UNIVERSITY OF CALIFORNIA SAN DIEGO

Towards Neural Network Embeddings of Optimal Motion Planners

A thesis submitted in partial satisfaction of the
requirements for the degree
Master of Science

in

Electrical Engineering (Signal and Image Processing)

by

Mayur Joseph Bency

Committee in charge:

Professor Michael C. Yip, Chair
Professor Nikolay A. Atanasov
Professor Sonia Martinez Diaz

2018

DEDICATION

To my family, for their infinite patience, support, and love.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

This work and all the research presented in this document would not have been possible without the support and guidance of my advisor Michael Yip. It's been quite a ride navigating the opaque waters of academia, and it's always nice to know someone on the inside.

It's always important to find people who can help make research not only a successful endeavour, but also a fun and enriching experience. To that end, I was lucky to have been introduced to the people in Advanced Robotics and Controls Lab (ARCLab). In particular and in no particular order, I'd like to acknowledge Nikhil Das, Ojash Neopane, Ahmed Qureshi, Florian Richter, Anthony Simeonov, Jun Zhang, Brian Wilcox, Carlos Nieto-Granda and Andrew Saad for an equal mix of academic discussion and fun delightful conversations.

And finally, I'd like to acknowledge with immense gratitude my parents and my brother for all the patience, sacrifice, and support, without whom this experience would have been impossible.

ABSTRACT OF THE THESIS

Towards Neural Network Embeddings of Optimal Motion Planners

by

Mayur Joseph Bency

Master of Science in Electrical Engineering (Signal and Image Processing)

University of California San Diego, 2018

Professor Michael C. Yip, Chair

Fast and efficient path generation is critical for robots operating in complex environments. This motion planning problem is often performed in a robot's actuation or configuration space, where popular pathfinding methods such as A*, RRT*, get exponentially more computationally expensive to execute as the dimensionality increases or the spaces become more cluttered and complex. On the other hand, if one were to save the entire set of paths connecting all pair of locations in the configuration space a priori, one would run out of memory very quickly. In this work, we introduce a novel way of producing fast and optimal motion plans by using a stepping neural network approach, called OracleNet. OracleNet uses Recurrent Neural Networks to determine end-to-end trajectories in an iterative manner that implicitly generates optimal motion plans with minimal loss in performance in a compact form. The algorithm is straightforward in implementation while consistently generating near-optimal paths in a single, iterative, end-to-end roll-out. In practice, OracleNet generally has fixed-time execution regardless of the configuration space complexity while outperforming popular pathfinding algorithms in complex environments and higher dimensions.

Chapter 1

Introduction

Being able to come up with a quick and accurate motion plan is critical to robotic systems. Motion planning involves finding a connection between two locations while avoiding obstacles and respecting bounds placed on the robot's movement. Such problems include the Piano Mover's Problem of figuring out the best way to move a piece of furniture through narrow corridors and learning obstacle avoidance policies through model predictive control for quadcopters [1]. The majority of work in solving the motion planning problem involves online computation of graphical or grid search strategies that scale poorly with dimensions, or sampling based strategies that scale better with dimensions but are highly dependent on the complexity of the environments. Furthermore, a fundamental trade-off has existed with available algorithms — the trade-off between finding the optimal solution and finding a feasible solution quickly.

The main contribution of this work is a novel approach to the general motion planning problem that leverages a neural-network based path generator, and produces feasible paths in fixed time that mimic an oracle algorithm (one that can always generate the optimal paths across the entire configuration space of the robot/environment, for any start or end goal). Our approach leverages the Recurrent Neural Network (RNN) in order to mimic the stepwise output of an oracle planner, moving from the start to the end location in a relatively smooth manner (Fig. 1.1 gives

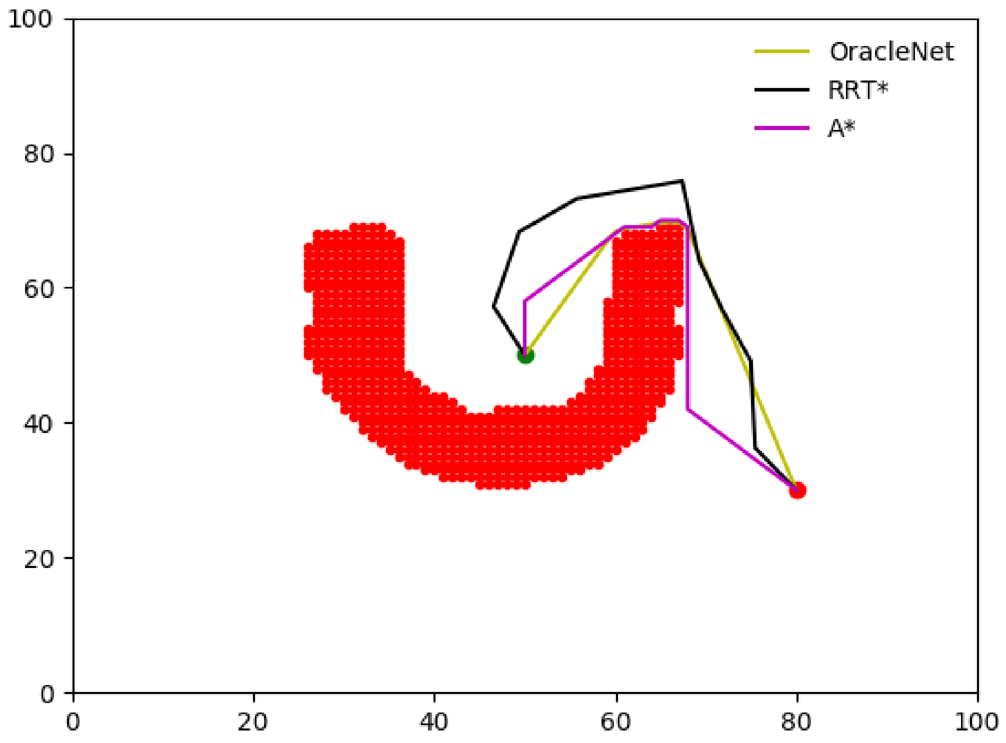


Figure 1.1: An example path generated by OracleNet, shown alongside the results of popular path-finding algorithms A* and RRT*.

an example). Several important advantages of OracleNet that are demonstrated in this work: (1) *it generates extremely fast and optimal paths online*; (2) *it offers a valid path if one exists in practice 100% of the time*; (3) *it has consistent performance regardless of the configuration space complexity*; and (4) *it scales almost linearly with dimensions*. We demonstrate the results of our method on a point-mass robot, 3, 4, 6-degrees of freedom (DoF) simulation robots, and finally on a 7 DoF Baxter robot (both in simulation and with the physical robot). We also show that our algorithm scales close to linear with increase in dimensions, making it significantly faster and more efficient for online motion planning for non-trivial problems, compared to the polynomial or worse time complexity of popular motion planners. In general, this approach offers the starting concept of formulating the path planning problem with a sequential neural network-based solver (i.e. *neural motion planning*) in which many algorithmic variants can be considered, such as those that operate on environments outside the training set, operating on dynamic environments, and operating on dynamical systems that follow complex sets of ODEs/PDEs.

Chapter 2

Preliminaries

Motion planning begins with the concept of the *configuration space* of an agent. Each point in this space is designed to represent complete and unique information about the state(s) of the robot relevant to the planning problem. For example, a simple point-mass robot operating in a 2D grid system will have the planar $x-y$ cartesian coordinates as its configuration space, whereas a 6-DoF holonomous manipulator may have each joint representing a dimension in its 6-dimensional configuration space [1]. The type of configuration space chosen may have stationary, time-varying, constrained, and movable-object representations, as per the requirements of the planner. The problem, therefore, is to find a curve in configuration space that connects the start and goal points and avoid all configuration space obstacles [3].

Motion planning for an agent operating in a configuration space can take the form of path planning or trajectory planning. Path planning is tasked with generating a set of points in the configuration space and is not concerned with how the agent is following those points. On the other hand, trajectory planning describes how the configuration of the agent evolves over time as well, with a trajectory being a time-parametrized function defined over a planning horizon [4]. In this work, we restrict ourselves to path planning and use the terms motion planning and path

[1]Configuration spaces in general are *non-Euclidean* [2].

planning interchangeably, with the possibility of extending to trajectory planning for modelling dynamical systems being discussed in Chapter 6.

This chapter is aimed at establishing the following two goals for the reader. Section 2.1 seeks to establish a backdrop for the problem that is being attempted in this thesis, mainly by briefly describing some of the so-called “classics” of motion planning algorithms. Section 2.2 continues further by describing modern approaches to motion planning with a focus on the state of learning-based algorithms and where the algorithm proposed in this work sits.

2.1 Background

The notion of *completeness* is important when discussing the effectiveness of motion planners. A motion planning algorithm defined according to Chapter 1 is said to be complete if a solution can be returned in a finite amount of time, if one exists, and return failure otherwise. However, such algorithms are often computationally expensive and inefficient and therefore are rarely used in practice. The Piano Mover’s Problem is one such example of a PSPACE-hard [2] problem in finding complete solutions.

The need for practical solutions led to the idea of *resolution completeness*, which redefines or rather relaxes the completeness condition to return a valid solution based on the resolution parameter of the algorithm is set fine enough. Several kinds of these algorithms exist, such as the following.

- *Graph-search methods* discretize the configuration space of the agent as a graph (in the special case of the nodes being orthogonal to each other, it may also be referred to as a grid), where the nodes represent a finite collection of accessible agent configurations and the edges represent transitions between nodes. Each edge may have a cost (possibly

[2]PSPACE is denoted as a class of problems that can be solved in polynomial space (also known as computer memory). PSPACE-hard problems are known to be computationally intractable and therefore, probably not worthwhile to search for an efficient algorithm to search for an optimal solution. Instead, developing approximations and heuristics would be more appropriate. Note that PSPACE-hard also implies NP-hard.

directional) associated with its transition. Given queried start and goal nodes, the path planning problem is then defined as solving for a minimum-cost path in such a graph, with a path being defined as a sequence of nodes to be followed starting from the start and terminating at the goal. Graph search methods are limited to optimize only over a finite set of paths, namely those that can be constructed from the node-traversals allowed to the agent.

- *Artificial potential fields* have also been proposed as an alternative to graph-search methods for their speed and relative extensibility to higher dimensions. Khatib [5] pioneered the use of the artificial potential field method in the context of obstacle avoidance, with their approach involving the use of potential fields in work space [3] as opposed to configuration space. Repulsive potential fields are placed around obstacles and forbidden regions, and an attractive potential field is placed around the goal, with the reasoning that the agent experiences a generalized force equal to the negative of the total potential gradient. This force drives the agent "downhill" towards its goal configuration until it reaches a minimum and the optimization terminates. As with any optimization problem with ill-defined convexity, the main disadvantage of this approach is the tendency for the agent to settle in local minima. Although several heuristics have been proposed, the problem has not been fully solved yet.

- *Cell decomposition* methods work by partitioning the configuration space into disjoint sets, called cells [3]. These cells form the nodes of a non-directed connectivity graph *G*. Two nodes are adjacent if they share a common boundary and an adjacency graph is computed [4]. After the space decomposition into cells has been completed, the planner uses the adjacency

[3] Work space for an agent is defined as the physical world it interacts with. As opposed to configuration space which can span an arbitrary number of dimensions depending on the agent, work space does not go beyond three dimensions.

[4] An adjacency graph encodes the adjacency relationship of cells, where a node corresponds to a cell and an edge connects nodes of adjacent graphs

graph to search for a path connecting the cells containing the start and goal. The most popular cell decomposition technique is the trapezoidal decomposition [6] where the planar configuration space is split into polygonal regions, with 2^m-tree decomposition used for higher dimensions. There are chiefly two disadvantages to this cell decomposition. First, the complexity of these algorithms is exponential in the dimensions of the configuration space. Second, there is a high cost associated with determining whether a cell is entirely contained within the configuration space or not.

Compared to *complete* planners, these planners based on *resolution completeness* demonstrated remarkable performance in accomplishing various tasks in complex environments within reasonable time bounds [7]. However, as pointed out for each of the described methods above, their practical applications were mostly limited to configuration spaces with up to five dimensions, since graph search and decomposition-based methods suffered from large number of nodes and cells, and potential field methods from local minima. Part of the reason why the aforementioned algorithms struggle with the curse of dimensionality is having to know complete and explicit information of the configuration space. This may also scale poorly with a large number of obstacles.

Sampling-based motion planning algorithms were developed as a way to avoid this problem. Lindemann and LaValle [8] provides an extensive look at sampling-based planners. Due to their effectiveness and computational efficiency, sampling-based planners have gained the status of being the preferred kind of planners for all sorts of planning problems, from static two-dimensional planar grid-maps to complex high dimensional robots in dynamic environments. Instead of using an explicit representation of the configuration space, sampling-based planners rely on a collision-checking module. The planner repeatedly samples the configuration space and adds collision-free points to a roadmap of feasible trajectories. providing information. The roadmap is then used to construct the solution to the original motion-planning problem.

As a further approximation to the *completeness* problem, the concept of *probabilistic*

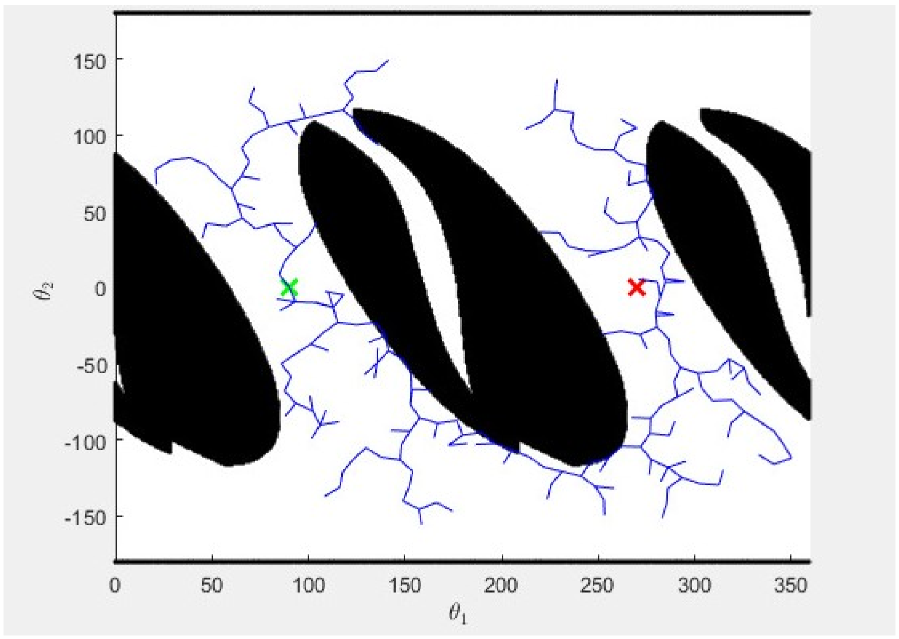


Figure 2.1: An example of the Rapid-exploring Random Tree algorithm being used on a sample two-dimensional configuration space to plan between two points (green being start and red being goal). The configuration space here is the joint-space for a 2-link planar manipulator, with θ_1 and θ_2 representing the joint angles of the manipulator in degrees.

completeness was introduced with sample-based planners. This guarantees that the probability of the planner returning a valid solution, should one exist, converges to one as the number of samples approaches infinity. Moreover, Barraquand et al. [9] proved that the rate of decay of the probability of failure is exponential, under the assumption that the environment has good “visibility” properties [5]. Fig. 2.1 shows a sample-based planner run on a configuration space. Note that the planner itself does not know anything about the obstacles until the collision checker returns a value.

[5]Barraquand et al. [9] analyzed the relation between the probability of failure and running time as a response to the explosion of sampling-based randomized planners in the late nineties and their ability to seemingly perform well with virtually any type of robots and empirically observed success. They define two metrics for the “goodness”” of a configuration space and under the assumption that those two metrics hold, the running time is shown to grow only as the absolute value of the logarithm of the probability of failure.

2.2 Related Work

A range of techniques to solve the motion planning problem has been proposed in the past two decades, from algorithms that emphasize optimality over computational expense, to those that trade-off computational speed with optimality. This trade-off is in part to meet the need of problem spaces that motion planners are trying to solve today, which involve solving for complex, high-dimensional, dynamically-constrained systems and environments. Traditional algorithms such as A* [10] that search on a connected graph or grid, while fast and optimal on small grids, take exponentially longer to compute online with increasing grid sizes and environment complexity. Sampling-based strategies such as RRT have better computational efficiency for searching in high dimensional spaces [11] but get slowed down by their "long tail" in computation time distribution in complex environments. Apart from using grid-based and sampling-based motion planners, optimizing over trajectories has also been proposed, with approaches such as using potential fields to guide a particle's trajectory away from obstacles [5] and reformulating highly non-convex optimization problems to respect hard constraints [12]. A review of recent algorithms and performance capabilities of motion planners can be found in González et al. [13].

The challenge of creating and optimizing motion plans that incorporate the use of neural networks has long been a problem of interest, though computational efficiency in solving for deep neural networks has only recently made this a practical avenue of research. Glasius et al. [14] was an early attempt to link neural networks to path planning by specifying obstacles into topologically ordered neural maps and using neural activity gradient to trace the shortest path, with neural activity evolving towards a state corresponding to a minimum of a Lyapunov function. More recently, Chen et al. [15] proposed a method that enables the representation of high dimensional humanoid movements in the low-dimensional latent space of a time-dependent variational autoencoder framework.

Reinforcement Learning (RL) approaches have also been proposed for motion planning applications. Tamar et al. [16] introduced a fully differentiable approximation of the value-iteration algorithm capable of predicting outcomes that involve planning-based reasoning. However, their use of Convolutional Neural Networks (CNN) to represent this approximation limits their motion planning to only 2D grids, while generalized motion planning algorithms can be extended to arbitrary dimensions.

Learning to generate motion plans has also been considered via a Learning-from-Demonstration (LfD) approach. Using an expert (usually human) to provide demonstrations of desired trajectories, LfD methods are able to generalize within the set of demonstrations an approximate, underlying sequence or policy that reproduces the demonstrated behavior. LfD has been successfully applied in various situations that involve challenging dynamical systems or nuanced activities such as autonomous helicopter aerobatics [17], emulating gestures [18], and making coffee [19]. Our path generating algorithm can be considered an extension of LfD since the lines between motion planning, LfD, imitation learning, and model predictive control are getting blurred with advancements in machine intelligence.

Chapter 3

Methods

3.1 Problem Definition

In this work, we use the standard definition of configuration spaces (c-spaces) to construct the environment in which our motion planning algorithm operates. For a robot with d degrees-of-freedom (DoF), the configuration space represents each DoF as a dimension in its coordinate system. Each d-dimensional point in the c-space represents the d joint angles of the robot and therefore, the full configuration of the robot in the real world. Due to this property, motion planning in c-spaces is simpler than in geometric spaces. A motion planning task involves connecting two points in a d-dimensional c-space, with obstacles mapping from Cartesian space to c-space in a nonlinear fashion through forward-kinematic collision checks. The GJK algorithm for collision detection developed by Gilbert et al. [20] is a popular method used to determine if a given point in c-space is in obstacle region or not. A necessary assumption for our algorithm is knowledge of the configuration space, which is defined by a given robot present in a given static constrained environment. For example, these problems arise in solving for optimal routes in a large network of roads or navigating through the lanes of a crowded warehouse.

Let $X \subset R^d$ be the c-space. Let $X_{obs} \subset X$ be the obstacle region, such that $X \backslash X_{obs}$ is an

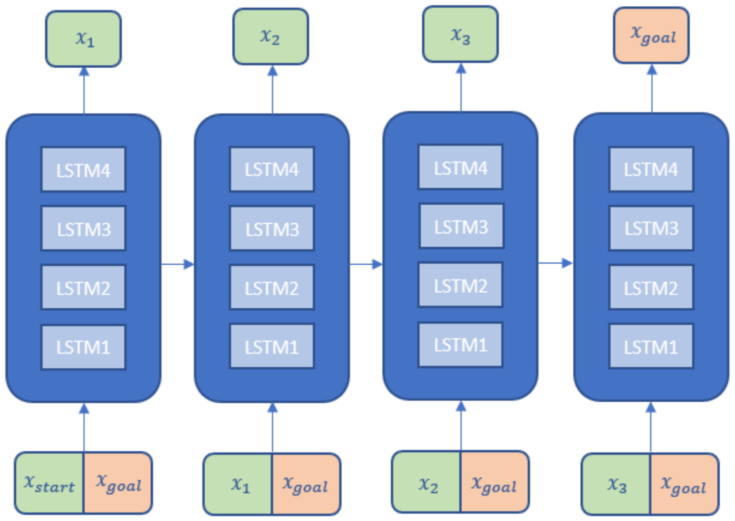


Figure 3.1: An "unfolded through time" strategy for motion planning is proposed in OracleNet. Note that within this RNN structure, the Long Short Term Memory (LSTM) hidden layer weights are shared across timesteps as the inputs get iteratively updated and concatenated.

open set, and denote the obstacle-free space as $X_{free} = cl(X \setminus X_{obs})$, where $cl()$ denotes the closure of a set. The initial start point x_{start} and the goal x_{goal}, both elements of X_{free}, are provided as query points. The objective here is to find a collision-free path between x_{start} and x_{goal} in the c-space. Let x be defined as a discrete sequence of waypoints. x is considered valid if it is continuous, collision free (each waypoint in the generated path should lie in X_{free}), and feasible ($x(0) = x_{start}, x(t) = x_{goal}$ for some finite t).

3.2 Proposed Algorithm

We propose to solve the problem of generating goal-oriented path sequences by passing in a goal location as an auxiliary input at each step of the prediction, which provides a reminder to the network about where it should ultimately converge. At each step, the input vector is concatenated with the desired goal location and the resulting augmented input vector is used to train an RNN model. The RNN is trained on optimal trajectories that span the entire configuration space, which allows it to internalize an oracle-mimicking behavior that generates optimal path sequences during rollouts. The network comprises of stacked Long Short Term Memory (LSTM) layers that preserve information over a horizon of outputs [21], with the output layer being fully connected to the final LSTM hidden layer. This overcomes the issue of vanishing gradients in vanilla RNN when the required memory length is more than a few steps, decreasing its memory capacity. In this work we concern ourselves with fixed short-term memory; our experiments showed that using LSTMs resulted in better performance over vanilla RNNs, even though training LSTMs are usually more difficult. Our approach may be used to encode dynamical system behaviors as well, though we do not explicitly explore this application in this paper. Fig. 3.1 is a schematic of the network architecture used in our approach. The error signal which is used to train on is the mean square error (MSE) between the predicted output and the teaching signal. The number of layers was decided on by empirically converging to an appropriate size based on the dimensionality of the problem to be solved. Increasing the number of layers as we increase the dimension of the c-space to capture additional degrees of freedom in the training set lead to better performance. A practical limit in expanding network size is given by Funahashi and Nakamura [22] that proved that any finite time trajectory of a given n-dimensional dynamical system can be approximately realized by the internal state of the output units of a continuous time recurrent neural network with n output units, some hidden units, and an appropriate initial condition. Exact numbers for network size and depth are provided in Chapter 4.

3.3 Training Set Creation and Offline Training

A training set consisting of a number of valid paths created by an "expert" planner, which we call the Oracle. We used A* as our expert to generate an optimal set of paths for training. The c-space is sampled to create a graph with connected nodes. Two nodes are randomly selected without replacement (based on a uniform distribution) from the set of nodes present in X_{free}

and A* is executed to find the optimal path connecting them. This process is repeated N times to obtain a training set consisting of N "expert" paths. Each generated path is split into their composing waypoints to make each individual waypoint represent a sample in the training set. The teaching signal corresponding to each sample then becomes the next waypoint in the path sequence. That is, if x is a path with τ waypoints, the path is split into $\{x(0), x(1), ..., x(\tau-1)\}$ and the corresponding teaching signals become $\{x(1), x(2), ..., x(\tau)\}$. If we assume that $x(\tau)$ is the goal point in a sequence, then the auxiliary input is concatenated as $\tilde{x}(t) = [x(t), x(\tau)]$, where t ranges from 0 to τ. The LSTM network is trained on the $\tilde{x} \forall t, N$.

3.4 Online Execution through Bi-directional Path Generation

Given a trained network, for testing, we select two points y_{start} and y_{goal} (not from the training set) from X_{free} and attempt to roll out a path connecting them while avoiding obstacles. The network generates a sequence of waypoints until a final connection to y_{goal} occurs to complete the path. After the process is terminated, all the sequentially generated outputs are formatted as waypoints in the generated path. To make the path generating process more robust, a bi-directional path generation is used [1]. We start the generation process from y_{start} and y_{goal} points simultaneously and make the two branches grow towards each other. The process is terminated when the two branches meet, and the branches are then stitched together to form a complete path. Fig. 3.2 shows the bidirectional stepping behavior. Instead of forcing the path to "grow" towards an arbitrarily selected fixed goal point, the network now has the option to target a constantly shifting goal point (the current point of the other branch). This increases the chance of the convergence point falling with the Oracle paths on which OracleNet was trained, thus increases feasibility and success rates, while having no impact on path roll-out time.

[1]This is not to be confused with bidirectional RNNs [23]

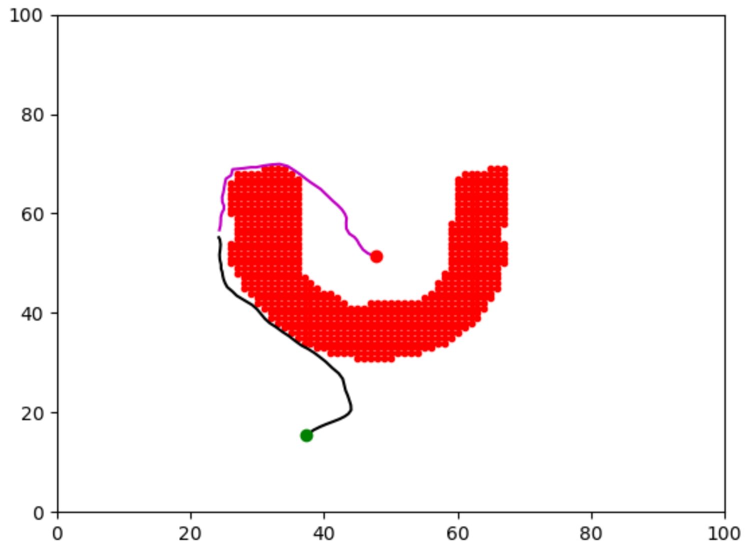


Figure 3.2: Example of bi-directional path generation (not rewired). The green and the red marker are the start and the goal respectively. The black path grows from the start while simultaneously, the magenta path grows from the goal. The process is terminated when the heads of the paths get within a set distance of each other.

3.5 Repair and Rewire

It is to be expected that OracleNet will never yield an exact duplication of an Oracle's behavior. Also, since the expert demonstration paths are sampled randomly on the c-space, there may be regions in the c-space that the network has never seen in the dataset. Because of these unseen regions, the generated waypoints may not adhere strictly to obstacle boundaries and cut corners through obstacle regions at times. This in practice happens less than 3% in all paths generated by the network. Naive methods may exist such as obstacle padding. However, we propose a *repair* module to fix violating waypoints as they appear online while generating the path. The repairing strategy used is straightforward to implement. When current waypoint $x(t)$ is generated inside an obstacle region, a direction is randomly selected at a step distance ε from x(t-1) reach a new candidate $x_{new}(t)$. Random samples are taken until the first feasible $x_{new}(t)$ is found. $x_{new}(t)$ then replaces x_t and then OracleNet continues. An assumption on the step size ε is made to such that it will not cross over small features in the configuration space. With an

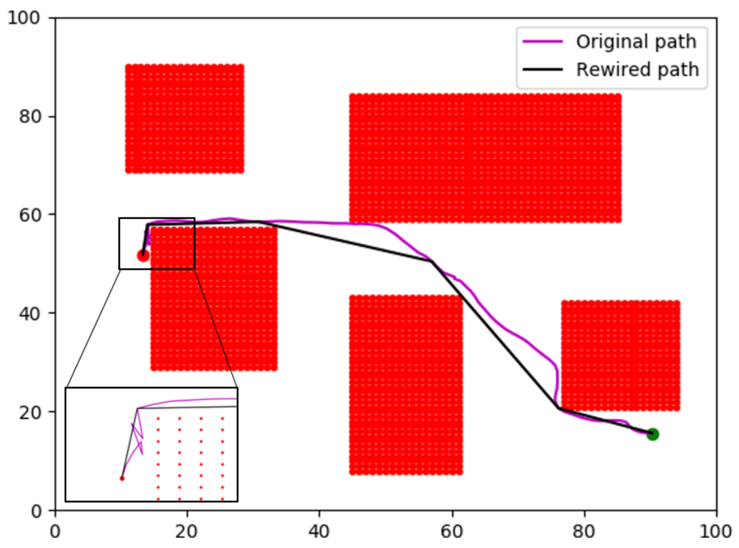


Figure 3.3: Example of bi-directional, repaired, and rewired path generation. Figure inset shows the repair module in action (magenta line). After the path is repaired and converges successfully, the rewire module is called to remove superfluous nodes and any kinks the repairing may have introduced (black line).

appropriately chosen ε, feasible paths are generated 100 % of the time in practice. Alternatively, one can produce candidates $x_{new}(t)$ by running a single pass of $\tilde{x} = [x(t-1), R]$ where $R \in X_{free}$ is a randomly selected goal state. This results in a random walk within the solution space of the network that effectively produces a similar effect, without requiring the need to define a new parameter ε. A final strategy is to generate a new $x(t)$ using a one-step output of a motion planner such as RRT. We chose to take results from the ε method as it was sufficient and fast in achieving 100% success rate in practice with minimal effect on performance.

The repair methods above, along with general network noise, will result in paths that can be non-smooth. To deal with this, we propose a *rewiring* process that removes unnecessary nodes in the paths by evaluating if a straight trajectory connecting two non-consecutive nodes in the path is collision-free. Similar to the rewiring algorithm used in RRT*, a lightweight implementation of this algorithm has very little processing overhead and thus can be used without a noticeable increase in path generation times. It is interesting to note that the final path thus generated and processed has a practical time complexity of $O(n)$ (as shown through results in Section III).

The full Algorithm for OracleNet in online path generation is provided in Algorithm 1.

Algorithm 1 OracleNet (Online Path Rollout)

```
1: procedure OracleNet($x_{start}, x_{goal}$)
2:     $G \leftarrow TrainedNetwork()$                    ▷ OracleNet is assumed to be trained
3:     $x_{current} \leftarrow x_{start}$
4:     $path \leftarrow []$
5:     while $x_{current} \neq x_{goal}$ do
6:         $x_{current} \leftarrow G([x_{current}, x_{goal}])$
7:         if $Obstacle(x_{current})$ then
8:             $x_{current} \leftarrow Repair()$
9:         end if
10:        $path.append(x_{current})$
11:    end while
12:    $path \leftarrow Rewire(path)$
13:    return $path$
14: end procedure
```

Chapter 4

Results

Table 4.1: Speed and Optimality of OracleNet benchmarked against A* and RRT* in a 2D environment

	Completion Time			Optimality (Ratio of Path Lengths$^+$)	
Environment	A* (s)	RRT* (s)	OracleNet (s)	OracleNet / A*	OracleNet / RRT*
Simple 1	0.08 (0.06)	1.98 (3.68)	0.13 (0.18)	0.94 (0.09)	0.84 (0.16)
Simple 2	0.09 (0.07)	1.23 (1.76)	0.24 (0.18)	0.95 (0.02)	0.86 (0.11)
Simple 3	0.07 (0.06)	0.93 (2.9)	0.16 (0.19)	0.955 (0.02)	0.86 (0.10)
Simple 4	0.08 (0.05)	1.53 (2.43)	0.18 (0.20)	0.96 (0.09)	0.87 (0.12)
Difficult 1	0.07 (0.05)	2.69 (3.54)	0.18 (0.10)	0.96 (0.03)	0.87 (0.10)
Difficult 2	0.07 (0.05)	3.67 (6.31)	0.18 (0.12)	0.96 (0.03)	0.88 (0.12)
Difficult 3	0.09 (0.06)	5.17 (11.57)	0.17 (0.12)	0.95 (0.21)	0.87 (0.12)
Difficult 4	0.05 (0.04)	8.79 (12.81)	0.18 (0.09)	0.94 (0.14)	0.89 (0.11)

Values are listed as "mean (standard deviation)" for 8 different environments.
$^+$*Lower is better. Below 1 means that OracleNet produces shorter paths.*

To appropriately evaluate the performance and capability of OracleNet, we test it on a number of distinct environments. The experiments were conducted in a 2D Gridworld with a point robot having translation capabilities only, 3-link, 4-link, and 6-link robot manipulators. For all experiments presented here, training is accomplished with Tensorflow [24] and Keras, a high-level neural network Python library [25], with a single NVIDIA Titan Xp used for GPU acceleration.

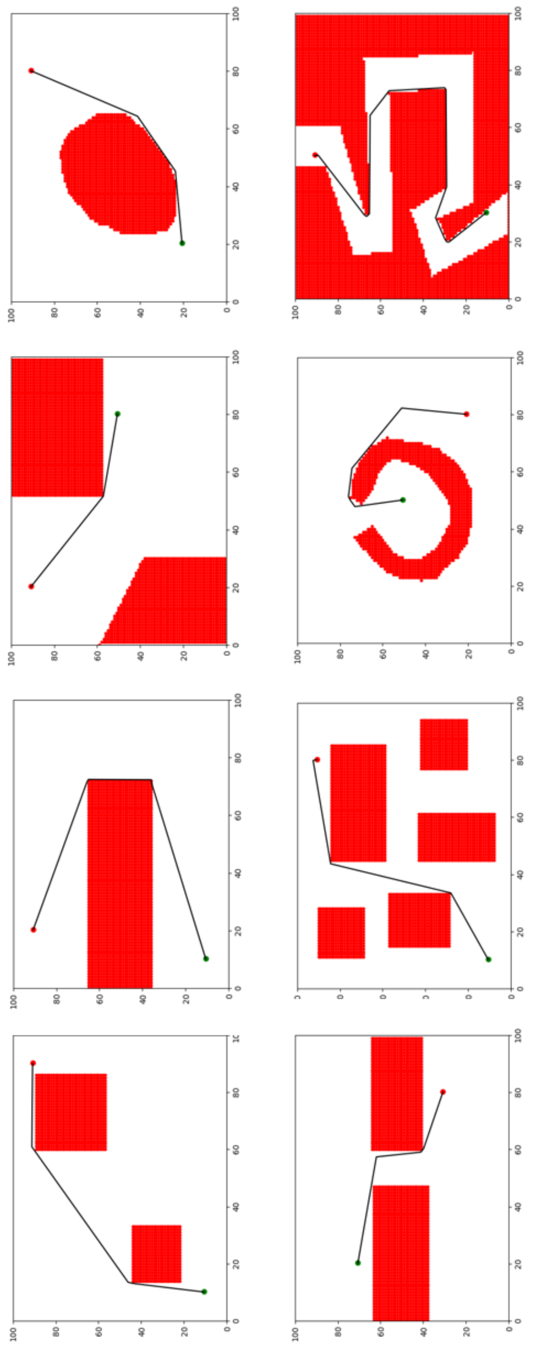


Figure 4.1: 8 environments are used in the 2D Gridworld experiments. The top row has “simple” environments, numbering from the left, while the bottom row has “difficult” environments. Each environment shows a single roll-out of OracleNet. Unlike pathfinding algorithms, no expanding search is required.

4.1 2D Gridworld

Fig. 4.1 shows snapshots of 8 environment examples, 4 considered “simple” environments for popular motion planning strategies such as RRT*, and 4 “difficult” environments. An environment is considered simple if the obstacles are convex and widely spaced apart, while difficult environments consist of either a large number of obstacles or highly non-convex obstacles forming narrow passageways. Each continuous space environment is 100 units in length and width. A* is run on a unit grid of 100 x 100 on this environment to generate 20,000 valid Oracle paths for training OracleNet. This set was split in accordance to the 80-20 rule, with 20% being kept for testing to make sure overfitting does not occur. The network architecture consists of 4 LSTM layers each of 256 hidden units. Due to the similarity in model architectures and dataset sizes, all 8 cases took 5.5 hours to converge.

To benchmark our algorithms performance with existing motion planning algorithms, we use RRT* and A*. RRT* is a version of RRT where heuristic cost is added to encourage optimality of the path in a sampling-based continuous space. Three performance metrics are used: success rate, roll-out time, and path optimality. A generated path is considered successful if none of the waypoints encroach into obstacle region. Roll-out time measures the time taken for the network to generate waypoints from start to goal (or, in the bidirectional case, the time taken for both branches to meet). Path optimality is simply the fraction of the path length generated by OracleNet when benchmarked against paths generated by A* and RRT* respectively. 1000 randomly initialized trials were conducted for each of the 8 environments. Table 4.1 shows that OracleNet manages to be comparable to A* and faster than RRT* even for a small 2D grid while being slightly more optimal in both cases. This is due to the rewiring module and the network being able to generate points in continuous space as opposed to being restricted to a discrete grid in A*. Thus, the proposed algorithm may be a suitable alternative even for small grids with limited connectivity, if path optimality is the priority.

4.2 3-Link Planar Manipulator

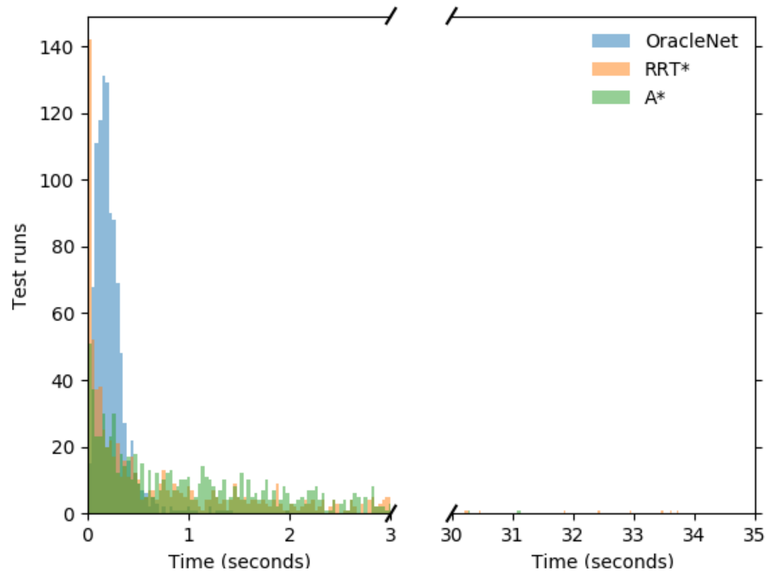


Figure 4.2: A histogram of the test cases used to evaluate performance on a 3-link arm. Refer Table 4.2 for means and standard deviations. Note the Gaussian shape of the distribution for OracleNet, compared to the left-skewed exponential distributions of A* and RRT*. Higher standard deviations for A* and RRT* cause the generation times to have a much wider spread, while OracleNet's much tighter spread indicates its consistency in performance and near fixed-time execution.

To demonstrate the extensibility of the proposed method to higher dimensions, we tested the algorithm on a 3-link manipulator. The base link has movement range 0 to 2π while the subsequent links can move between $-\pi$ to π. To train the RNN, we discretized the 3-dimensional joint angle c-space into a 3D uniform grid with 50 nodes on each axis, resulting in a total of 50^3 = 125,000 uniformly spaced nodes. As in the 2D case, A* is used to generate the training set. To get an accurate representation of the complete c-space in the training set (which directly correlates to the increased number of nodes in the grid used here for training), we use 400,000 examples paths and follow the same steps described for the 2D case. Keeping in mind the increased number of dimensions to learn, we updated the architecture to have 6 layers with 256 units each. For our performance evaluation, we randomly generate 1000 pairs of start and goal locations in continuous c-space. Paired with the repair and rewire modules discussed in the previous sections, we observed a 100% success rate in finding feasible paths. Generation times and path optimality is benchmarked against A* and RRT*. A* for the 3D case, given space connectivity in 3D space, is allowed to have a maximum of 26 neighbors for maximum path optimality. Table 4.2 shows that OracleNet scales much better than pathfinding algorithms such as A* and RRT*. An interesting observation to note is the low standard deviation of path generation times for OracleNet. This is further expanded on in the histogram of the test cases shown in Fig. 4.2. This is indicative of the consistency of OracleNet in producing its paths across the entire c-space, whereas A* and RRT* are heavily influenced by the relative locations of the query points and the obstacles. More about this is discussed in Section V.

4.3 4-Link Planar Manipulator

To further study performance scaling in higher dimensions, OracleNet is implemented for 4-link robot manipulator, keeping the link geometries and constraints from before. Note that even though the workspace appears similar to the one used in the previous experiment, the c-space completely changes due to the extra link. Each joint angle axis of the c-space is discretized into 40 uniformly spaced nodes, giving rise to a total of 40^4 = 2.56 x 10^6 nodes. The network architecture was updated to a deep network consisting of 6 stacked LSTM layers, each with 400 units. As before, the training set for OracleNet is generated using A*, with a maximum of 1 million valid paths used to train the network. 1000 test cases randomly sampled through the continuous c-space, and 100 % success rate is achieved. Table 4.2 shows the relevant performance statistics.

Table 4.2: Performance of RNN-based motion planners in higher dimensions benchmarked against A* and RRT*

	Completion Time			Optimality (Ratio of Path Lengths$^+$)	
Environment	A* (s)	RRT* (s)	OracleNet (s)	OracleNet / A*	OracleNet / RRT*
3-link (3D)	2.707 (4.28)	3.83 (6.68)	0.22 (0.21)	1.01 (0.14)	0.75 (0.20)
4-link (4D)	61.87 (95.07)	18.21 (14.76)	1.18 (0.87)	0.99 (0.13)	0.86 (0.11)
6-link (6D)	-	29.32 (6.25)	1.24 (0.72)	-	0.85 (0.17)

Values are listed as "mean (standard deviation)".
$^+$Below 1 means that OracleNet produces shorter paths.

4.4 6-Link Planar Manipulator

Next, OracleNet is demonstrated on a 6-link robot manipulator. To get a reasonably sized grid for discretization the c-space, we used a very sparse resolution of 10 uniformly spaced samples per axis, making a total of 10^6 nodes in the uniform grid. Similar to the previous experiment, 1 million valid paths generated using A* was used for training. As before, 1000 test cases were generated using OracleNet with 100 % success rate. Table 4.2 shows the relevant performance statistics.

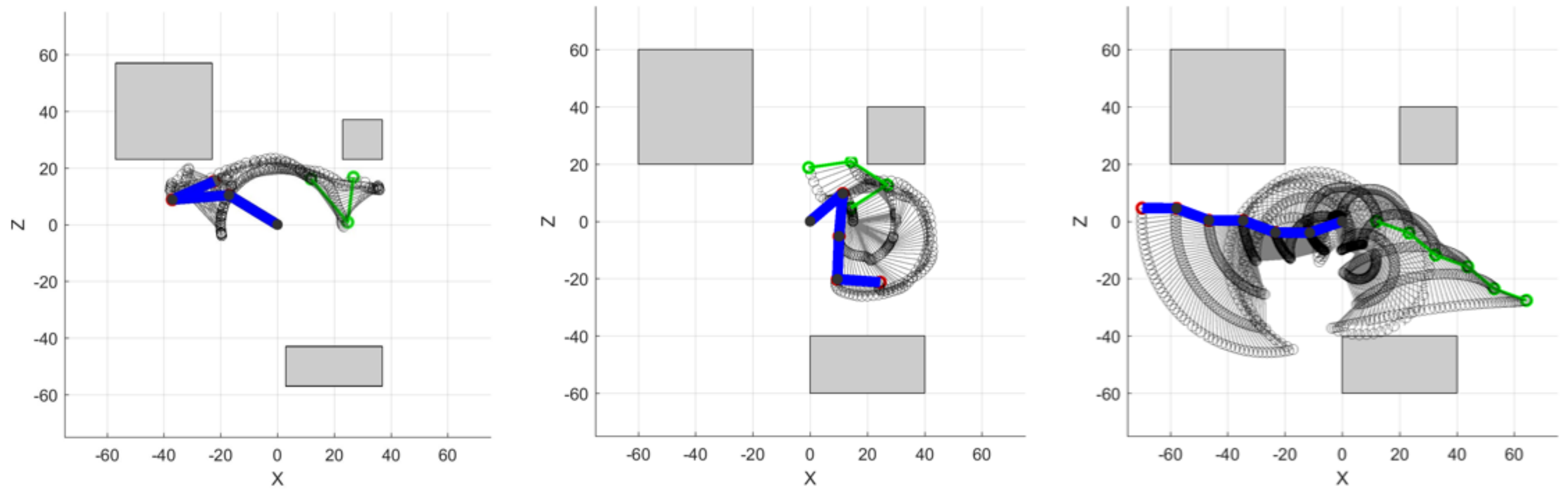


Figure 4.3: Example plots of successful paths generated on 3-link, 4-link, and 6-link robots (from the left) using our algorithm, showing the trail of the links as it makes its way from its initial joint configuration (marked as green).

4.5 Baxter Simulation

As a final demonstration of OracleNet's capabilities, we show its application on a humanoid dual-arm Baxter robot[1]. We perform two tasks (more about how the tasks were selected further into this section) with the proposed network architecture and training procedure on a simulator for the Baxter.

The Baxter robot used for our experiments, shown in Fig. 4.4 [2]. ROS (Robot Operating System) SDK (version: ROS Kinetic Kame) is used to control and program the robot with Baxter RSDK running on Ubuntu 16.04 LTS. ROS is a collection of software frameworks for robot software development, widely used for perception, planning, localization and mapping, and various other algorithms. In this experiment, robot control is achieved by passing in joint angle information at each time-step.

We train the network to output a sequence of 7 DoF joint-angles (for one arm) for the robot to follow from a starting configuration to a goal configuration. Fig. 4.5 shows the setup of the environment for the robot. The robot, after training, is expected to generate a list of feasible configurations (if they exist) connecting given start and goal configurations. Due to the large number of dimensions in the configuration space, it becomes infeasible to discretize it to a grid of reasonable resolution and use graph search algorithms such as A* to generate optimal paths. Instead, we use MoveIt! and leverage Open Motion Planning Library (OMPL) [3] to generate a set of paths between randomized valid configurations using RRT-Connect [26]. RRT-Connect is an efficient sampling-based planner that combines RRT's sampling scheme with a simple greedy heuristic to generate quick single-query paths.

We generate a relatively small dataset with 100,000 paths and train on it using a model

[1]Baxter is a robot manufactured by Rethink Robotics. Although originally envisioned as an industrial robot aimed at small and medium companies, it has gained popularity as a research and teaching tool due its ease of setup and relative low-cost.

[2]Image taken from sdk.rethinkrobotics.com, ©2015 Rethink Robotics.

[3]MoveIt! is an open source motion planning framework that incorporates motion planning, manipulation, 3D perception, kinematics, control and navigation. For planning in particular, it uses several sample-based planners defined in OMPL.

r0.5

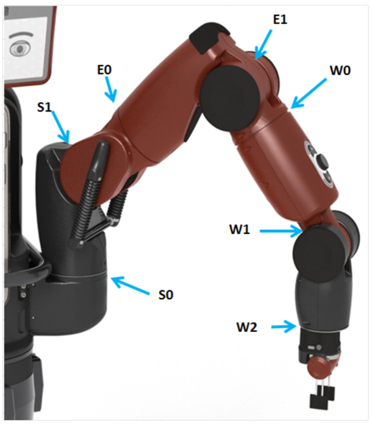


Figure 4.4: A diagram of one of the arms of a Baxter robot with its 7 revolute joints labelled. The robot consists of a torso, 2 DoF head and two 7 DoF arms (shoulder joint: s0, s1; elbow joint: e0, e1; wrist joint: w0, w1, w2), integrated cameras, sonar, torque sensors, and direct programming access via a standard ROS interface.

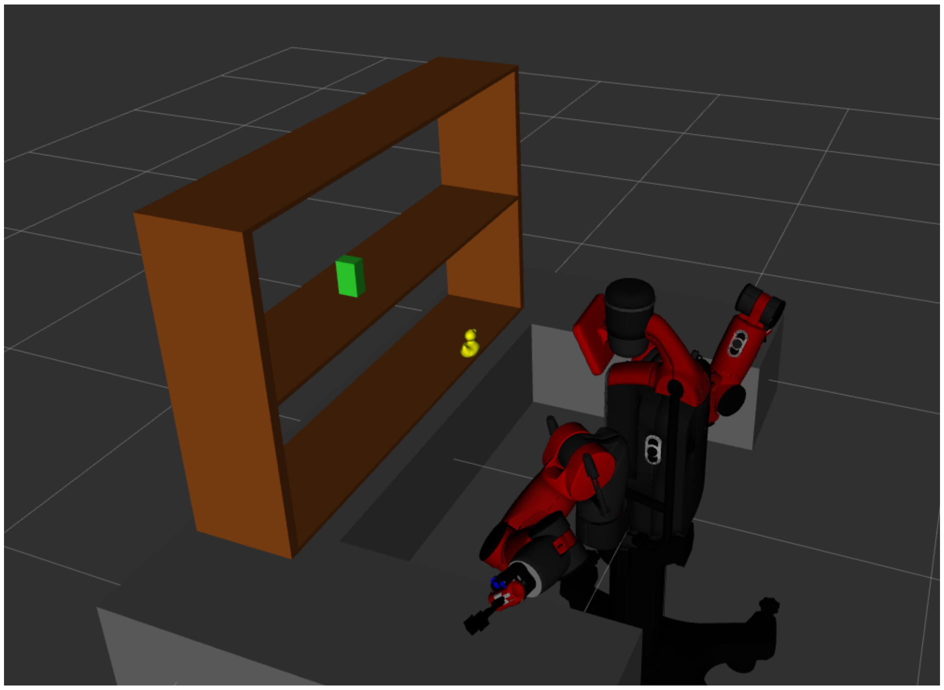


Figure 4.5: The robot is boxed in between three tables with a two-stories shelf in front of it. Two objects are placed on the shelf - a cuboidal object on the upper level and a duck on the lower level.

with a reduced number of layers (2 and 3 layers with 400 units each were experimented with). Due to the significantly constrained work-space in which the Baxter has to operate, connecting arbitrary configurations for testing may not always lead to viable paths. Even RRT-Connect, the "expert planner" used here, frequently failed to generate viable paths before timing-out and since OracleNet by definition tries to emulate its expert's behaviour, its failure in generating paths for certain configurations is only to be expected. We are confident that with a higher quality training dataset, our algorithm will be able to replicate the near-perfect success rates posted for the 6-Link manipulator in the previous section. In all cases, however, OracleNet was able to generate smooth paths connecting the chosen configurations with average and standard deviation of generation times as 0.56 and 0.39 seconds respectively.

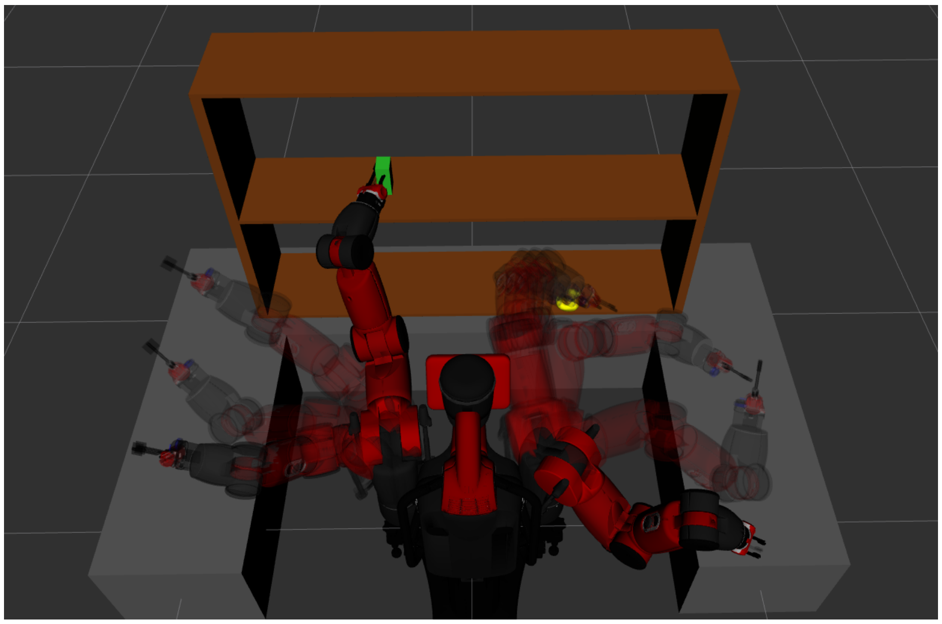
To simulate real-word demands made of planners and manipulation-oriented robots such as the Baxter, we selected two simple tasks for the Baxter to perform with the joint-angle sequences being produced by OracleNet. It is important to note at this point that *task-planning is not the objective here*, and so the start and goal configurations for each leg of the task were manually selected. We also take advantage of the fact that the Baxter's two arms are symmetric. In other words, by changing signs of joint angles appropriately, one can get both arms to act as mirror images of each other with a single 7 DoF configuration. With this in mind, it is entirely feasible to train only on one 7 DoF arm and simultaneously control both arms locally. The caveat here is that each arm has to be restricted to its half of the workspace, since the algorithm will not recognize the other arm as an object in the configuration space and so, will not be able to plan around it while generating paths during testing.

For the first task, we command the robot to move its arms from their initial positions, plan towards configurations where they are able to grab [4] objects on the shelf, and finally return to their configurations with the held objects. See Fig. 4.6 for a composite image of the simulated

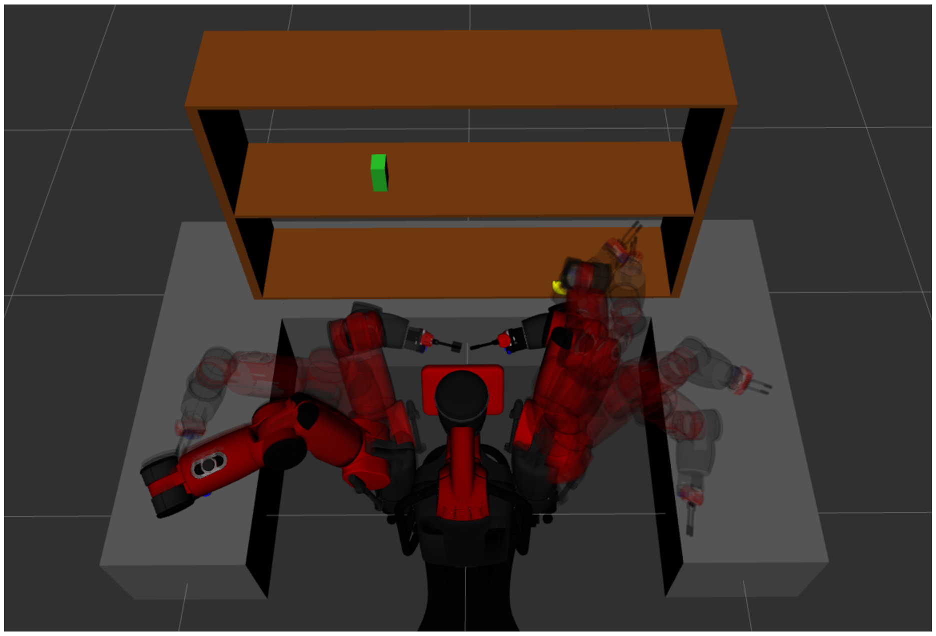
[4]The 7 DoF arm of the Baxter does not include its gripping end-effector in its configuration, as shown in Fig. 4.4. Even though in our results and simulations we show the Baxter "gripping" the objects in question, in practice the objects are merely attached to the end-effector and act as an extra 9^{th} link of the arm. An alternative approach would be to use vacuum (suction) grippers to grab and hold on to objects.

Baxter executing OracleNet generated paths for this task.

For the second task, we demonstrate weak cooperative capabilities of OracleNet with a simple robot hand-over task (refer Fig. 4.6). Once again, we stress here that our proposed algorithm does not solve task planning but merely is a vehicle to generate motion plans extremely fast. The left arm, after being initialized to a position on the left table, is instructed to plan towards the duck on the shelf. After it reaches the desired configuration and grabs the duck, the left arm is instructed to plan towards a predetermined hand-over location where it can safely transfer the object to the right arm without being in danger of colliding with it. Meanwhile, the right arm also simultaneously moves from its initialized position towards the hand-over location to meet the left arm. After the duck has been successfully passed between the two arms, the right arm is then instructed to plan towards to a location on the right arm after which the task is declared to be successfully executed.



(a)



(b)

Figure 4.6: (a) A composite image of the Baxter executing the first task. The trail of the arms show the path taken by them to reach their respective objects. The right arms plans towards the duck while the right arm plans towards the green cuboid. (b) A composite image of the Baxter executing the second task. The right arms plans towards the duck and after reaching a suitable position, grabs it. Meanwhile, the left arm plans towards a predetermined position in the center for the handover. The right arm then plans towards this position with the duck in grasp. Finally, after the handover is successfully executed, the left arm, along with the duck, moves to a position on the left table and the experiment is terminated.

4.6 Scalability and Effects of Path Length

Fig. 6.1 shows the scalability of OracleNet with dimensions. In conjunction with Table 4.1, the figure shows that OracleNet provides minimal improvements to readily accessible algorithms like A* in low dimensional cases with small grid sizes (low resolution). In spaces with dimensions greater than 2, OracleNet scales far better, as A* (paired with a Euclidean distance heuristic) scales exponentially worse with dimensions, and RRT* (depending on the trade-off selected between step size and path optimality), scales polynomial at best. Results presented in Table 4.2 and Fig. 6.1 present the trends that show a modest increase in execution time for OracleNet while path optimality rivals A* and far outstrips RRT*.

Fig. 4.7 shows the spread of generation times of paths when arranged as a function of path length (demonstrated on the 3-link arm). OracleNet has a linear relationship to how far away the query points are located, meaning that stepping through the environment is done consistently with a fixed time. For pathfinding algorithms such as A*, the farther apart the query points are, the more chances are a number of nodes required to be explored increases exponentially (polynomial with an appropriately chosen admissible heuristic). The fixed-time behavior of OracleNet is especially valuable in time-critical scenarios where a feasible path must be known in fixed time.

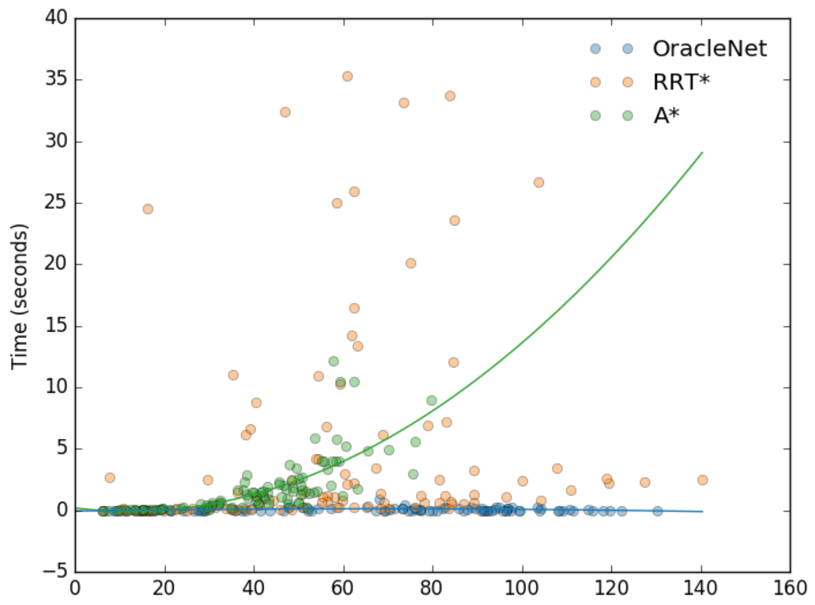


Figure 4.7: OracleNet and A* path generation times plotted as a function of path lengths for the 3-link robot presented here. Note the near fixed-time generation times for OracleNet, irrespective of where the query points are located in the entire c-space.

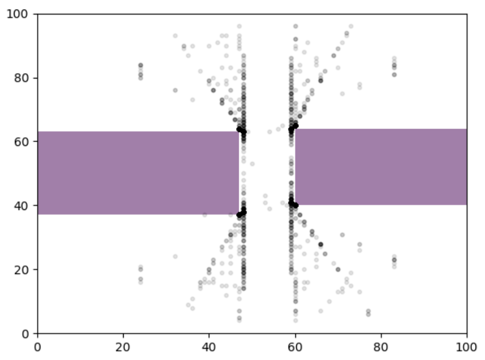
Chapter 5

Efficient Sample Generation

This chapter deals with the discussion of a few additions to the original algorithm presented in Chapter 3. Even though the experiments performed in this chapter seem to have far better results than the original OracleNet as defined in the preceding chapters, due to constraints of time a more thorough investigation could not be performed and therefore, the results presented in Chapter 4 still stand as OracleNet's benchmark.

Recall that the network is trained on a set of dense paths - i.e., paths that have been generated by searching over a graph of nodes. A waypoint in a given path is considered to be *contributing* if after performing the rewiring operation described in 3.5, that waypoint is retained. In a configuration space that is not overly crowded with obstacles, this might mean that several paths will have points that do not contribute to the path. The question then becomes - is it possible for the network to be trained to generate only contributing points in path with the implicit understanding that the straight line connecting two contributing points remains completely in collision-free configuration space?

The network as defined in the 3 takes in the current waypoint in the path and based on the context of the supplied goal, outputs the next waypoint in the path. Under this new idea of generating only contributing points in a path, the trained network should output only a sequence


(a)

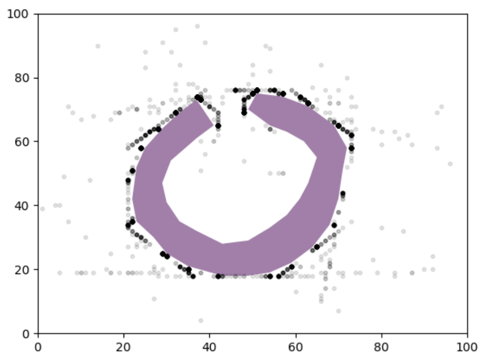

(b)

Figure 5.1: Two of the gridworld environments used in Chapter 4 with only the contributing points in the entire training set plotted. The darker a point is, the more persistently it appears in the training set and the more likely it is being produced as a contributing point in context of a path.

of contributing points, thereby greatly reducing the total number of points generated per path. To do this, the training set has to be processed accordingly. Each path in the training set is ran through the rewiring algorithm and all the non-contributing points are removed. Since we use A* to generate our training set and due to its property of generating optimal paths over the defined graph, certain sections of the graph tend to be visited more frequently than others. This ends up reducing the training set to a relatively small subset of the total nodes that persistently contribute the most in paths connecting arbitrary points. Fig. 5.1 shows an example of this compacted training set for two sample environments.

Since the network now can be trained on this reduced data-set, the network size can also be suitably scaled down. This adds to the already substantial speed-up provided by having to generate only contributing points. Experiments were performed with this scaled down dataset and network on some of the gridworld environments presented in Chapter 4. The results were promising when measured with the metrics defined in the preceding chapter. Fig. 5.3 shows this in action on two 2D Gridworld environments and a 3 dimensional configuration space derived from a 3-Link Manipulator. Note the relative lack of superfluous non-contributing points in the generated path. Since the straight lines connecting each of these points together remains completely in collision-free configuration space, the network becomes free from having to generate dozens of points.

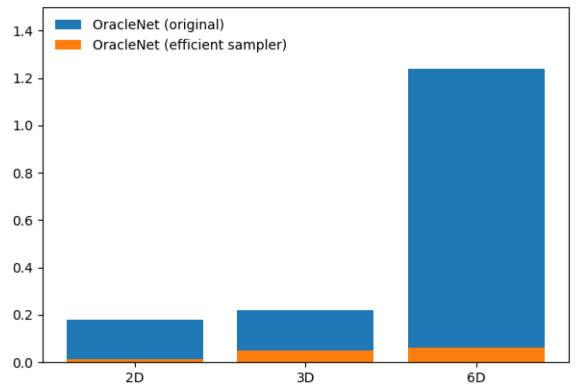
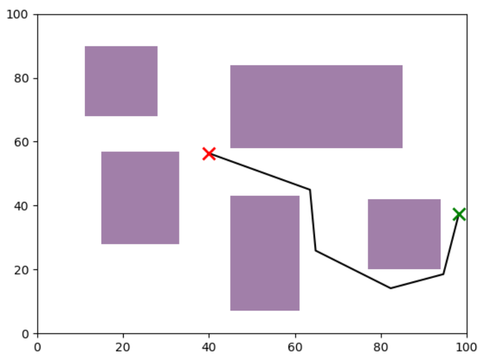
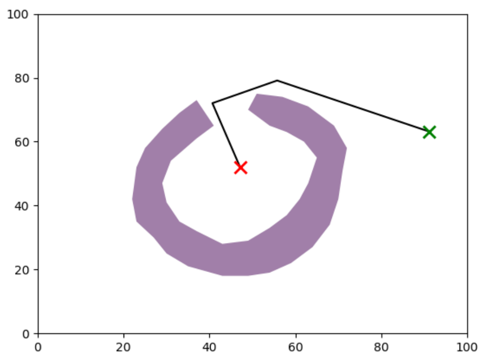


Figure 5.2: A comparison of average path generation times for the original OracleNet and the modified version presented in this chapter. The largest reduction in path generation time comes from the higher dimensional configuration spaces, where we believe we overestimated network sizes in the first place.

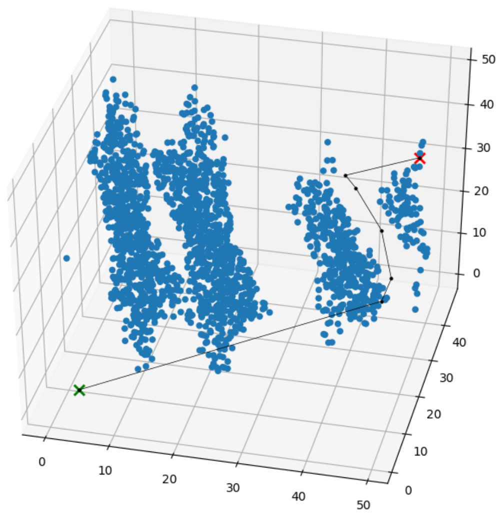
Fig. 5.2 shows an overlay of path generation speeds for the original OracleNet algorithm and the modified version presented in this chapter. The combined effect of smaller networks and fewer forward passes dramatically reduce total path generation times, as evidenced in the graph. This further strengthens our claim of OracleNet being a fixed-time planner which is orders of magnitude more favourably scalable than traditional planners, as the increase in path generation times with increasing configuration space dimensions is shown to be negligible.



(a)



(b)



(c)

Figure 5.3: Sample paths generated by the modified OracleNet path generation network. The effect of going from generating dozens of points to generating less than 5 has a dramatic effect on how many times the network has to perform a forward pass and therefore, speed of path generation.

Chapter 6

Conclusion

The three main qualities required of any successful motion planner are feasibility, speed, and optimality. While feasibility is usually non-negotiable, speed and optimality can often be traded for each other depending on what the priorities are. This work attempts to achieve both high speed as well as optimality while simultaneously preserving feasibility. The experiments performed in the various environments presented cover a range of cases that support this effort.

Using a discrete uniform grid to build the training set allows us to estimate the efficiency of the dataset and the generalizing ability of the network. The fact that it is possible to successfully generate paths between two continuous free floating points when it is trained only on a sparse discrete grid indicates generalizing ability of the network. The 6-link experiment is a good example of this. Even with just 10 uniformly spaced discrete samples per axis, the network managed to find optimal collision-free paths connecting continuous points sampled arbitrarily in the c-space. A second metric used to define training set efficiency is the so-called “c-space compression factor”, F. This factor refers to the ratio of all collision-free paths that can possibly be generated on the grid to the number of paths actually used in the training set. A well-trained network with strong generalizing capabilities will have a high value for this ratio. Table 6.1 shows the F values recorded for the experiments presented in Section III. The 2D experiments conducted

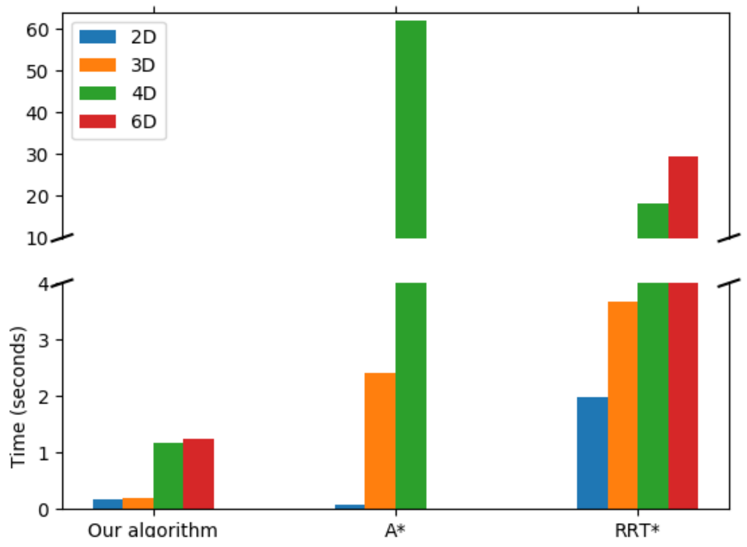


Figure 6.1: A comparison of average path generation times for our algorithm, A*, and RRT* for all the cases presented here. Because of the unreasonably high complexity of running A* in a 6-dimensional c-space with full connectivity, it was chosen not to be included. The scalability for online path generation is far better for OracleNet and produces near-optimal paths through the entire space which cannot be done practically with other competing methods (i.e. saving and referencing a subset of optimal paths in memory)

here are low-dimensional spaces with a higher fraction of obstacle samples, therefore a relatively large dataset is required to represent the obstacles implicitly. On the other hand, the sparsity of high dimensional spaces can be leveraged to get good generalizing performance even with a relatively small dataset. Note that F, on its own, is not sufficient to estimate the generalizing power of the network; it has to be taken in context with the resolution of the grid used to discretize the continuous c-space.

Another effect of network size beyond the required training data directly correlate with path generation times, an increasing network size will increase the path generation time at a comparable rate. While overfitting is avoided (as described in Section III), larger networks produce minimal performance improvements, and it is worth empirically honing it to an appropriate network size. A strategy to reduce network size while retaining performance and potentially improving robustness is using dropout [27].

A unique property of OracleNet is with regards to real-time execution. Unlike any other

Table 6.1: C-space compression factors for the environments presented in Section IV

Environment	C-space Compression Factor, F
Gridworld (2 dimensional c-space)	2500
3-link (3 dimensional c-space)	25000
4-link (4 dimensional c-space)	3,000,000
6-link (6 dimensional c-space)	1,000,000

motion planning strategy (with the exception of the potential field strategies), a movement can be initiated immediately before a path through the environment is found. Because OracleNet encodes optimal behaviors, one can execute OracleNet under the belief that any step it takes will move towards the right direction. Issues such as repair and rewiring can be resolved by considering an N-step horizon, which is a typical strategy in online motion planning and specifically re-planning. Thus, in practice, one can expect that a robot can move and react instantaneously once given its goal state.

The speed and invariance of performance to environments do come at a cost, namely the creation of a dataset, and time and computation power to accurately generate paths. This cost is easily accepted if the algorithm is viewed as an extremely fast way to create online optimal and feasible motion plans for any start and goal state, in any trained environment, given that the process of creating training data and training of the network occurs offline. As mentioned previously, potential real-world applications of this algorithm include warehouse scenarios, robot grasping, and hand-off.

Given a foundational framework for mimicking Oracles using neural networks, several extensions can be pursued. Future work can involve adapting to dynamic environments, unseen environments via transfer learning, improving sampling strategies for Oracles (i.e. training data selection), and combinations of these methods for task-based planners.

Appendix A

Performance trends with variable dataset and network sizes

To analyze the learning capabilities of OracleNet, we performed experiments on environment 'Difficult 2' (refer to Fig. 4.1) with variable dataset and network sizes. Success rates and path optimality (benchmarked against A*) are used to measure performance trends. For obvious reasons, it is not possible to compare all dataset sizes to all network sizes, therefore the trends presented here are meant to serve as an approximate guide to selecting reasonable datasets and network configurations. Note that all trends presented here do not involve the use of the repair and rewire modules, since the purpose is to (1) study the role of dataset and network sizes in generating successful paths, and (2) to show that with sufficient data and an appropriately chosen well-trained network, OracleNet is able to generate near-optimal feasible paths on its own and repair is needed only to handle the edge cases while rewire takes it closer to being absolutely optimal. Thus, the paths generated for the analysis presented here are directly generated by the network and do not involve any correction/processing.

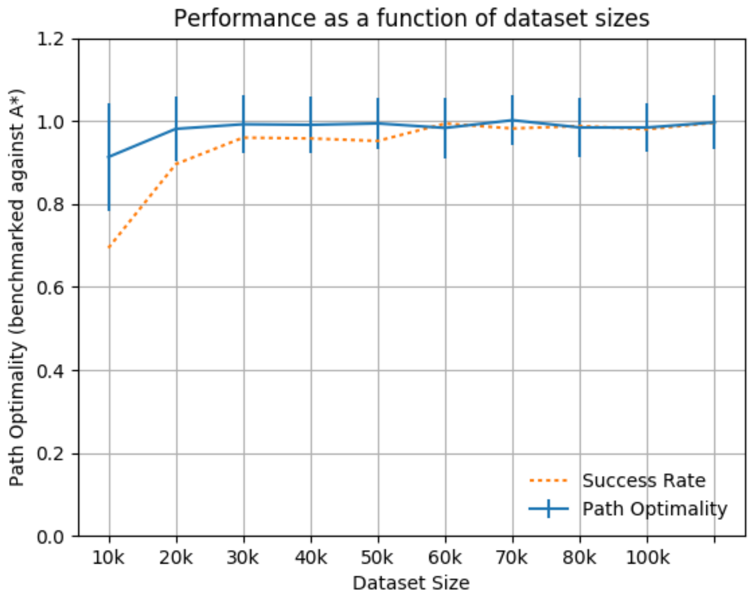


Figure A.1

A.1 Variable Dataset Sizes

The network size is kept fixed here (refer to the network architecture described in 4) and the dataset size is incrementally increased to a maximum of 100,000 valid A* generated paths. It is to be expected that the network will predict paths more accurately when a larger training set is provided. Note that success rates reach greater than 90% with 20,000 training paths and continue on an upward trend till it reaches 99% when 60,000 paths are considered for training. Path optimality for OracleNet climbs quickly to reach A*'s level of optimality and does not seem to be as dependent on dataset size as success rate (only the optimality statistics of successful paths are considered).

From the graph, it can be inferred that selecting a dataset size between the two aforementioned values to get a good raw success rate while relying on repair/rewire to handle the remaining 3 % of the cases is the reasonable strategy to choose. Also, the environment chosen here has a large number of arbitrarily placed obstacles leading to a high degree of non-linearity in path shapes, hence a larger than average dataset may be required.

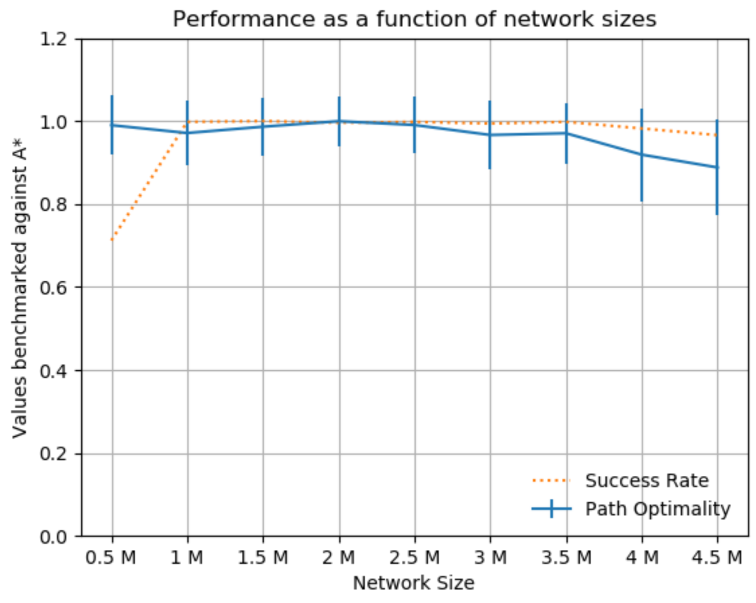


Figure A.2

A.2 Variable Network Sizes

The dataset size is kept fixed here at 100,000 valid paths while the network size (number of parameters, with each increment indicating an additional layer with 256 units) is kept variable. Note that the network configuration used in the paper and in the previous section has 2 million (2 M) parameters. It has already been established in the previous section that using 100,000 paths with 2 M parameters leads to $> 99\%$ success rate, which is corroborated here as well. It is interesting to note the falling performance metrics when network size grows beyond a certain range. This can be attributed to the network having too many trainable weights as compared to training data which causes it to overfit quickly and perform poorly with unseen test sets.

Bibliography

[1] Tianhao Zhang, Gregory Kahn, Sergey Levine, and Pieter Abbeel. Learning deep control policies for autonomous aerial vehicles with mpc-guided policy search. In *Robotics and Automation (ICRA), 2016 IEEE International Conference on*, pages 528–535. IEEE, 2016.

[2] Harold Scott Macdonald Coxeter. *Non-euclidean geometry*. Cambridge University Press, 1998.

[3] Howie M Choset, Seth Hutchinson, Kevin M Lynch, George Kantor, Wolfram Burgard, Lydia E Kavraki, and Sebastian Thrun. *Principles of robot motion: theory, algorithms, and implementation*. MIT press, 2005.

[4] Brian Paden, Michal Čáp, Sze Zheng Yong, Dmitry Yershov, and Emilio Frazzoli. A survey of motion planning and control techniques for self-driving urban vehicles. *IEEE Transactions on Intelligent Vehicles*, 1(1):33–55, 2016.

[5] Oussama Khatib. Real-time obstacle avoidance for manipulators and mobile robots. In *Autonomous robot vehicles*, pages 396–404. Springer, 1986.

[6] Franco P Preparata and Michael I Shamos. *Computational geometry: an introduction*. Springer Science & Business Media, 2012.

[7] S. S. Ge and Y. J. Cui. Dynamic motion planning for mobile robots using potential field method. *Auton. Robots*, 13(3):207–222, November 2002. ISSN 0929-5593. doi: 10.1023/A:1020564024509. URL `https://doi.org/10.1023/A:1020564024509`.

[8] Stephen R Lindemann and Steven M LaValle. Current issues in sampling-based motion planning. In *Robotics Research. The Eleventh International Symposium*, pages 36–54. Springer, 2005.

[9] Jérôme Barraquand, Lydia Kavraki, Jean-Claude Latombe, Rajeev Motwani, Tsai-Yen Li, and Prabhakar Raghavan. A random sampling scheme for path planning. *The International Journal of Robotics Research*, 16(6):759–774, 1997.

[10] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4 (2):100–107, 1968.

[11] Steven M LaValle. Rapidly-exploring random trees: A new tool for path planning. 1998.

[12] Nathan Ratliff, Matt Zucker, J Andrew Bagnell, and Siddhartha Srinivasa. Chomp: Gradient optimization techniques for efficient motion planning. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pages 489–494. IEEE, 2009.

[13] David González, Joshué Pérez, Vicente Milanés, and Fawzi Nashashibi. A review of motion planning techniques for automated vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 17(4):1135–1145, 2016.

[14] Roy Glasius, Andrzej Komoda, and Stan CAM Gielen. Neural network dynamics for path planning and obstacle avoidance. *Neural Networks*, 8(1):125–133, 1995.

[15] Nutan Chen, Maximilian Karl, and Patrick van der Smagt. Dynamic movement primitives in latent space of time-dependent variational autoencoders. In *Humanoid Robots (Humanoids), 2016 IEEE-RAS 16th International Conference on*, pages 629–636. IEEE, 2016.

[16] Aviv Tamar, Yi Wu, Garrett Thomas, Sergey Levine, and Pieter Abbeel. Value iteration networks. In *Advances in Neural Information Processing Systems*, pages 2154–2162, 2016.

[17] Pieter Abbeel, Adam Coates, and Andrew Y Ng. Autonomous helicopter aerobatics through apprenticeship learning. *The International Journal of Robotics Research*, 29(13):1608–1639, 2010.

[18] Sylvain Calinon, Florent D'halluin, Eric L Sauser, Darwin G Caldwell, and Aude G Billard. Learning and reproduction of gestures by imitation. *IEEE Robotics & Automation Magazine*, 17(2):44–54, 2010.

[19] Jaeyong Sung, Seok Hyun Jin, and Ashutosh Saxena. Robobarista: Object part based transfer of manipulation trajectories from crowd-sourcing in 3d pointclouds. In *Robotics Research*, pages 701–720. Springer, 2018.

[20] Elmer G Gilbert, Daniel W Johnson, and S Sathiya Keerthi. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal on Robotics and Automation*, 4(2):193–203, 1988.

[21] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[22] Ken-ichi Funahashi and Yuichi Nakamura. Approximation of dynamical systems by continuous time recurrent neural networks. *Neural networks*, 6(6):801–806, 1993.

[23] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[24] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, and Matthieu Devin. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[25] François Chollet. Keras. `https://github.com/fchollet/keras`, 2015.

[26] James J Kuffner and Steven M LaValle. Rrt-connect: An efficient approach to single-query path planning. In *Robotics and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on*, volume 2, pages 995–1001. IEEE, 2000.

[27] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.