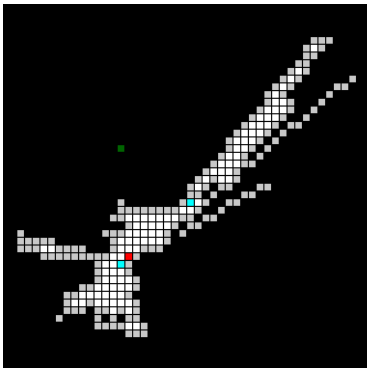


(b) PathBench: Start

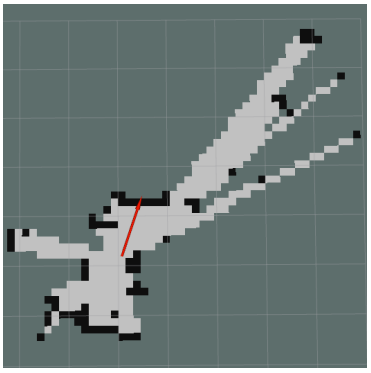


(c) Rviz: Start

(d) Real-world: Suggested WP 1

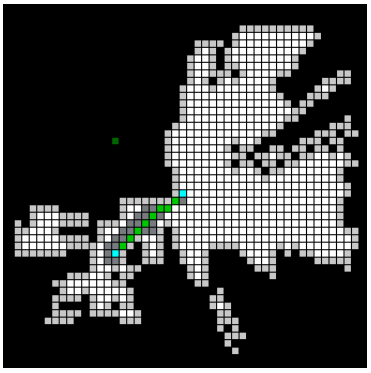


(e) PathBench: Suggested WP 1

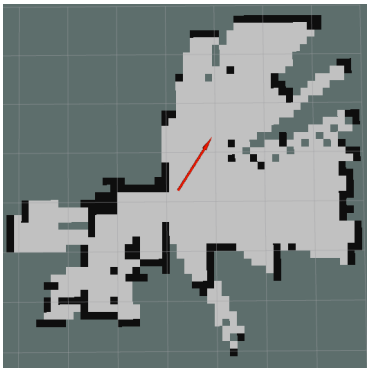


(f) Rviz: Suggested WP 1

(g) Real-world: Reached WP 1

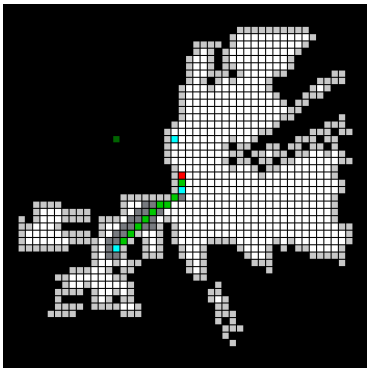


(h) PathBench: Reached WP 1

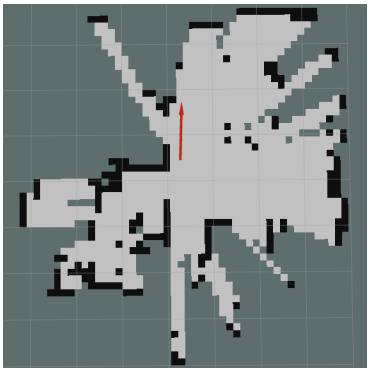


(i) Rviz: Reached WP 1

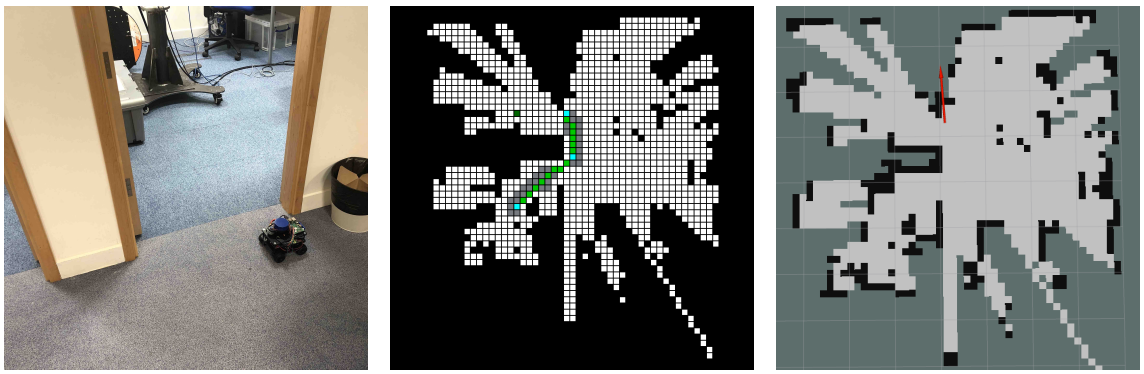
(j) Real-world: Suggested WP 2



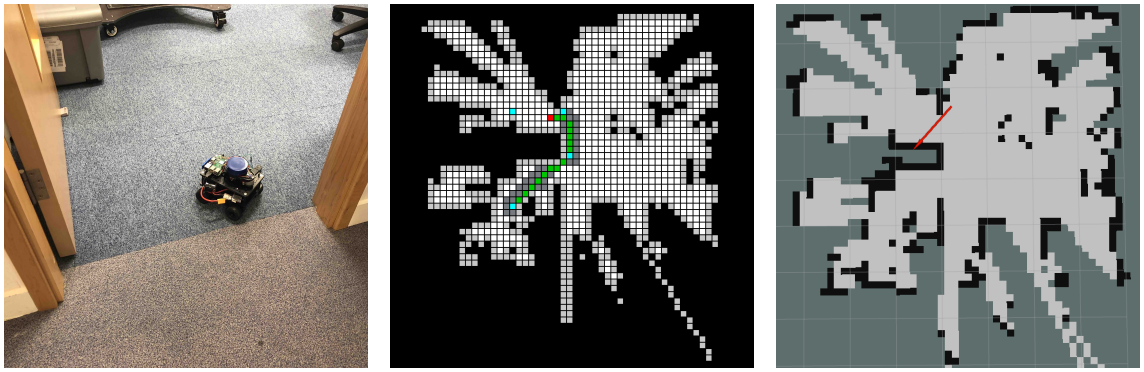
(k) PathBench: Suggested WP 2



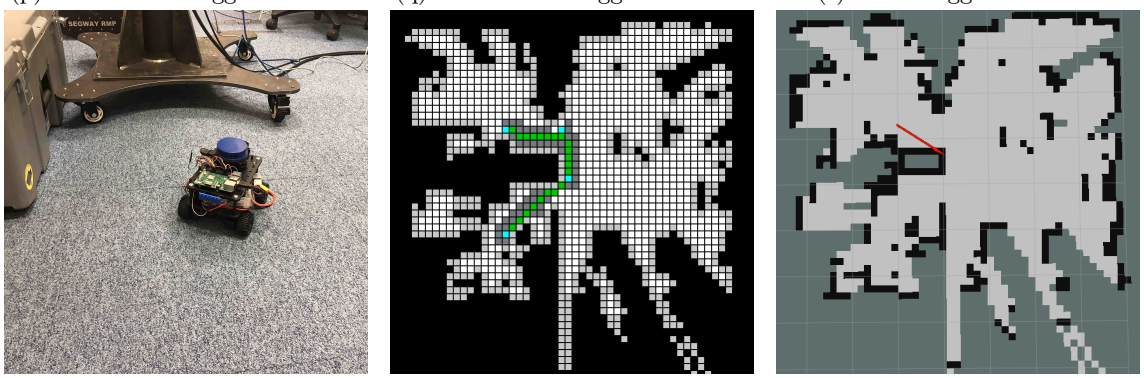
(l) Rviz: Suggested WP 2



(m) Real-world: Reached WP 2 (n) PathBench: Reached WP 2 (o) Rviz: Reached WP 2



(p) Real-world: Suggested WP 3 (q) PathBench: Suggested WP 3 (r) Rviz: Suggested WP 3



(s) Real-world: Final (t) PathBench: Final (u) Rviz: Final

Figure 6.21: Robot Path Planning using Global Way-point LSTM Planner run on the trajectory from Figure 6.19. The left image represents real-word view of the robot, the center image represents our live PathBench simulator visualisation and the right image represents the live Rviz simulator from ROS. The PathBench and Rviz images are cropped so that we only show the relevant information. The true dimension of the grid is 128×128

Name	Value
Goal Found	`True`
Grid Cell Size	0.15 meters
Map Size	128×128
Obstacles	92.03%
Original Distance	15.00/2.25 meters
Distance	28.56/4.824 meters
Time	365.252545 seconds/6.08 minutes
Distance Left	0.00/0.00 meters
Pick Ratio	[0.0%, 33.33%, 33.33%, 0.0%, 0.0%, 33.33%, 0.0%, 0.0%, 0.0%, 0.0%]
GK Improvement	100.00%
GK Distance	28.56/4.824 meters
GK Distance Left	0.00/0.00 meters
WP	`4`
WP In-Between Distance	9.04/1.356 meters
Total Search	0.47%
Total Fringe	0.31%
Session Search	0.13%
Session Fringe	0.09%

Table 6.28: Reported real-world statistics on the run from Figure 6.21. The PathBench and Rviz images from Figure 6.21 are cropped so that we only show the relevant information. The true dimension of the grid is 128×128

The results show that the robot successfully found a path in a partial knowledge environment (See Figure 6.21) by placing the last global way-point on the goal. The evaluation metrics reported in Table 6.28 show that we retain the same behaviour discussed in the **Analyser** simple and complex routines (maximal GK Improvement, low number of way-points with high way-point in-between distance; it should be noted that time statistic is influenced by the speed of the robot). Therefore, we have shown that the Global Way-point LSTM Planner supports partial knowledge environments in which classic offline solutions such as A* are unable to find a path to the goal without making use of an exploration method. Lastly, we have showed that the simulator is compatible with the *gmapping ROS* package and has support for live updates.

Chapter 7

Conclusion

In this chapter, we are going to summarise our findings and address future work.

7.1 Summary

In conclusion, we have successfully applied Machine Learning methods to the pathfinding problem. We have developed a hybrid solution which combines classic and ML methods to solve the path planning problem while maintaining support in full and partial knowledge environments, reducing the memory load compared to A*, generalising well in unknown environments and theoretically satisfying the real-world industrial application requirements.

The summary of the contributions is presented as follows (the source code can be found at https://gitlab.doc.ic.ac.uk/ait15/individual-project):

- Algorithmic solutions:
  - **CAE Online LSTM Planner** - We have managed to fix the long corridor issue from [15] to some extent by using a CAE network architecture
  - **LSTM Bagging Planner** - We have improved the overall performance of the Online LSTM and CAE Online LSTM Planners by combining their solutions
  - **Global Way-point LSTM Planner** - We have shown that the proposed solution theoretically achieves lower average case time and space complexity than A* (in 2D and 3D environments), is online, supports partial knowledge environments, reduces the memory load compared to A*, is robust to unknown environments and theoretically satisfies the real-world industrial application requirements (depending on the choice of local kernel)
- PathBench:
  - **Simulator** - We have built a simulator to visualise path planning algorithms. Moreover, the simulator abstracts the interactions between the robot and the environment for faster development and testing of new solutions
  - **Generator** - We have developed methods for generating synthetic ML training datasets
  - **Trainer** - We have built a training environment to boost the productivity of testing new ML architectures
  - **Analyser** - We have developed an analyser tool to create custom benchmarking statistics in order to assess the performance of the proposed solution
  - **ROS Real-time Extension** - We have added support for real-world simulation by implementing an updatable map environment which is compatible with the *gmapping ROS* package

- Theoretic and real-world evaluations:
  - **Complexity and Theoretical Analysis** - We have stated the theoretical worst (and average) case time and space complexities for all discussed algorithms. We have shown that the proposed solution theoretically achieves lower average case time and space complexity than A*
  - **Empirical Methods** - We have practically evaluated the proposed solutions by running empirical routines using custom benchmarking statistics. We have proved that the Global Way-point LSTM Planner significantly reduces the memory load compared to A* and that it generalises well in unknown environments
  - **Real-world Evaluation** - We have tested the performance of the proposed solution on real-world occupancy grid maps generated by real-world robots. We have implemented the proposed solution on a real-world robot and tested it at Imperial College London. We have proved that the Global Way-point LSTM Planner supports partial knowledge environments

7.2 Future Work

We believe that more work can be done in the area of pathfinding ML methods to improve the overall performance of the Global Way-point LSTM Planner and potentially find more solutions:

- **Parallelising Issue Fix** - This should be a top priority as it is a trivial performance boost that cannot be used due to the possible issues in the *pytorch* framework
- **Path Refining Techniques** - The produced path is sometimes unnecessarily long and can be shortened by applying some path refining techniques (e.g. [1] uses this kind of techniques to generate a high-quality path) such as a moving average. Not only that we reduce the distance by a significant amount, but we can also reduce the collision chance by pushing the path further away from the obstacles (if we use the moving average technique)
- **Advanced Synthetic Data Generation Techniques** - More generation methods should be investigated such as: Maze Generation, Cellular Automata Cave Generation and Generative Adversarial Networks (GANs) [31, 14, 32, 33]
- **Real Datasets** - The problem with synthetic generated datasets is that the ML models might learn the generation procedure and will be biased when dealing with unknown environments (i.e. the generalisation property is deteriorated). Therefore, having a real dataset should improve the overall performance of the ML models and boost the generalisation property
- **Advanced Hyper-parameter Search** - As seen in the real-world experiments, the hyper-parameter choice (kernels, kernel priority and max global kernel iterations) is crucial. Moreover, we share the same issue with the training performance of ML methods. Therefore, more work should be done in this area by adopting different search strategies that improve the overall performance of the ML models such as: Grid Search, Random Search and optimisation techniques (Bayesian Optimisation, Gradient-based Optimisation, Line Search, Golden Section, Newton Methods, Lipschitz Optimisation) [35]
- **New Machine Learning Models** - More ML methods should be investigated such as an LSTM network that uses Attention, Gated Recurrent Units (GRUs), PCA, GANs for generating datasets, Reinforcement Learning approaches (Deep Q-Networks (DQNs), Value Iteration Networks (VINs)) and many more [31, 14, 32, 33]
- **Practical Evaluations for Theoretical Properties** - We have shown that the proposed solution has support for partial knowledge environments, reduces the memory load compared to A* and generalises well in unknown environments, but we have only theoretically proven the real-world industrial application requirements. Therefore, more work should be taken to empirically assess the theoretically proven properties

Bibliography

[1] M. Inoue, T. Yamashita, and T. Nishida, "Robot Path Planning by LSTM Network Under Changing Environment," in *Advances in Computer Communication and Computational Sciences*, pp. 317–329, Springer, 2019.

[2] D. González, J. Pérez, V. Milanés, and F. Nashashibi, "A Review of Motion Planning Techniques for Automated Vehicles.," *IEEE Trans. Intelligent Transportation Systems*, vol. 17, no. 4, pp. 1135–1145, 2016.

[3] S. M. LaValle and J. J. Kuffner Jr, "Randomized kinodynamic planning," *The international journal of robotics research*, vol. 20, no. 5, pp. 378–400, 2001.

[4] H. M. Choset, S. Hutchinson, K. M. Lynch, G. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun, *Principles of robot motion: theory, algorithms, and implementation*. MIT press, 2005.

[5] F. Duchoň, A. Babinec, M. Kajan, P. Beňo, M. Florek, T. Fico, and L. Jurišica, "Path planning with modified a star algorithm for a mobile robot," *Procedia Engineering*, vol. 96, pp. 59–69, 2014.

[6] Z. Zhang and Z. Zhao, "A multiple mobile robots path planning algorithm based on a-star and dijkstra algorithm," *International Journal of Smart Home*, vol. 8, no. 3, pp. 75–86, 2014.

[7] W. Y. Loong, L. Z. Long, and L. C. Hun, "A star path following mobile robot," in *2011 4th International Conference on Mechatronics (ICOM)*, pp. 1–7, May 2011.

[8] S. M. LaValle, "Rapidly-exploring random trees: A new tool for path planning," 1998.

[9] S. Rodriguez, X. Tang, J.-M. Lien, and N. M. Amato, "An obstacle-based rapidly-exploring random tree," in *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, pp. 895–900, IEEE, 2006.

[10] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The international journal of robotics research*, vol. 30, no. 7, pp. 846–894, 2011.

[11] C. Szepesvári, "Algorithms for reinforcement learning," *Synthesis lectures on artificial intelligence and machine learning*, vol. 4, no. 1, pp. 1–103, 2010.

[12] J. K. Satia and R. E. Lave Jr, "Markovian decision processes with uncertain transition probabilities," *Operations Research*, vol. 21, no. 3, pp. 728–740, 1973.

[13] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[14] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. Available: http://www.deeplearningbook.org; accessed June 17, 2019.

[15] F. Nicola, Y. Fujimoto, and R. Oboe, "A LSTM Neural Network applied to Mobile Robots Path Planning," in *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*, pp. 349–354, IEEE, 2018.

[16] L. Lee, E. Parisotto, D. S. Chaplot, and R. Salakhutdinov, "LSTM Iteration Networks: An Exploration of Differentiable Path Finding," 2018.

[17] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "Ros: an open-source robot operating system," in *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*, (Kobe, Japan), May 2009.

[18] I. A. Şucan, M. Moll, and L. E. Kavraki, "The Open Motion Planning Library," *IEEE Robotics & Automation Magazine*, vol. 19, pp. 72–82, December 2012. http://ompl.kavrakilab.org.

[19] I. A. Şucan and S. Chitta, "MoveIt," Available: https://moveit.ros.org; accessed June 17, 2019.

[20] M. Moll, I. A. Sucan, and L. E. Kavraki, "Benchmarking motion planning algorithms: An extensible infrastructure for analysis and visualization," *IEEE Robotics & Automation Magazine*, vol. 22, no. 3, pp. 96–102, 2015.

[21] M. G. Dissanayake, P. Newman, S. Clark, H. F. Durrant-Whyte, and M. Csorba, "A solution to the simultaneous localization and map building (slam) problem," *IEEE Transactions on robotics and automation*, vol. 17, no. 3, pp. 229–241, 2001.

[22] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd ed., 2009.

[23] C. Luo, M. Krishnan, M. Paulik, and G. E. Jan, "An effective trace-guided wavefront navigation and map-building approach for autonomous mobile robots," in *Intelligent Robots and Computer Vision XXXI: Algorithms and Techniques*, vol. 9025, p. 90250U, International Society for Optics and Photonics, 2014.

[24] S. Rajko and S. M. LaValle, "A pursuit-evasion bug algorithm," in *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No. 01CH37164)*, vol. 2, pp. 1954–1960, IEEE, 2001.

[25] I. Kamon, E. Rimon, and E. Rivlin, "Tangentbug: A range-sensor-based navigation algorithm," *The International Journal of Robotics Research*, vol. 17, no. 9, pp. 934–953, 1998.

[26] V. Lumelsky and A. Stepanov, "Dynamic path planning for a mobile automaton with limited information on the environment," *IEEE transactions on Automatic control*, vol. 31, no. 11, pp. 1058–1063, 1986.

[27] J. D. Gammell, S. S. Srinivasa, and T. D. Barfoot, "Informed RRT*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic," *arXiv preprint arXiv:1404.2334*, 2014.

[28] S. S. Ge and Y. J. Cui, "Dynamic motion planning for mobile robots using potential field method," *Autonomous robots*, vol. 13, no. 3, pp. 207–222, 2002.

[29] J. Barraquand, B. Langlois, and J.-C. Latombe, "Numerical potential field techniques for robot path planning," *IEEE transactions on systems, man, and cybernetics*, vol. 22, no. 2, pp. 224–241, 1992.

[30] A. Woods and H. La, "Dynamic Target Tracking and Obstacle Avoidance using a Drone," 12 2015.

[31] T. M. Mitchell, *Machine Learning*. McGraw-Hill Science/Engineering/Math, 1997.

[32] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Malaysia; Pearson Education Limited,, 2016.

[33] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

[34] C. J. B. Yann LeCun, Corinna Cortes, "The MNIST Database," 1998. Available: http://yann.lecun.com/exdb/mnist/; accessed June 17, 2019.

[35] E. K. Chong and S. H. Zak, *An introduction to optimization*, vol. 76. John Wiley & Sons, 2013.

[36] C. Olah, "Understanding LSTM Networks," 2015. Available: https://colah.github.io/posts/2015-08-Understanding-LSTMs/; accessed June 17, 2019.

[37] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[38] P. Shinners, "Pygame," 2011. Available: http://pygame.org/; accessed June 17, 2019.

[39] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[40] U. India, "Tensorflow or PyTorch : The force is strong with which one?," 2018. Available: https://medium.com/@UdacityINDIA/tensorflow-or-pytorch-the-force-is-strong-with-which-one-68226bb7dab4; accessed June 17, 2019.

[41] A. Holkner, "Pyglet documentation," 2017. Available: https://buildmedia.readthedocs.org/media/pdf/pyglet/latest/pyglet.pdf; accessed June 17, 2019.

[42] C. Bartneck, M. Soucy, K. Fleuret, and E. B. Sandoval, "The robot engine—making the unity 3d game engine work for hri," in *2015 24th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN)*, pp. 431–437, IEEE, 2015.

[43] D. Shreiner, *OpenGL Reference Manual: The Official Reference Document to OpenGL, Version 1.2*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 3rd ed., 1999.

[44] M. Foord and C. Muirhead, *IronPython in action*. Manning Publications Co., 2009.

[45] V. Blomqvist, "pymunk documentation," 2019. Available: https://buildmedia.readthedocs.org/media/pdf/pymunk/latest/pymunk.pdf; accessed June 17, 2019.

[46] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[47] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[48] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[49] D. Holden, J. Saito, T. Komura, and T. Joyce, "Learning motion manifolds with convolutional autoencoders," in *SIGGRAPH Asia 2015 Technical Briefs*, p. 18, ACM, 2015.

[50] I. Jolliffe, *Principal component analysis*. Springer, 2011.

[51] S. Wold, K. Esbensen, and P. Geladi, "Principal component analysis," *Chemometrics and intelligent laboratory systems*, vol. 2, no. 1-3, pp. 37–52, 1987.

[52] G. H. Alex Krizhevsky, Vinod Nair, "The CIFAR-10 Dataset," 2009. Available: https://www.cs.toronto.edu/~kriz/cifar.html; accessed June 17, 2019.

[53] T. G. Dietterich, "Ensemble methods in machine learning," in *International workshop on multiple classifier systems*, pp. 1–15, Springer, 2000.

[54] "Colab System Specs," 2019. Available: https://colab.research.google.com/drive/151805XTDg--dgHb3-AXJCpnWaqRhop_2#scrollTo=gsqXZwauphVV; accessed June 17, 2019.

[55] S. Song, F. Yu, A. Zeng, A. X. Chang, M. Savva, and T. Funkhouser, "Semantic scene completion from a single depth image," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1746–1754, 2017.

[56] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux, "API design for machine learning software: experiences from the scikit-learn project," in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pp. 108–122, 2013.

[57] F. Krüger, *Activity, Context, and Plan Recognition with Computational Causal Behaviour Models*. PhD thesis, 12 2016.

[58] A. Howard, L. Parker, and G. S. Sukhatme, "The sdr experience: Experiments with a large-scale heterogeneous mobile robot team," vol. 21, 06 2004.

[59] S. Gholami Shahbandi and M. Magnusson, "2d map alignment with region decomposition," *Autonomous Robots*, vol. 43, pp. 1117–1136, Jun 2019.

[60] R. Tang, "Custom Player plugins," Available: http://robotang.co.nz/projects/robotics/custom-player-plugins/; accessed June 17, 2019.

[61] "YDLidar X4 Datasheet," Available: https://www.elecrow.com/download/X4_Lidar_Datasheet.pdf; accessed June 17, 2019.

[62] G. T. K. T. J. W. Martin Fisch, Joern Messner, *PerceptionBot: Design and Development of an Autonomous Robotic Platform with Features such as Perception, Path Planning, Control, and Object Detection*. PhD thesis, 2019.

[63] G. Halfacree and E. Upton, *Raspberry Pi User Guide*. Wiley Publishing, 1st ed., 2012.

[64] "Raspbian," Available: https://www.raspbian.org; accessed June 17, 2019.

[65] J. Hannemann and G. Kiczales, "Design pattern implementation in java and aspectj," *ACM Sigplan Notices*, vol. 37, no. 11, pp. 161–173, 2002.

[66] G. E. Krasner, S. T. Pope, *et al.*, "A description of the model-view-controller user interface paradigm in the smalltalk-80 system," *Journal of object oriented programming*, vol. 1, no. 3, pp. 26–49, 1988.

Appendix A

PathBench

A.1 Infrastructure

The **MainRunner** component is the main entry point of the platform and it coordinates all other sections. The **MainRunner** takes a master **Configuration** component as input which represents the main inflexion point of the platform. It describes which section (**Simulator**, **Generator**, **Trainer**, **Analyser**) should be used and how (The master **Configuration** fields can be seen in Tables A.1, A.2 and A.3).

The **Services** component is a bag of **Service** components which is injected into all platform classes in order to maintain global access to the core libraries[1]. A **Service** component is created for most external libraries to encapsulate their APIs and provide useful helper functions[2]. Moreover, by making use of the Adapter Pattern we can easily switch third party libraries, if needed, and the code becomes more test friendly. Finnaly, the **Services** container can be mocked together with all its **Service** components, thus avoiding rendering, file writing and useless printing.

The **Simulator** was build by following the Model-View-Controller (MVC) pattern [66]. The **Model** represents the logic part, the **View** renders the **Model** and the **Controller** handles the input from the keyboard and mouse, and calls the appropriate functions from the associated **Model**. The **EventManager** component is a communication service which allows the **Model** to update the **View** as there is no direct connection between them (from **Model** to **View**, the other way is).

The **Debug** component is a printing service which augments printing messages with different decorators such as time-stamp and routes the messages to a specified IO stream or standard out. It also provides a range of debugging/printing modes: None (no information), Basic (only basic information), Low (somewhat verbose), Medium (quite verbose), High (all information).

The **RenderingEngine** component is a wrapper around the *pygame* library and all rendering is routed through it.

The **Torch** service is not an actual wrapper around *pytorch*, but instead it defines some constants such as the initial random seed and the training device (CPU/CUDA).

The **Resources** service is the persistent storage system. It is a container of **Directory** components which represent an interface over the actual filesystem directories. It provides safe interaction with the filesystem and a range of utility directories: Cache (temporary storage used for speeding second runs), Screenshots, Maps (stores all user defined and generated maps), Images (stores images which

[1] Design known as Dependency Injection or Strategy Pattern [65]
[2] Design known as the Adapter Pattern [65]

can be converted to internal maps), Algorithms (stores trained machine learning models), Training Data (stores training data for machine learning models). The main serialisation tool is *dill* which is a wrapper around *pickle* with lambda serialisation capabilities, but custom serialisation is allowed such as tensor serialisation provided by *pytorch* or image saving by *pygame*.

The **AlgorithmRunner** manages the algorithm session which contains the **Algorithm**, **BasicTesting** and **Map**. The **AlgorithmRunner** launches a separate daemon thread that is controlled by a condition variable. When writing an **Algorithm**, special key frames can be defined (e.g. when the trace is produced) to create animations. Key frames release the synchronisation variable for a brief period and then acquire it again, thus querying new rendering jobs.

The **Utilities** section provides a series of helper methods and classes: **Maps** (holds in-memory user defined **Map** components), **Point**, **Size**, **Progress** (progress bar), **Timer**, **MapProcessing** (feature extractor used mainly in ML sections).

A.2 Master Configuration and User Commands

Configuration Field	Type	Description
load_simulator	`bool`	If the simulator should be loaded
clear_cache	`bool`	If the cache should be deleted after the simulator is finished
simulator_graphics	`bool`	If graphics should be used or not; evaluation is always done without graphics
simulator_grid_display	`bool`	The map can be visualised as a plain image or a grid; the window size is defined based on the choice
simulator_initial_map	`Map`	The map used in **AlgorithmRunner** service
simulator_algorithm_type	`Type[Algorithm]`	The algorithm type used in **AlgorithmRunner** service
simulator_algorithm_parameters	`Tuple[List[Any], Dict[str, Any]]`	The algorithm parametrs in the form of *args and **kwargs which are used in **AlgorithmRunner** service
simulator_testing_type	`Type[BasicTesting]`	The testing type used in **AlgorithmRunner** service
simulator_key_frame_speed	`int`	The refresh rate interval during each key frame; a value of 0 disables the key frames
simulator_key_frame_skip	`int`	How many key frames are skipped at a time; used to speed up the animation when frames per second are low
simulator_write_debug_level	`DebugLevel`	The debugging level (None, Basic, Low, Medium, High)

Table A.1: Simulator master **Configuration** fields

Configuration Field	Type	Description
generator	`bool`	If the generator should be loaded
generator_gen_type	`str`	Generation type; can choose between "uniform_random_fill", "block_map" and "house" (See Figure 4.5)
generator_nr_of_examples	`int`	How many maps should be generated; 0 does not trigger generation
generator_labelling_atlases	`List[str]`	Which **Map Atlas**es should be converted to training data
generator_labelling_features	`List[str]`	Which sequential features should be extracted for training conversion
generator_labelling_labels	`List[str]`	Which sequential labels should be extracted for training conversion
generator_single_labelling_features	`List[str]`	Which single features should be extracted for training conversion
generator_single_labelling_labels	`List[str]`	Which single labels should be extracted for training conversion
generator_aug_labelling_features	`List[str]`	Which sequential features should be augmented for training data defined by generator_labelling_atlases
generator_aug_labelling_labels	`List[str]`	Which sequential labels should be augmented for training data defined by generator_labelling_atlases
generator_aug_single_labelling_features	`List[str]`	Which single features should be augmented for training data defined by generator_labelling_atlases
generator_aug_single_labelling_labels	`List[str]`	Which single labels should be augmented for training data defined by generator_labelling_atlases
generator_modify	`Callable[[Map], Map]`	Modifies the given map using the custom function

Table A.2: Generator master **Configuration** fields

Configuration Field	Type	Description
trainer	`bool`	If the trainer should be loaded
trainer_model	`Type[MLModel]`	The model which will be trained
trainer_custom_config	`Dict[str, Any]`	If a custom configuration should augment the **MLModel** configuration
trainer_pre_process_data_only	`bool`	If the trainer should only pre-process data and save it; it does not overwrite cache
trainer_bypass_and_replace_pre_processed_cache	`bool`	If pre-processed data cache should be bypassed and re-computed

Table A.3: Trainer master **Configuration** fields

Key	Action
up arrow	moves agent up
left arrow	moves agent left
down arrow	moves agent down
right arrow	moves agent right
c	compute trace
s	stop trace animation (requires key frames)
r	resume trace animation (requires key frames)
m	toggle map between **SparseMap** and **DenseMap**
p	take map screenshot
mouse hover	reports hovered cell coordinates (requires at least Medium debugging level)
mouse left click	moves **Agent** to mouse location
mouse right click	moves **Goal** to mouse location

Table A.4: Simulator user commands

Appendix B

Methods

B.1 Packing and Unpacking

Packing is used to speed up the forward pass through the LSTM layer and mask unused data (because data is given as a variable length sequence). By making use of packing, we can remove the need to pad the inputs with a special token in order to convert the variable size sequence into a constant size sequence. Moreover, if the pad token is not chosen correctly the network might learn from it and become biased towards one action (e.g. if the labelling pad token is 0, then the 0 action might get preferred over the other actions). When packing a batch sequence, all sequences have to be sorted in reverse order of the sequence length (largest sequence first). The packing is done by using the function `pack_sequence` from *pytorch* which returns a `PackedSequence` object. The `PackedSequence` transforms the data into a continuous 1D tensor by concatenating all batch tensors. The structure of the original data is still preserved into the `batch_sizes` attribute. The whole packing procedure has complexity $\mathcal{O}(n \log(n))$ (sorting is $\mathcal{O}(n \log(n))$ and `pack_sequence` is $\mathcal{O}(n)$, where n is the batch size). The following code snippet showcases an example of the packing procedure:

```
import torch
from torch.nn.utils.rnn import pack_sequence, PackedSequence

a = torch.Tensor([1, 2])
b = torch.Tensor([3, 4, 5])
c = torch.Tensor([6])
seq = [a, b, c]
seq.sort(key=lambda el: el.shape[0], reverse=True)
packed_sequence: PackedSequence = pack_sequence(seq)

# Output: PackedSequence(data=tensor([3., 1., 6., 4., 2., 5.]),
#                        batch_sizes=tensor([3, 2, 1]))
```

Unpacking represents the inverse operation of packing, which reconstructs the original data from a `PackedSequence` object. Normally the `pad_packed_sequence` function is used to reconstruct the data, but because we pack both input and output we can speed up unpacking by directly accessing the `.data` attribute from the `PackedSequence` object (if we had to feed the previous action as it was done in [15] we couldn't have used this optimisation as we needed the full reconstructed data). Unoptimised unpacking is $\mathcal{O}(n)$ and optimised unpacking is $\mathcal{O}(1)$ where n is the batch size. The following code snippet showcases an example of the (un)optimised packing procedure:

```
# clone previous code snippet
from torch.nn.utils.rnn import pad_packed_sequence
```

```
# unoptimised
original_data, original_data_lengths = \\
        pad_packed_sequence(packed_sequence, batch_first=True)

# Output: (tensor([[3., 4., 5.],
#                  [1., 2., 0.],
#                  [6., 0., 0.]]), tensor([3, 2, 1]))

# optimised
data = packed_sequence.data
# Output: tensor([3., 1., 6., 4., 2., 5.])
```

Appendix C

Evaluation

C.1 Algorithms

This section contains extra evaluation results. For convenience we have provided the evaluated algorithms table as well (See Tables C.2, C.1).

Kernel	Training Data	Algorithm
CAE Online LSTM	block_map_10000	7
CAE Online LSTM	uniform_random_fill_10000	6
CAE Online LSTM	house_10000	8
CAE Online LSTM	uniform_random_fill_10000_block_map_10000_house_10000	9
CAE Online LSTM	uniform_random_fill_10000_block_map_10000	10
Online LSTM	uniform_random_fill_10000	1
Online LSTM	block_map_10000	2
Online LSTM	house_10000	3
Online LSTM	uniform_random_fill_10000_block_map_10000	4
Online LSTM	uniform_random_fill_10000_block_map_10000_house_10000	5

Table C.1: LSTM Bagging Planner (Algorithms 11 and 14) kernel configuration in priority order

Nr.	Planner	Training Data
0	A*	n/a
1	Online LSTM ([15])	`uniform_random_fill_10000`
2	Online LSTM	`block_map_10000`
3	Online LSTM	`house_10000`
4	Online LSTM	`uniform_random_fill_10000_block_map_10000`
5	Online LSTM	`uniform_random_fill_10000_block_map_10000_house_10000`
6	CAE Online LSTM	`uniform_random_fill_10000`
7	CAE Online LSTM ([1])	`block_map_10000`
8	CAE Online LSTM	`house_10000`
9	CAE Online LSTM	`uniform_random_fill_10000_block_map_10000`
10	CAE Online LSTM	`uniform_random_fill_10000_block_map_10000_house_10000`
11	LSTM Bagging	See Table C.1
12	Global Way-point LSTM GK: CAE Online LSTM	`block_map_10000`
13	Global Way-point LSTM GK: Online LSTM	`uniform_random_fill_10000_block_map_10000_house_10000`
14	Global Way-point LSTM GK: LSTM Bagging (proposed solution)	See Table C.1

Table C.2: Evaluated algorithms and their respective training dataset. All algorithms are colour-coded: A* is light-grey, Online LSTM Planner is red (solution with same training dataset as [15] is darker red), CAE Online LSTM Planner is blue (solution with same training dataset as [1] is darker blue), LSTM Bagging Planner is orange, Global Way-point LSTM Planner is half cyan and half global kernel colour (e.g. Algorithm 14 has both cyan and orange colours as it uses the LSTM Bagging Planner as the GK)

C.2 Online LSTM Planner Full Training Analysis

Model	Training Loss	Validation Loss	Evaluation Loss	Accuracy	Precision	Recall	F1	Confusion Matrix
1	0.033805	0.225089	0.141824	0.96	0.96	0.96	0.96	See Table C.4
2	0.032614	0.105727	0.077589	0.98	0.97	0.97	0.97	See Table C.5
3	0.110707	0.430041	0.357634	0.91	0.91	0.91	0.91	See Table C.6
4	0.029944	0.090220	0.071301	0.97	0.97	0.97	0.97	See Table C.7
5	0.025989	0.114388	0.115875	0.92	0.92	0.92	0.92	See Table C.8

Table C.3: Online LSTM Planner final training statistics

		Predicted							
	Action	0	1	2	3	4	5	6	7
Actual	0	175	0	7	0	0	1	3	0
	1	4	207	2	0	0	0	0	0
	2	0	0	97	0	2	0	0	0
	3	0	0	0	202	1	0	0	0
	4	0	0	0	0	135	1	0	0
	5	0	0	1	0	0	227	5	1
	6	0	0	0	1	1	0	160	0
	7	7	0	1	0	0	3	5	164

Table C.4: Confusion matrix for Algorithm 1

		Predicted							
Actual	**Action**	0	1	2	3	4	5	6	7
	0	140	2	0	0	0	0	0	0
	1	0	87	3	0	1	0	0	1
	2	0	0	340	1	0	0	0	0
	3	0	0	1	163	4	2	0	0
	4	0	0	0	1	265	1	0	0
	5	0	0	0	1	3	115	3	0
	6	1	0	0	0	0	0	284	0
	7	4	0	0	0	0	0	1	125

Table C.5: Confusion matrix for Algorithm 2

		Predicted							
Actual	**Action**	0	1	2	3	4	5	6	7
	0	145	1	0	0	1	1	0	2
	1	8	130	4	1	1	15	1	1
	2	0	3	168	3	0	1	0	0
	3	0	0	4	237	6	7	0	1
	4	0	0	0	7	189	4	0	1
	5	1	1	3	2	6	222	2	11
	6	0	0	1	1	0	4	164	5
	7	8	4	0	1	0	2	7	220

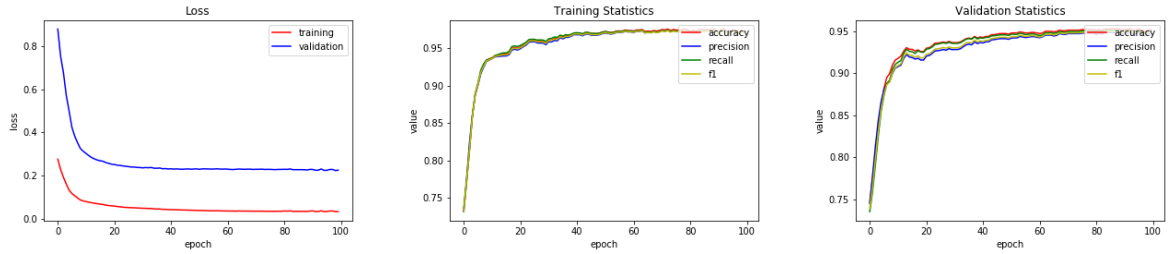
Table C.6: Confusion matrix for Algorithm 3

		Predicted							
Actual	**Action**	0	1	2	3	4	5	6	7
	0	339	1	5	1	0	0	1	2
	1	9	277	2	1	0	0	0	1
	2	0	5	313	0	1	0	0	0
	3	2	0	4	173	1	0	0	0
	4	0	0	2	0	114	2	0	0
	5	0	0	0	0	1	184	0	0
	6	0	0	0	0	1	0	162	1
	7	0	6	2	0	0	2	1	206

Table C.7: Confusion matrix for Algorithm 4

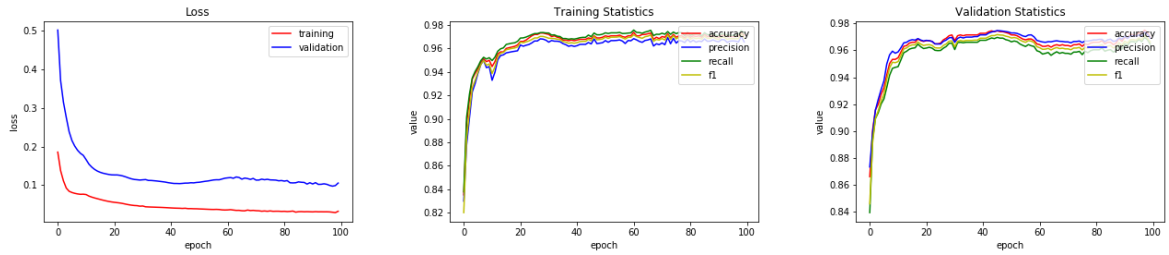
		Predicted							
Actual	**Action**	0	1	2	3	4	5	6	7
	0	177	2	0	0	0	0	0	6
	1	1	164	8	4	0	6	2	5
	2	0	1	142	2	1	1	11	0
	3	0	0	2	195	1	11	1	1
	4	0	0	3	6	224	1	1	1
	5	0	2	1	9	6	255	6	4
	6	0	0	0	0	4	2	219	2
	7	1	8	0	0	1	5	4	247

Table C.8: Confusion matrix for Algorithm 5



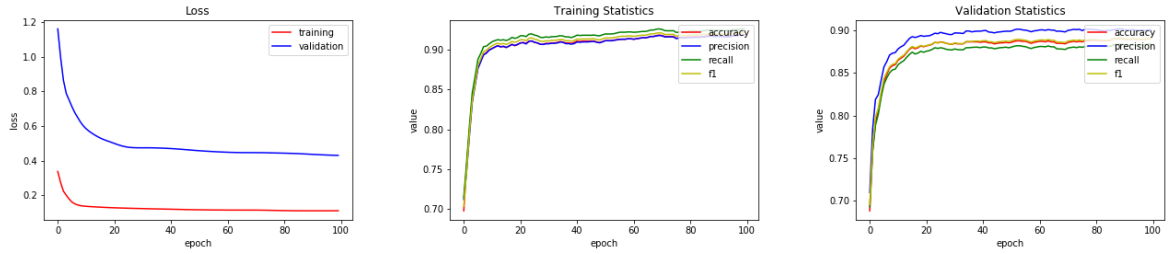
(a) Training/Validation Loss (b) Training Statistics (c) Validation Statistics

Figure C.1: Training statistics for Algorithm 1



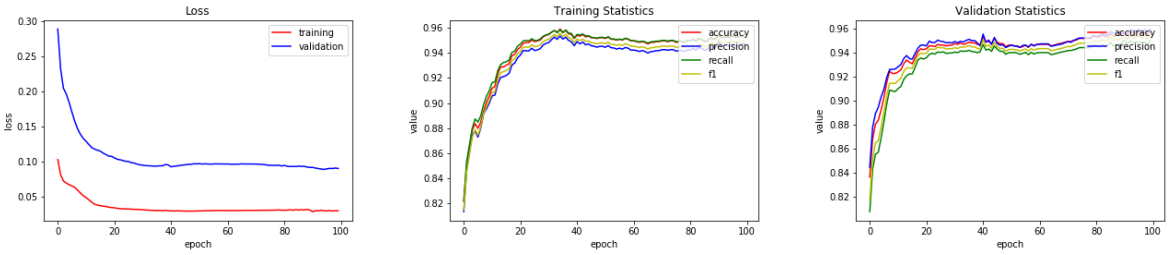
(a) Training/Validation Loss (b) Training Statistics (c) Validation Statistics

Figure C.2: Training statistics for Algorithm 2



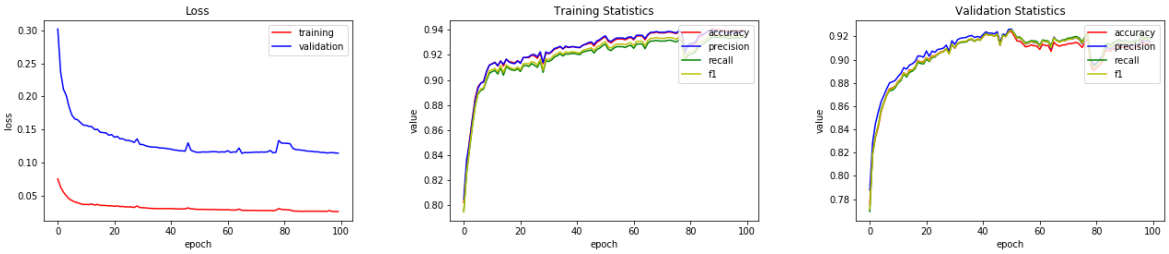
(a) Training/Validation Loss (b) Training Statistics (c) Validation Statistics

Figure C.3: Training statistics for Algorithm 3



(a) Training/Validation Loss (b) Training Statistics (c) Validation Statistics

Figure C.4: Training statistics for Algorithm 4



(a) Training/Validation Loss (b) Training Statistics (c) Validation Statistics

Figure C.5: Training statistics for Algorithm 5

C.3 CAE Online LSTM Planner Full Training Analysis

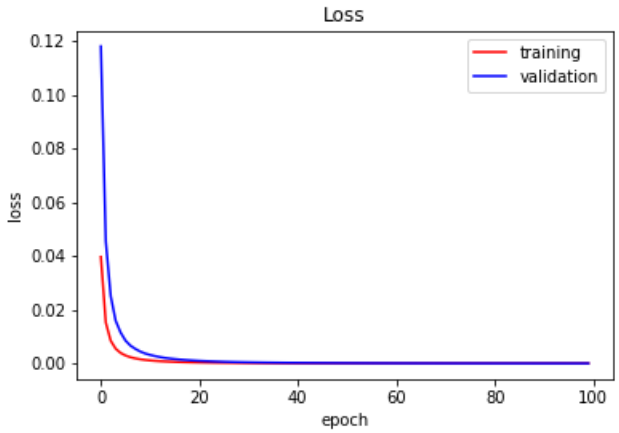


Figure C.6: Training/Validation Loss for CAE model (Algorithm 6) training loss (Train Loss: 0.000002, Validation Loss: 0.000005, Evaluation Loss: 0.000005)

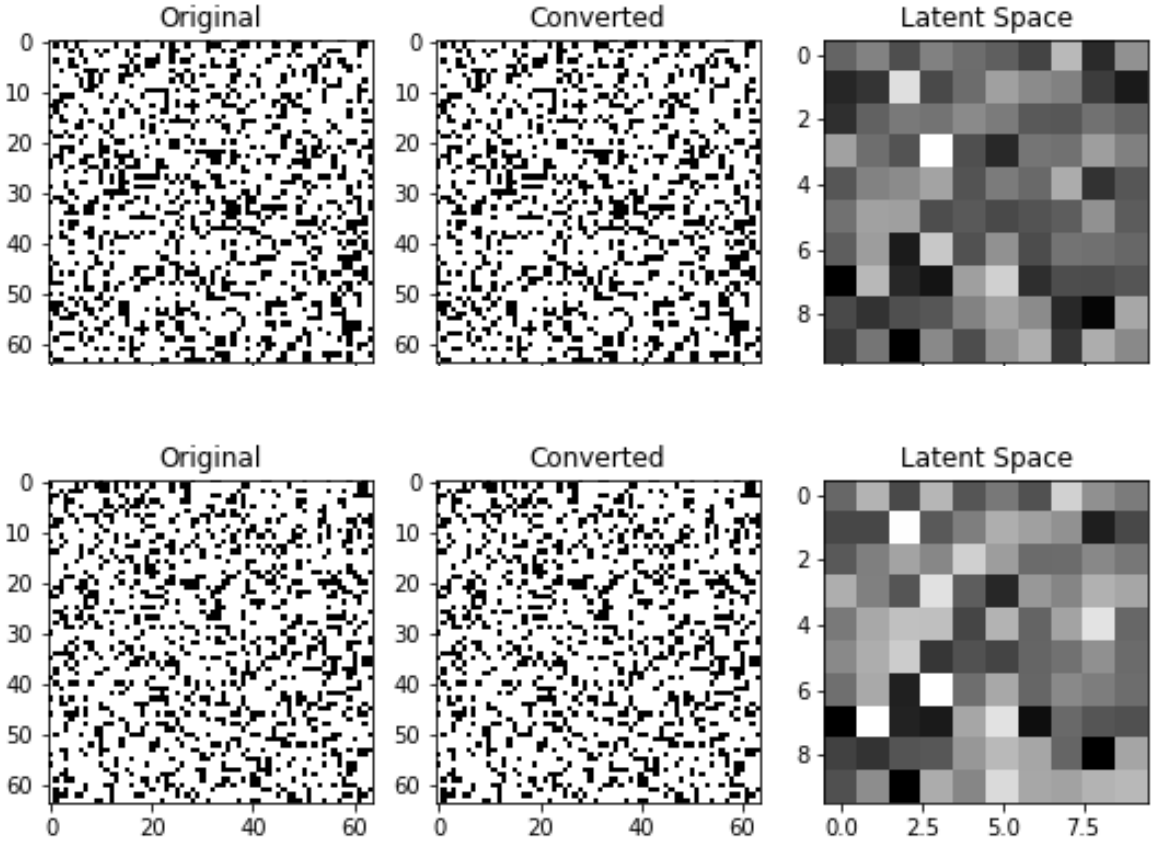


Figure C.7: CAE model (Algorithm 6) network analysis

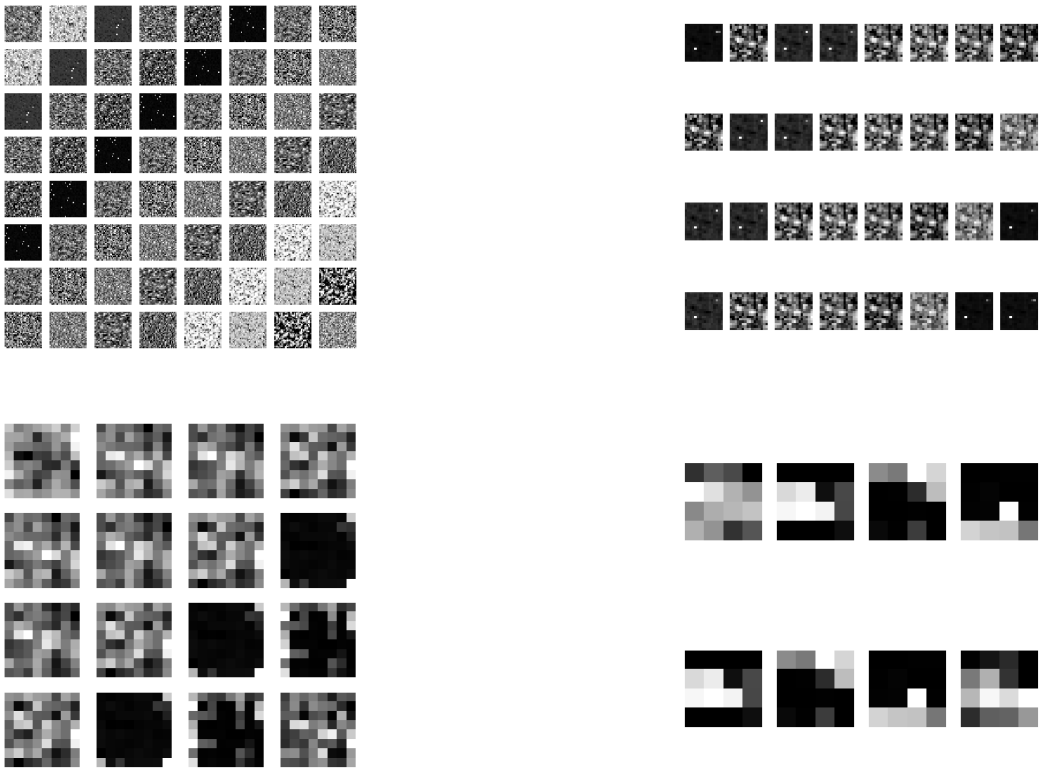


Figure C.8: CAE model (Algorithm 6) first map from Figure C.7 feature maps

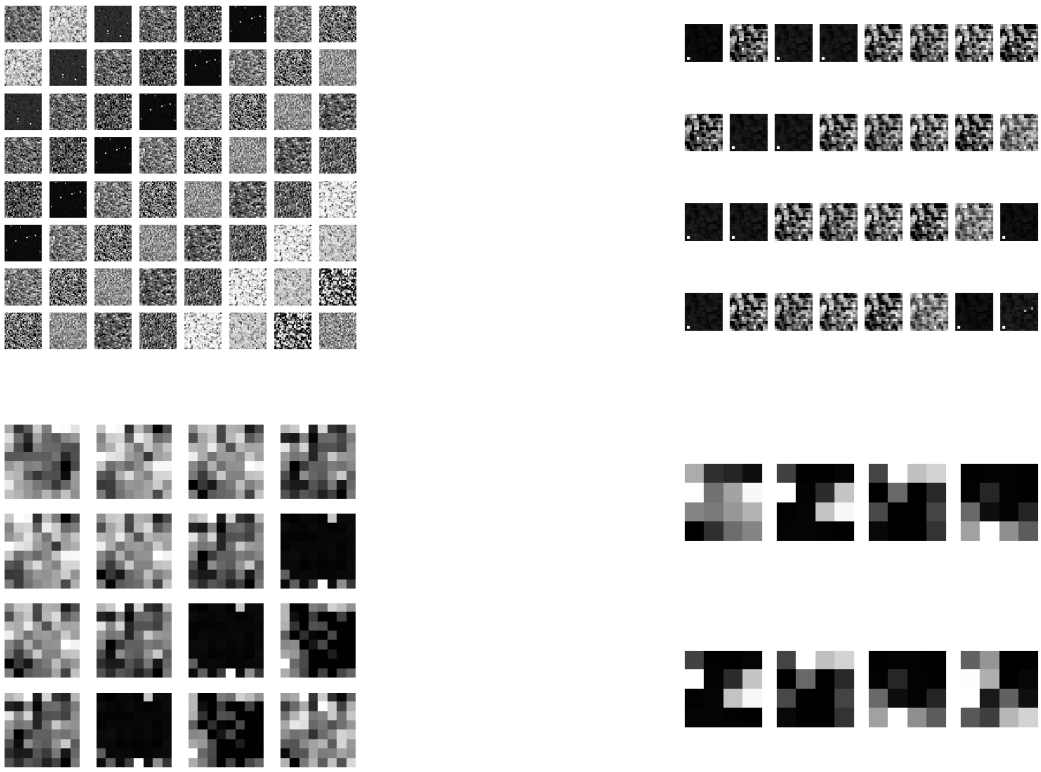


Figure C.9: CAE model (Algorithm 6) second map from Figure C.7 feature maps

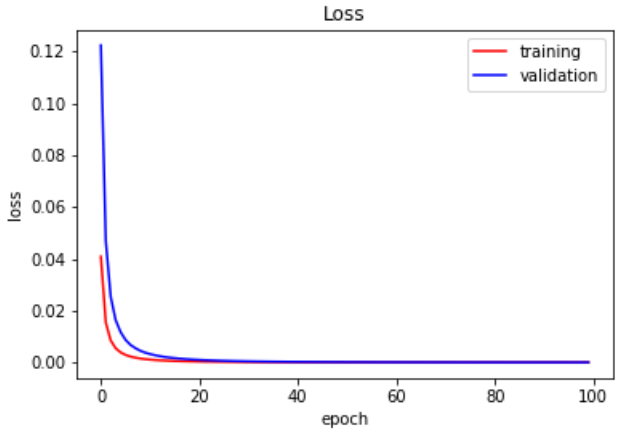


Figure C.10: Training/Validation Loss for CAE model (Algorithm 7) (Train Loss: 0.000002, Validation Loss: 0.000005, Evaluation Loss: 0.000005)

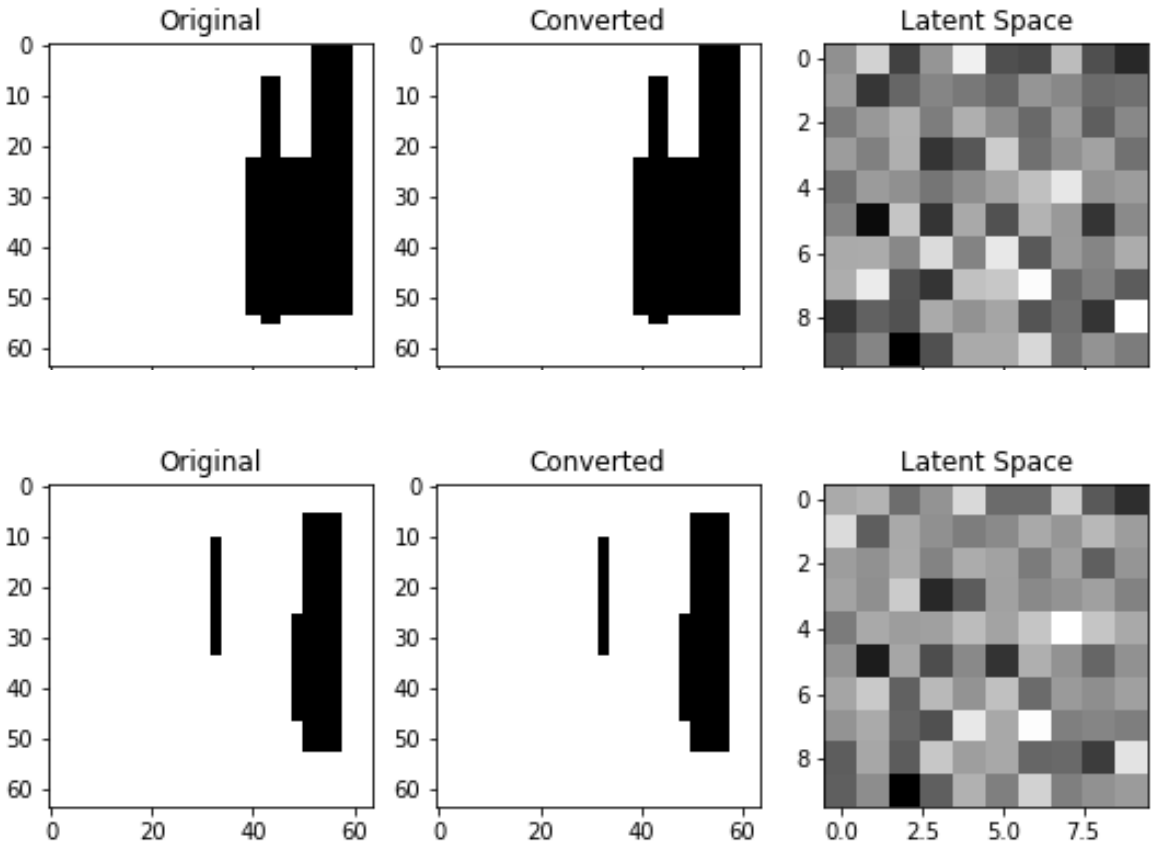


Figure C.11: CAE model (Algorithm 7) network analysis

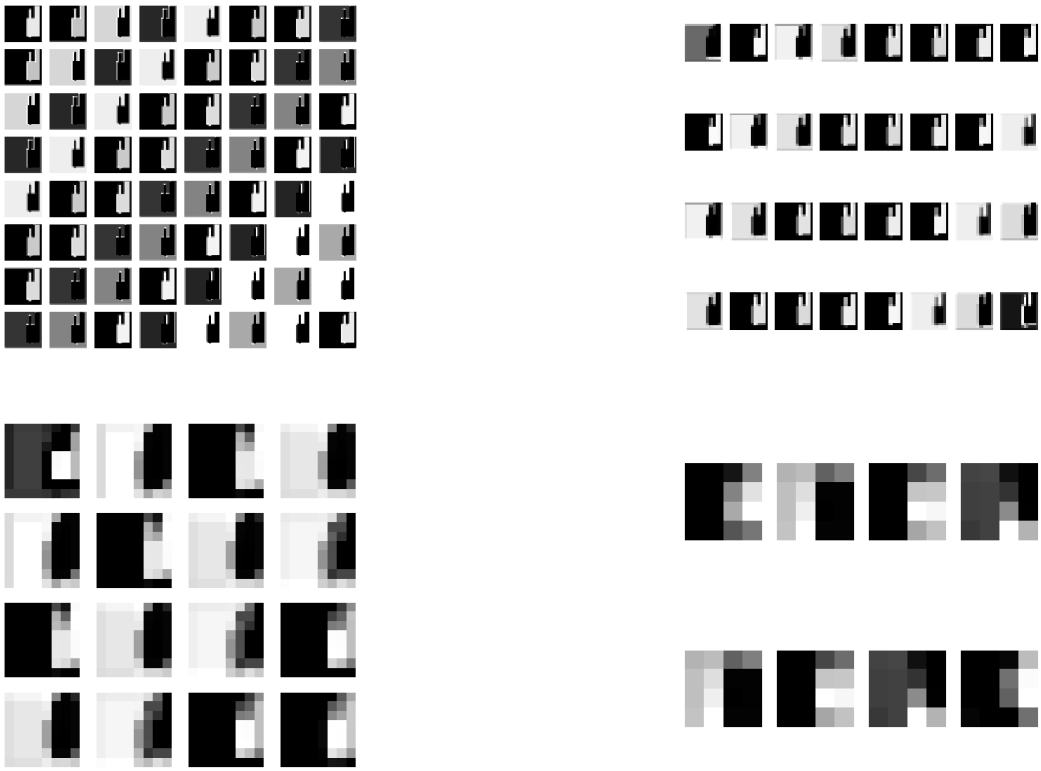


Figure C.12: CAE model (Algorithm 7) first map from Figure C.11 feature maps

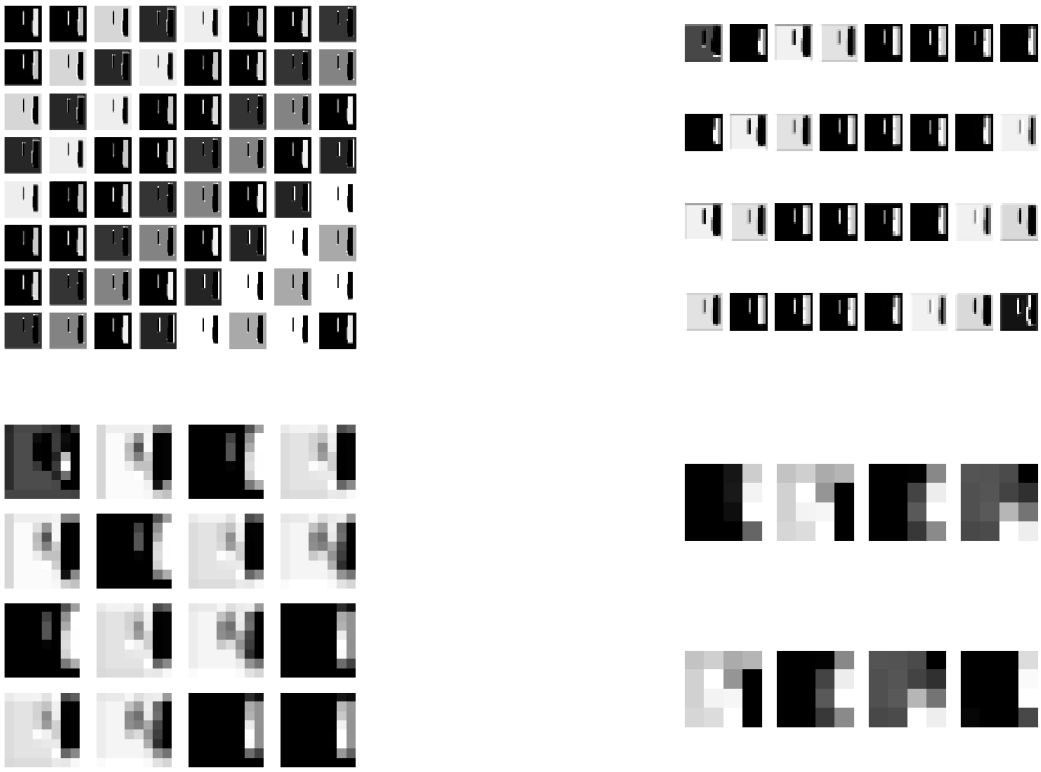


Figure C.13: CAE model (Algorithm 7) second map from Figure C.11 feature maps

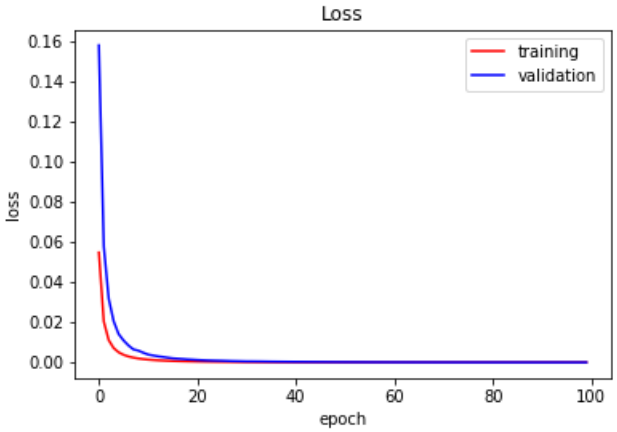


Figure C.14: Training/Validation Loss for CAE model (Algorithm 8) (Train Loss: 0.000002, Validation Loss: 0.000007, Evaluation Loss: 0.000007)

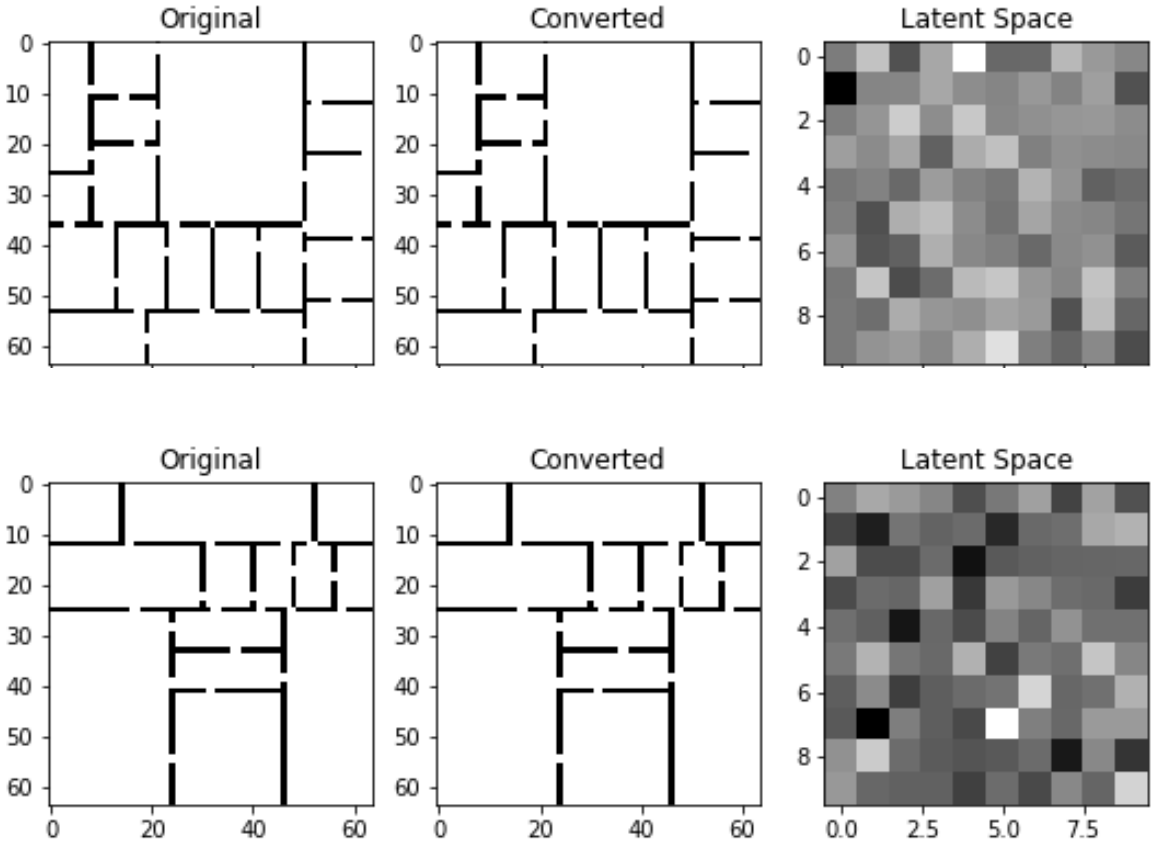


Figure C.15: CAE model (Algorithm 8) network analysis

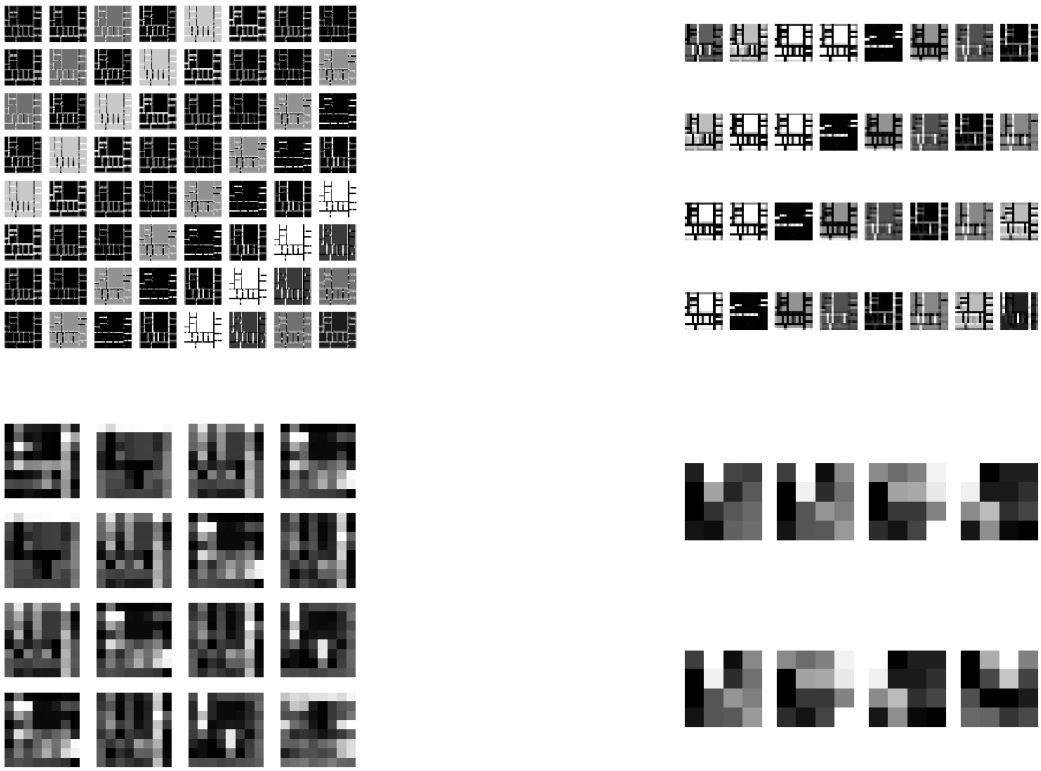


Figure C.16: CAE model (Algorithm 8) first map from Figure C.15 feature maps

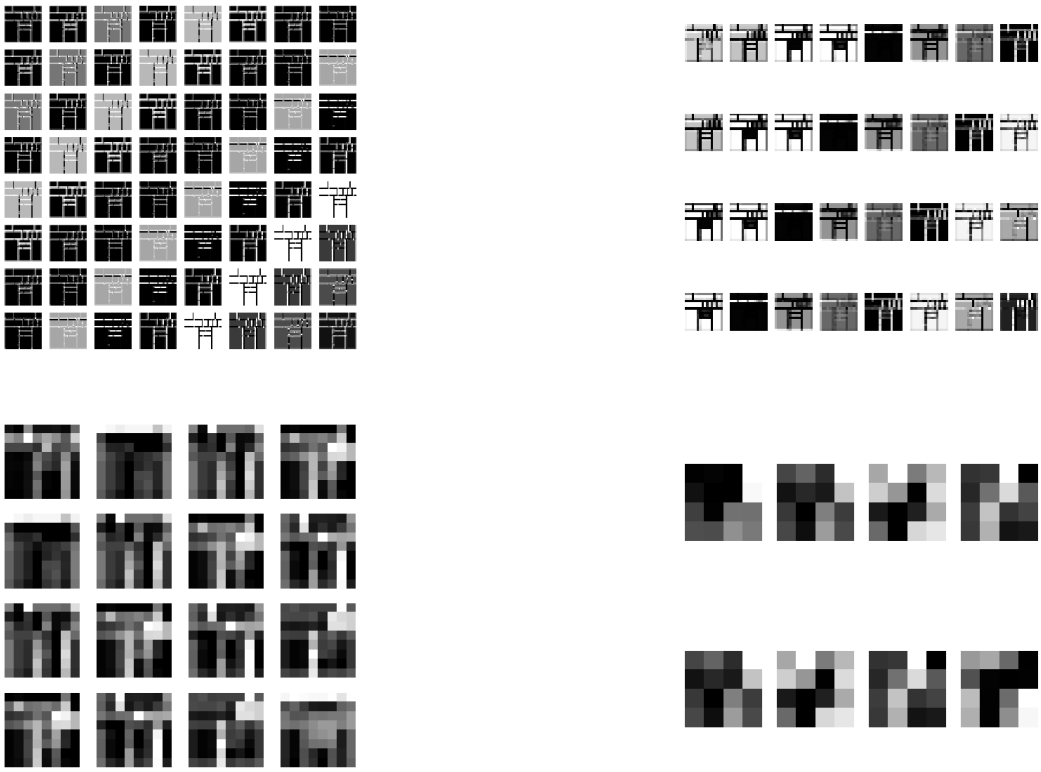


Figure C.17: CAE model (Algorithm 8) second map from Figure C.15 feature maps

Model	Epochs	Training Loss	Validation Loss	Evaluation Loss	Accuracy	Precision	Recall	F1	CM
6	34	0.042641	0.153901	0.178480	0.96	0.96	0.95	0.95	See Table C.10
7	25	0.031109	0.169480	0.145604	0.96	0.96	0.95	0.95	See Table C.11
8	45	0.146206	0.393990	0.610441	0.87	0.88	0.88	0.88	See Table C.12
9	37	0.019767	0.066242	0.097563	0.95	0.95	0.94	0.94	See Table C.13
10	50	0.033152	0.118695	0.096448	0.92	0.92	0.92	0.92	See Table C.14

Table C.9: CAE Online LSTM Planner final training statistics (CM is short-hand for Confusion Matrix)

		Predicted							
	Action	0	1	2	3	4	5	6	7
Actual	0	217	2	0	1	0	0	0	2
	1	2	243	5	3	0	0	0	2
	2	0	2	109	3	1	0	0	0
	3	0	0	2	143	1	0	0	0
	4	0	0	1	1	174	1	3	1
	5	0	0	0	1	0	235	2	3
	6	7	1	0	0	1	1	121	5
	7	2	3	0	0	0	0	0	313

Table C.10: Confusion matrix for Algorithm 6

		Predicted							
	Action	0	1	2	3	4	5	6	7
Actual	0	294	1	0	0	4	0	0	0
	1	2	106	3	1	0	0	0	2
	2	0	0	235	0	0	0	11	0
	3	0	0	0	130	2	0	0	0
	4	5	0	0	1	186	0	0	0
	5	0	0	0	5	4	192	3	1
	6	0	0	3	0	0	1	396	2
	7	3	5	1	0	0	0	10	194

Table C.11: Confusion matrix for Algorithm 7

		Predicted							
	Action	0	1	2	3	4	5	6	7
Actual	0	148	2	0	0	0	1	0	4
	1	1	255	3	4	0	10	2	6
	2	0	15	116	6	0	1	10	4
	3	0	16	4	222	6	9	4	5
	4	10	1	0	2	217	8	0	0
	5	0	2	0	10	6	250	5	20
	6	0	0	3	0	0	1	192	6
	7	5	13	0	1	0	9	8	234

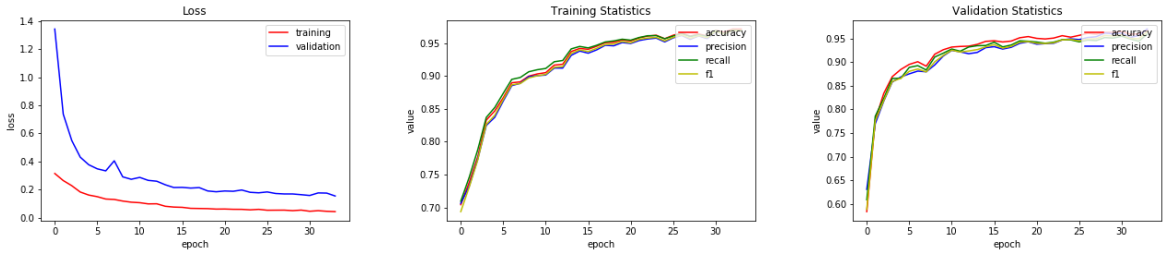
Table C.12: Confusion matrix for Algorithm 8

		Predicted							
	Action	0	1	2	3	4	5	6	7
Actual	0	131	0	0	1	0	0	3	1
	1	8	216	0	0	0	0	1	4
	2	0	1	199	2	0	0	0	0
	3	4	0	3	119	0	0	0	0
	4	0	0	0	0	100	0	0	0
	5	0	0	0	10	0	125	12	1
	6	0	0	0	0	0	5	283	3
	7	3	1	0	0	0	1	11	256

Table C.13: Confusion matrix for Algorithm 9

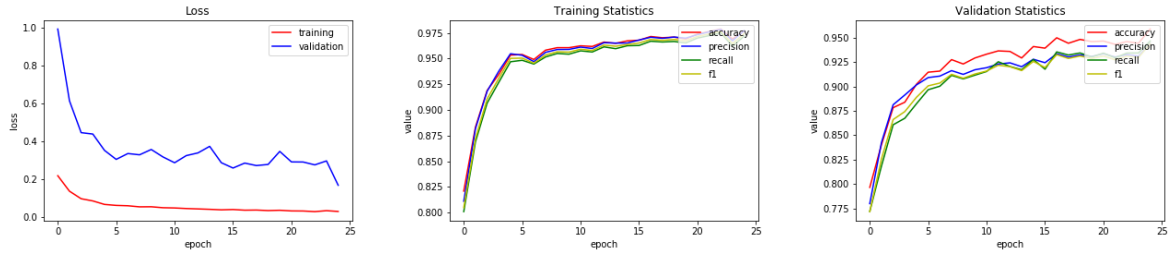
		Predicted							
	Action	0	1	2	3	4	5	6	7
Actual	0	175	0	2	0	0	0	0	2
	1	7	238	8	7	0	0	0	0
	2	0	2	208	1	0	0	0	1
	3	1	2	6	106	4	6	0	0
	4	0	0	0	2	145	4	0	0
	5	1	0	0	2	6	204	5	1
	6	1	0	0	0	0	7	193	5
	7	5	0	0	0	1	17	10	154

Table C.14: Confusion matrix for Algorithm 10



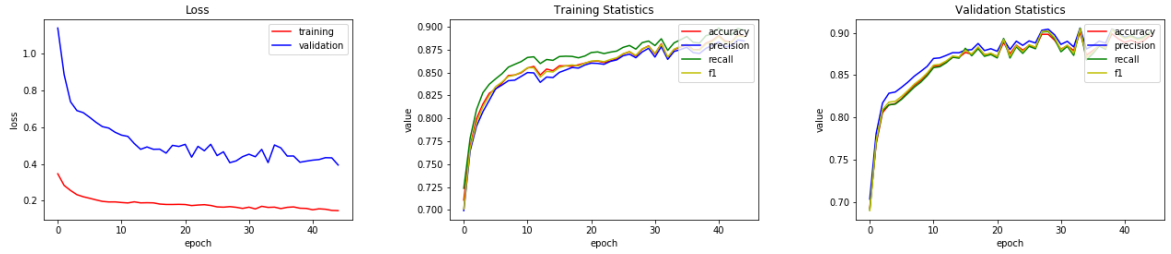
(a) Training/Validation Loss (b) Training Statistics (c) Validation Statistics

Figure C.18: Training statistics for Algorithm 6



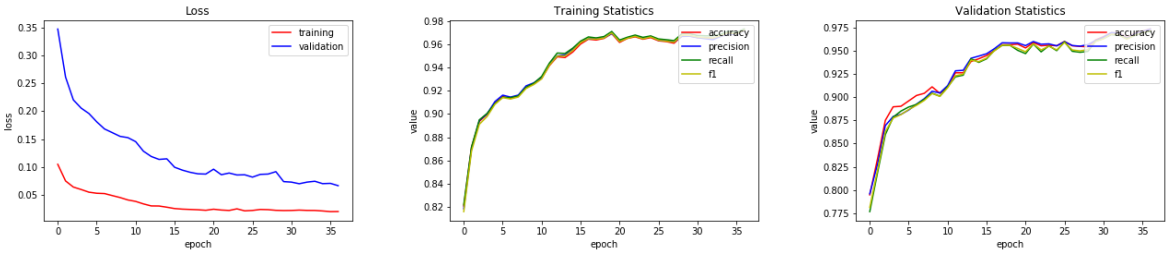
(a) Training/Validation Loss (b) Training Statistics (c) Validation Statistics

Figure C.19: Training statistics for Algorithm 7



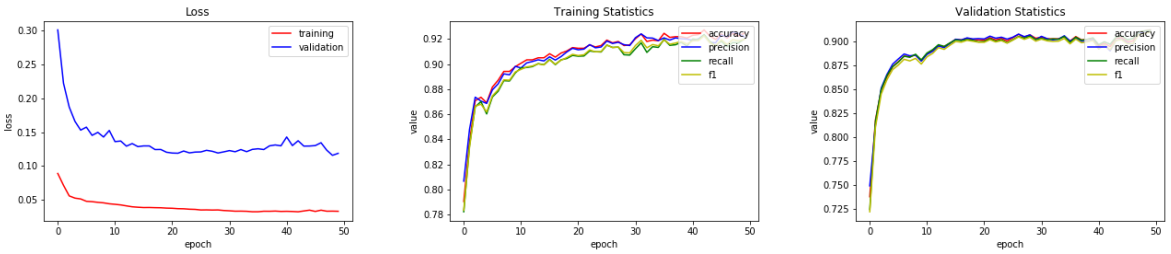
(a) Training/Validation Loss (b) Training Statistics (c) Validation Statistics

Figure C.20: Training statistics for Algorithm 8



(a) Training/Validation Loss (b) Training Statistics (c) Validation Statistics

Figure C.21: Training statistics for Algorithm 9



(a) Training/Validation Loss (b) Training Statistics (c) Validation Statistics

Figure C.22: Training statistics for Algorithm 10