UNIVERSITY OF CALIFORNIA SAN DIEGO

Towards End-To-End Learning-Based Algorithms in Motion Planning

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

in

Computer Science

by

Yinglong Miao

Committee in charge:

Professor Michael C. Yip, Chair
Professor Henrik I. Christensen, Co-Chair
Professor Hao Su

2020

The thesis of Yinglong Miao is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Co-Chair

Chair

University of California San Diego

2020

DEDICATION

To my family, friends and mentors, for their patience, support, and love.

EPIGRAPH

What I cannot create, I do not understand.

Richard Feynman

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to acknowledge my advisor, Professor Michael Yip, for his support and guidance for my research projects during my Master's study. I am very fortunate to be able to participate in the Advanced Robotics and Controls Lab (ARCLab). My two years have been fulfilling thanks to this opportunity.

I would like to acknowledge my Ph.D. mentor, Ahmed Qureshi, for his patience and guidance for my projects. It has been helpful to work with someone with similar areas of interest. Meanwhile, I'd like to acknowledge my colleagues in the lab and professors I have met in classes, for their help and insightful discussions throughout the journey.

Finally, I'd like to acknowledge my parents, home-stay family Florita and Roy, and friends for their patience, support, and love. My two years of Master's study would not be as colorful without them.

Section 3.2 and Section 3.3 in Chapter 3, and Section 4.2 in Chapter 4, in full, are a reprint of the material with minor modifications as it appears in Motion Planning Networks: Bridging the Gap Between Learning-Based and Classical Motion Planners 2020. Qureshi, Ahmed H.; Miao, Yinglong; Simeonov, Anthony; Yip, Michael C., IEEE Transactions on Robotics, 2020. The thesis author was the co-author of the published paper.

Section 3.4 in Chapter 3, and Section 4.3 in Chapter 4, in full is currently being prepared for submission for publication of the material. Authors include Li, Linjun; Miao, Yinglong; Qureshi, Ahmed H.; Yip, Michael C.. The thesis author was the shared first author of this material.

VITA

2017	Junior Research Assistant, the Chinese University of Hong Kong
2018	B.Sc. in Computer Science, the Chinese University of Hong Kong
2019	Teaching Assistant, Department of Computer Science and Engineering University of California, San Diego
2020	M.Sc. in Computer Science, University of California, San Diego

PUBLICATIONS

"Motion Planning Networks: Bridging the Gap Between Learning-based and Classical Motion Planners" *IEEE Transactions on Robotics* (2020).

ABSTRACT OF THE THESIS

Towards End-To-End Learning-Based Algorithms in Motion Planning

by

Yinglong Miao

Master of Science in Computer Science

University of California San Diego, 2020

Professor Michael C. Yip, Chair
Professor Henrik I. Christensen, Co-Chair

Motion planning is one of the most critical tasks in robotics, as it is one of the few critical functions for robot autonomy. This component requires fast computations and generalization to different environments for problems such as collision avoidance. Deep learning, as a new fast-growing field, offers great advances in computational speed and generalization. It has shown success in computer vision and reinforcement learning, which is closely related to motion planning. In this thesis, we will investigate the combination of learning and motion planning methods. Specifically, we separately consider individual components of motion planning tasks. By combining the proposed learning-based methods for each component, we can obtain an

integrated end-to-end learning-based motion planning algorithm. We show experimental results for each component. In general, our learning-based methods showed high computational speed with generalization in several motion planning tasks.

Chapter 1

Introduction

1.1 Background and Motivation

1.1.1 Motion Planning

Planning refers to the problem of finding a sequence of actions to achieve certain goals. It is a fundamental problem in computer science and a general problem in everyday life. Motion planning is one planning problem that aims to find a sequence of motion controls for the robot or a general agent to achieve specific goals. This field has been studied for more than five decades in literature, with one well-known example problem, the Piano Mover's problem [92].

Motion planning is also used in other areas, such as computer graphics and computational biology, where motions are taken into consideration [56]. For example, in computer graphics, objects need to follow specific motion dynamics. These tasks require some level of motion planning[56]. In terms of algorithms, motion planning field combines methods from combinatorial search (e.g., greedy method, graph theory), optimization (e.g., trajectory optimization), control theory (e.g., stability analysis), geometry (e.g., analysis of the shape of the robot and obstacles, geometric control) and machine learning (e.g., reinforcement learning, imitation learning). Thus motion planning is a meaningful research direction with practical applications and can benefit many other areas, while itself can take advantage of novel research from different domains.

In research, as a result of a variety of constraints, motion planning in robotics includes

several different subfields. These include, but are not limited to, planning under partial observations and planning with dynamics constraints. Planning under partial observations is a general problem to apply motion planning to real-time execution of robots in real environments. It requires local planning and updating global information. For example, the famous SLAM algorithm is one way to update the global information from local perceptions[25]. Planning with dynamics constraints is important in underactuated robotics, where the controls connecting robot states cannot be directly obtained [35]. For example, cars driving at high speed and satellites often fall into this scenario [35]. The dynamics constraints in this problem need to be explicitly considered during planning. This problem is generally termed kinodynamic motion planning in the motion planning community, which we will work on later in this thesis.

In terms of algorithms, complete motion planning algorithms, including search-based methods and sampling-based methods, are often uninformed. However, as they can cover the entire search space asymptotically, correctness can be guaranteed. In this thesis, we are more interested in making motion planning more intelligent and efficient. Instead of using classical approaches with hand-crafted heuristics, we are interested in applying artificial intelligence to this problem. Specifically, given the success of deep learning in robotic applications such as autonomous driving, deep learning is a promising method to improve motion planning, especially in problems where domain knowledge is hard to obtain. In the big picture, this direction might be directly related to the advancement in automation of robotics and the realization of scalable artificial intelligence.

We have identified two potential components in a motion planning pipeline that can benefit from deep learning: collision checking and path planning. Collision checking is usually the computational bottleneck in the motion planning pipeline [60]. The algorithmic structure of it indicates deep learning can be applied. Meanwhile, in path planning, several past works have studied how to make planning more informed. Given the success of deep learning in reinforcement learning, it can be one way to guide path planning heuristically.

1.1.2 Deep Learning

The field of deep learning has been studied since the 1950s [83]. It uses models such as neural networks to extract information from past experiences. During extraction, only useful information is kept by statistical analysis. Hence the extracted information is embedded in space with a much lower dimension, similar to compressing. Later, when query, the embedded information can be used to find a similar experience in the past. Since the embedded space is highly complex, it can mimic abstraction during human reasoning. Due to this advantage, deep learning can often achieve human-level reasoning, even under variations in the problem.

However, deep learning has shown practical values since very recently, because of the advances of computational power. Since then, it has demonstrated success in areas such as computer vision [36], natural language processing [22], and reinforcement learning [93], due to the power of pattern recognition.

The advantages of deep learning can be summarized as low computational cost and high accuracy, with the ability to generalize to different inputs. It can extract features that are better than human knowledge. These advantages are promising for motion planning, as the eventual goal of motion planning is to generate real-time planning and execution for robots. Specifically, the computational speed and the power of pattern recognition of deep learning can help find sub-optimal paths faster than classical methods. Meanwhile, deep learning's generalizability can help with recognizing different inputs, such as different planning scenes.

1.1.3 Deep Learning in Motion Planning

There exist past works that have shown the success of deep learning in motion planning and related fields. Examples are the applications of reinforcement learning and imitation learning in optimal control, navigation and manipulation tasks [71][79]

Reinforcement learning algorithms focus on learning a good policy that produces optimal actions in each state. On the other hand, imitation learning mimics expert policy. If the expert

policy is optimal, then imitation learning also achieves the same goal as reinforcement learning.

These learning algorithms often directly work on the joint distribution of state and control. In contrast, motion planning algorithms work on the state distribution, and only implicitly obtains the control sequences. This difference brings up our intuition of learning the state distribution. Learning only the state distribution can potentially reduce the amount of information to be captured by the neural network. Meanwhile, the connection between states can be achieved by steering functions, such as boundary value problem (BVP) solvers.

Another significant difference between learning methods and motion planning algorithms is that learning approaches do not have completeness guarantees as motion planning algorithms. Unlike motion planning algorithms, the safety of the learned policy cannot be guaranteed during online execution.

1.1.4 Summary

In this chapter, we have previously shown the importance of research in the motion planning field and how it can be benefited from deep learning. We have also identified two possible components where deep learning can be applied. In this thesis, we will show our work on a learning-based collision checker and a motion planner. Meanwhile, we also investigate life-long learning, which relates to real-life motion planning applications. In the end, we extend the learning-based motion planner to the kinodynamic motion planning setting, where the control sequence is required.

1.2 Problem Definition

This section will give the formal problem definition of motion planning, collision checking, and deep learning.

1.2.1 Motion Planning

The problem formulation of our motion planning problem is aligned with the formulation of the Markov decision process (MDP) [26], as the physical motion model satisfies Markov assumption [6]. Unlike MDP, the observable Markov decision process (POMDP) includes an additional observational model, as the search space is not fully observable [3]. Although we do not consider the observation model in this thesis, we want to give a complete picture of the general motion planning problem. Hence, we use the more general POMDP formulation to introduce the general problem.

The input to a general motion planning problem contains x_0 (start state), x_G (goal state) and W (planning scene, or world). The output is a trajectory $x(\cdot)$ and $u(\cdot)$ indexed by time t. The world W is usually not fully-observable, and in which case, it is represented by the observation sequence $z(\cdot)$. The mathematical formulation of this as an optimization problem is as follows:

$$
\begin{aligned}
\min_{x,u} \quad & J(x,u;W) \\
\text{s.t.} \quad & \dot{x}(t) = f_w(x(t),u(t),t), \forall t \text{ (motion model)} \\
& z(t) = f_z(x(t),t,W), \forall t \text{ (observation model)} \\
& f_x(x(t),t) \leq V_x(t), \forall t \text{ (task-specific constraints, such as collision free)} \\
& x(t) \in \mathscr{X}, u(t) \in \mathscr{U} \text{ (state boundary and control boundary)} \\
& x(0) = x_0 \text{ (start constraints)} \\
& d(x(T),x_G) \leq V_G \text{ (goal constraints)}
\end{aligned}
\tag{1.1}
$$

Here J denotes a general cost function that can be defined by the specific problems. f_w and f_z denote the general motion model and observation model, respectively. f_x specifies a general function that represents metrics in the state space at each time. For instance, one implementation of this can be the shortest distance to obstacles. V_x specifies the valid criteria for the state metrics. $\mathscr{X}$ and $\mathscr{U}$ specify the feasible state space and control space, respectively. V_G determines the size

of the goal region, and d denotes a general distance function between two states. Specifically, in Euclidean space, d is the Euclidean distance.

Implementations of the functions stated above create different variants of motion planning problems. These variants include geometric motion planning, kinodynamic motion planning, moving objects, partial observation, manipulation, and locomotion, to name a few. In this thesis, we will focus on geometric motion planning and kinodynamic motion planning with complete knowledge of the environment. We will give a detailed description of these two specific tasks later in this section.

In computer science, problems are classified based on if efficient algorithms exist to solve them. In this notion, realistic motion planning problems are mostly NP-hard. As a well-known problem instance, the Piano Mover problem is considered to be PSPACE-hard [80]. However, by discretization or probabilistic approaches, the problems can still be solved efficiently in practice. These methods will be discussed in detail in chapter 2.

The evaluation criteria of motion planning algorithms include **feasibility** (Correctness and Completeness), **optimality**, and **efficiency** (time efficiency and memory efficiency), according to the general evaluation criteria in algorithm design [20][56]. In most motion planning algorithms, these notions are either resolution-wise or asymptotically probabilistic. Evaluations of the above notions can be obtained for many motion planning algorithms such as A* and RRT*. The theoretical performance often studies the convergence rate of a specific notion of evaluation. In contrast, experimental evaluation gives actual performance obtained from experiments.

The above defines motion planning problems in continuous space. A discretized version is usually used in real-life applications and actual implementations. Below is a discretized

definition of the problem to make our discussion more complete.

$$
\begin{aligned}
\min_{x,u} \quad & J(x,u;W) \\
\text{s.t.} \quad & x[t+1]-x[t]=f_w(x[t],u[t],t),\forall t \text{ (motion model)} \\
& z[t]=f_z(x[t],t,W),\forall t \text{ (observation model)} \\
& f_x(x[t],t)\leq V_x(t),\forall t \text{ (task-specific constraints, such as collision free)} \\
& x[t]\in\mathscr{X},u[t]\in\mathscr{U} \text{ (state boundary and control boundary)} \\
& x[0]=x_0 \text{ (start constraints)} \\
& d(x[T],x_G)\leq V_G \text{ (goal constraints)}
\end{aligned}
\tag{1.2}
$$

Here the motion model can have different implementations based on the integration method. Meanwhile, we only consider the discretized time step in this formulation. However, in actual definition, the mapping from the discretized time step to the time needs to be considered.

Geometric Motion Planning

In geometric motion planning, the differential motion model is not needed. Instead, planning needs to make sure the state trajectory is continuous. Controls are not considered in this problem. The discrete version usually linearly interpolates states by a fixed short distance. Thus the planning needs to make sure the discretized trajectory is collision-free w.r.t. to the specific resolution defined by the fixed distance.

Kinodynamic Motion Planning

Kinodynamic motion planning explicitly considers the motion model. Hence controls also need to be determined during planning. Similar to geometric motion planning, the discrete version also checks constraints with a particular resolution. Real-time execution using the planned state and control trajectories is often affected by noise. Thus, robustness also needs to be considered for a complete kinodynamic motion planning algorithm targeting real-life

applications.

1.2.2 Collision Checking

The input of a collision checker contains O, denoting a representation of the obstacles, which can be partial or compact, and x denoting the test state for collision query. In robotic applications, both robots and objects have shapes. Hence collision checkers can also take input O_1 and O_2 denoting two shapes and corresponding locations.

The output of a collision checker has several different specifications, often depending on the problem requirements. Several possible queries are:

- in collision or not
- probability of collision
- collision region
- distance of two inputs (point-set distance or set-set distance)

Based on the input type, the problem has several variants, including deforming objects, moving objects, and partial representation, to name a few[103]. The performance of a collision checker includes **correctness** and **efficiency** (time and memory).

1.2.3 Deep Learning

Deep learning is often classified into supervised learning and unsupervised learning[31]. In this thesis, we will focus on supervised learning. The input of a supervised learning algorithm contains a training set D_{train}, a validation set D_{val} and a test set D_{test}, where each dataset is in the form of

$$D = \{(x_i, y_i) : x_i \sim P_x(\cdot)\}.$$

Here P_x denotes some probability distribution of x. The deep learning algorithm tries to learn a conditional distribution $P_{y|x}$ using only the training set to minimize the loss $\sum_i L(\hat{y}_i, y_i)$ on test

set, with some predefined loss function L. Here $\{\hat{y}_i\}$ are the outputs of the algorithm using the test set. They can be considered as samples according to the learned conditional distribution $P_{y|x}(\cdot;\theta)$, where θ denotes the neural network parameters. Generally, the learned conditional distribution can also be represented as a functional mapping $f_\theta(x)$.

The evaluation of a deep learning algorithm usually considers its **generalization** and **sample efficiency**. Generalization measures how well the algorithm performs, given unseen data by measuring losses on training data and test data. Sample efficiency is termed as trainability and expressivity of neural networks in a recent paper [46]. It measures the size of the neural networks as a function of the dataset's size to achieve optimality in learning. A detailed discussion of deep learning can refer to [31]. In this thesis, we will also explore continual learning, which is a sub-field of deep learning.

Continual Learning

Traditional supervised learning tries to solve optimization problem given a collection of identically and independently distributed (i.i.d.) data $D = \{(x_i, y_i)\}_{i=1}^N$, w.r.t. some loss function defined on each sample, as we defined previously. However, real-world problems may break the i.i.d. assumption, where data may be ordered and locally correlated.

As defined in [63], in this problem, each sample is a tuple (x_i, t_i, y_i). where $t_i \in \mathscr{T}$ is a task descriptor, $x_i \in \mathscr{X}_{t_i}$ is the feature vector, and $y_i \in \mathscr{Y}_{t_i}$ is the target vector. The feature vector space and target vector space are parameterized by task t_i. The sample (x_i, t_i, y_i) may not follow i.i.d. distribution. For simplicity, in this thesis the data is assumed to be locally i.i.d., i.e. $(x_i, y_i) \overset{\text{i.i.d.}}{\sim} P_{x_i,y_i|t_i}(\cdot,\cdot)$. Additionally, the data of the same task is assumed to come together.

With this definition, it is interesting to analyze the effect of current tasks on previous or future tasks. [63] obtains an evaluation metrics regarding this. The performance decrease in past tasks is called catastrophic forgetting in the community. In contrast, the performance gain on future tasks can be referred to as transfer learning. The objective is to minimize the loss on the current task while minimizing the catastrophic forgetting effect and maximizing the transfer

learning effect.

1.3 Challenges

1.3.1 Motion Planning

In this thesis, the complication of geometric motion planning is the difficulty of obtaining a theoretical guarantee for heuristic methods' completeness, especially the learning-based approach. Kinodynamic motion planning also adds complexities. It requires explicit consideration of the motion dynamics, which is proved to be NP-hard [44]. Meanwhile, the state space geometry is non-trivial, and thus the distance function is more complicated than simple Euclidean distance.

1.3.2 Collision Checking

The challenge in collision checking is the difficulty to represent shapes and obtain reasoning based on them. Realistic objects often have complicated shapes, and direct analysis of them is both time-consuming and memory-inefficient. Thus a proper representation of the shape needs to be obtained to allow efficient processing without hurting the performance.

1.3.3 Learning-Based Motion Planning

Trajectories in motion planning are high dimensional, and thus learning the entire distribution is not a trivial problem.

1.4 Contribution

In this thesis, we have the following contributions:

- We constructed an efficient approximate collision checker using neural networks, which also achieves high accuracy on our dataset.

- We constructed a sequential predictive model for geometric motion planning that result in an informed and efficient algorithm with a high success rate and sub-optimality.
- We extended this motion planning method to the continual learning task where data comes in a streamline, thus achieving life-long learning.
- We constructed a sequential waypoint prediction model for kinodynamic motion planning, with two planning methods: line search and tree search. We showed improvement in computational time and sub-optimality of the algorithm in toy control problems.

1.5 Thesis Roadmap

In Chapter 2, we provide a literature review for the past works. After that, in Section 3.1, Section 3.2, Section 3.3 and Section 3.4 of Chapter 3, we provide detailed descriptions of our implementations. In Chapter 4, we provide the experimental results for our implemented methods. Chapter 5 concludes this thesis, and talks about future directions to extend this work.

Chapter 2

Literature Review

2.1 Motion Planning

2.1.1 Collision Checking

Collision checking is one crucial component in the motion planning pipeline. In motion planning, this is often treated as one black-box. Recent advances in deep learning might help with the efficiency of this component. In this section, we will review the classical approaches and several learning-based methods for collision checking.

Classical Approach

The research of collision checker has been around for more than three decades [103]. In actual implementation, there exist many libraries such as PhysX, Bullet, and V-Rep. Generally, collision checking algorithms follows three stages: preprocessing, the global phase, and the narrow phase [103].

The preprocessing phase processes the input objects into a hierarchical data structure. The data structure is often implemented by Hierarchical Boundary Volume (HBV) and is later used in other phases for efficient collision checking. The global phase uses the obtained data structure to rule out collision-free pairs of volumes. Several algorithms for this phase include brutal-force [105], spatial partitioning [109] and topological methods [19]. The narrow phase conducts collision checking for basic shapes represented by a set of geometric primitives. It

includes two sub-phases: filtering of collision-free geometric primitives, and exact detection of a small set of geometric primitives.

Collision checking of geometric primitives is a fundamental problem in computational geometry [65]. Based on the representation of geometric primitives, these include polygonal intersection checking [17] and non-polygonal intersection checking. Non-polygonal intersection checking algorithms include collision checking with parametric surfaces, implicit surfaces, CSG model, and point cloud [60].

Point cloud collision detection has recently emerged as point clouds are easy to obtain in reality. In this thesis, we are also using point cloud as inputs to collision checking algorithms. There are, however, very few works on collision checking with point clouds. Most algorithms in this line use a hierarchical data structure of point clouds to represent the shape. For local collision checking, they estimate surfaces using the point cloud, and then achieve collision checking using the estimated surfaces.

Specifically, in the earliest work in this direction [51], HBV is used to represent the object, where each node stores a sample of points of the point cloud with a sphere covering them. For local surface estimation, it uses weighted least square to obtain the implicit function $f(x)$ representing the distance from a point to the surface [1]. Then the collision is determined by checking for some sampled test points $\{p_i\}$ if there is some point p_i that satisfies $f_A(p_i) = f_B(p_i) = 0$. Here, f_A and f_B denote functions of distance to point clouds A and B, respectively.

The work achieved low memory consumption and efficient collision checking, with an average of $0.5 - 2.5$ms. However, preprocessing takes a noticeable amount of time. For 50000 points, it takes around 16s, and for 200000 points, it takes around 120s to build the hierarchical data structure, with a linear increase of time w.r.t. number of points [51].

Later work in [52] improves these results by using the proximity graph to approximate geodesic distance rather than Euclidean distance to compute the distance $f(x)$. Meanwhile, instead of blindly sampling test points, it locates test points directly from the surface's information. As a result, it achieves a time complexity of $O(\log\log N)$ for finding collision points with a

constant number of test samples.

Another work [27] uses the voxel representation of the scene to divide point cloud samples. Each voxel is later divided into a tree hierarchy of groups of points using the Axis-Aligned Bounding Box method. It tests the algorithm on a navigation task where the robot contains 3617 points, and the scene includes 587923 points. After a preprocessing of the raw point cloud, it achieves 25 minutes at the highest speed.

Learning-Based Collision Checking

[69] uses a machine-learning algorithm SVM to estimate the surface implicitly, and check collisions at the leaf level. BVH is also used for efficient collision detection. For one collision checking query, it takes around 500-1000 ms to obtain results for 10000 points. Meanwhile, the performance matches the collision checker on triangle mesh in navigation and manipulation tasks.

A more recent work [21] does not explicitly model the shape of the objects, thus bypassing the huge preprocessing overhead. Instead, it imitates proximal high-precision collision detectors used in motion planning algorithms. It achieves this by using a machine-learning method to learn a function $f(x)$, which indicates if the input in configuration space is in collision. Hence this method can be directly combined with motion planning algorithms for speedup.

2.1.2 Motion Planning

Here we give a literature review of motion planning. An informative textbook of motion planning can be found in [56]. Motion planning is a hard problem in terms of time complexity. For instance, the well-known Piano Mover's problem is proved to be PSPACE-hard [80]. However, in reality, practical motion planning problems have several relaxations, so they usually admit optimal solutions that can be found quickly. Motion planning algorithms can be generally classified into search-based motion planning and sampling-based motion planning [56], which will be introduced in detail later.

2.1.3 Heuristics in Motion Planning

Most research works in motion planning try to accelerate the algorithm by applying heuristics. Most algorithms extract heuristic information from the topological and geometric properties of the search space. Some examples are bidirectional search and search space restriction in Informed-RRT* [77][29].

For heuristic distance, [85] introduces two criteria of heuristic distance function in A* algorithm: **admissibility** and **consistency**. Admissibility is obtained if the heuristic distance is shorter than the actual distance. At the same time, consistency requires a much stronger property of triangle inequality. Specifically, consistency requires the difference of heuristic function at two points should be shorter than the actual distance between those two points [85]. The time improvement of using heuristic methods is theoretically showed in work such as A* and Informed RRT* [85][29].

Search-Based Motion Planning

Search-based motion planning is usually achieved by discretizing the search space and applying graph-search methods, such as BFS, DFS, A* [56][85]. Due to discretization, search-based motion planning can only achieve resolution completeness [56]. The size of the discretized graph usually determines the complexity. In the worst case, traversing the entire graph will take linear time w.r.t. the number of edges in the graph [56]. After a solution is obtained, the resolution can be refined to obtain a more optimal solution.

In research, the Search-based Planning Laboratory at CMU has done leading work in search-based motion planning algorithms. Some examples of their work include [12][41].

Sampling-Based Motion Planning

Unlike the search-based method, sampling-based methods do not discretize the space, but instead, randomly sample possible waypoints that probabilistically cover the search space. The sampled waypoints are later used to construct data structures such as graphs and trees for

discrete search. A general sampling-based motion planning algorithm is shown in [56]. After the initial solution is obtained, it can be further refined to obtain more optimal paths.

Classical sampling-based methods include random roadmap, and RRT-like algorithms such as RRT, RRT* [45], BIT* [96], FMT* [42], etc. The random roadmap covers the search space with graphs constructed from random samples. It can reuse graphs for the same environment [56]. Notice that the idea of storing graphs and reusing sees a possible extension. Specifically, a life-long learning approach in [7] stores a database of past feasible paths to aid future planning. However, it does not utilize the advances in machine learning and deep learning. In terms of the order of sampling, RRT-like methods can be classified into sequential sampling and batch sampling. Sequential sampling iteratively samples, and thus can be combined with online heuristic methods such as online learning. On the other hand, Batch sampling obtains all the samples at once.

For heuristic algorithms, there are at least two potential components in the algorithm where heuristics can be applied: sampling step and distance metrics. Past work such as informed-RRT* and FMT* explored heuristic sampling methods. Heuristics in distance metrics are useful in kinodynamic motion planning, since the search space is non-Euclidean.

Since sampling-based motion planning algorithms do not require discretization of the environment but rely on random sampling, classical methods achieve asymptotically probabilistic completeness and optimality[45].

Kinodynamic Motion Planning

Kinodynamic motion planning is hard as the dynamics constraints are hard to satisfy. In [44], the BVP problem for connecting two waypoints is proved to be NP-hard. In literature, there exist two general approaches for planning with motion constraints: direct optimization and hierarchical approach using waypoints.

Direct optimization formulates the problem as an optimal control problem. Hence algorithms in optimal control [57], and reinforcement learning [8] can be applied to this problem. The

advantage of this formulation is that theoretical advances in optimal control can be applied. For instance, sufficient conditions for optimality can be obtained from the Hamilton–Jacobi–Bellman equation [50]. In contrast, necessary conditions can be obtained from the Pontryagin's maximum principle [64]. These conditions can be used to characterize and possibly guide the search for optimal solutions.

However, as stated in [38], RL, which can be considered one algorithm in optimal control, lacks long-horizon planning capabilities. To deal with this challenge, A recent method tries to combine ideas of waypoints in motion planning with RL [38]. Our work also follows similar ideas of combining long-term planning and short-term optimization for connections.

The hierarchical approach satisfies the motion constraints by extending the local connector. The local connector can be implemented by random control sequences [24], motion primitives [88], and exact BVP solvers [107]. The BVP solver can usually be implemented by trajectory optimization method [107], or as local optimal control problems [16].

Specifically, random sampling of control sequences has the advantage of theoretical elegance. [58] uses this to prove the probabilistically asymptotic completeness and optimality of the SST (Sparse-RRT) method. Motion primitives can add more heuristics to planning as in [94]. However, as the motion primitive set is finite, this can suffer from a lack of completeness. BVP solvers cannot achieve actual completeness due to the time complexity. However, this method can allow more freedom than motion primitives, and provide more information, such as optimization loss. [107][100] are two works on this line of research.

Heuristics methods in kinodynamic motion planning have works in [53][108][61]. Specifically, [53][108] rely on an initial solution to obtain the informed region defined as

$$\{x : f(x) + h(x) \leq c\},$$

where f, h, and c denote the cost from start, the heuristic cost to goal, and the cost of previously found solution. [53] uses rejection sampling method, while [108] achieves this by iterative

sampling through Markov Chain Monte Carlo. [61] also uses heuristic distance, but defines several local regions around tree node, termed as DIS (Dominance-Informed Regions), to obtain informed planning. [68] develops theoretical results for admissible heuristics in kinodynamic motion planning.

2.2 Deep Learning

We use deep learning in this thesis to achieve three tasks:

- to learn a good representation of the environment which encodes the obstacle shapes
- to learn a good representation of the waypoints representing the trajectory
- to achieve continual learning in motion planning

In this section, we will cover the review for general deep learning in shape learning and continual learning, while also covering some related theoretical insights from information geometry.

2.2.1 Shape Learning

The theory of shape learning first starts from differential geometry [14] and spectral geometry [54]. Shape learning studies about learning a compact representation of the shape, which is related to the topic of data embedding in research [39]. In machine learning, data embedding belongs to the field of unsupervised learning, which tries to extract inter-dependencies within data. Several classical data embedding methods, including PCA [104] and isomap [4] are used to extract the embedded representation of data input.

In deep learning, autoencoders often achieve data embedding. Autoencoders are constructed by an encoder and a decoder [31]. The encoder embeds the data into a lower-dimensional latent space. At the same time, the decoder reconstructs the input from the latent space [31]. In this line of research, several advanced techniques include generative adversarial networks (GAN) [110] and variational autoencoder (VAE) [73].

Since we use point clouds and voxels to represent the objects, we will introduce here shape learning using these two representations. For deep learning on point clouds, several novel methods include [75][43]. The insights in the paper use the observation that the point cloud is an unordered set [75]. Hence functions such as sum and max, which preserve the unordered nature, need to be used in the neural networks [75]. For deep learning on voxels, the representation groups point clouds into several cells and achieve a more compact yet coarse representation of the object [111].

2.2.2 Trajectory Learning

Motion planning and control are achieved in deep learning by several methods, including reinforcement learning and imitation learning. Typically, reinforcement learning is an optimal control algorithm that uses a neural network as a general function approximator [98]. It iteratively improves the trajectory or policy by several methods. These methods include using the Bellman update rule in the value iteration method and the policy iteration method [98] and directly optimizing the expected trajectory reward by Monte Carlo sampling [99]. Thus RL implicitly learns the trajectory distribution by optimizing the reward. On the other hand, imitation learning learns the trajectory distribution by imitating the expert behaviors from recorded trajectories.

Imitation learning can be modeled as the following optimization problem:

$$\begin{aligned} \min_{\theta} \quad & D(\tau_\theta, \tau_e) \\ \text{s.t.} \quad & \text{constraints on } \tau_\theta \end{aligned} \tag{2.1}$$

Here τ_θ denotes the learned trajectory parameterized by θ, and τ_e denotes the expert trajectory. The constraints on the estimated trajectory include dynamics constraints and boundary conditions. Often due to the noise in practice and possible variations in the expert policy, the trajectory is considered to follow certain probabilistic distributions. The discrepancy between these probabilistic distributions can be modeled with several options, including KL divergence, Total

Variation, Jensen-Shannon divergence, Earth-Mover distance, and Wasserstein distance [2].

KL divergence is widely used in literature as in [67]. Since KL divergence is not symmetric, there are two different optimization problems, referred to as I-projection and M-projection in [67]. I-projection in our formulation is the following optimization problem:

$$\min_{\theta} D_{KL}(P_\theta, P_e) = \min_{\theta} \mathbf{E}_\theta[\log P_\theta(\tau)] - \mathbf{E}_\theta[\log P_e(\tau)],$$

while M-projection is given by:

$$\min_{\theta} D_{KL}(P_e, P_\theta) = \min_{\theta} \mathbf{E}_e[\log P_e(\tau)] - \mathbf{E}_e[\log P_\theta(\tau)].$$

Here $\mathbf{E}_\theta$ and $\mathbf{E}_e$ denotes the expectation w.r.t. the distribution $P_\theta(\cdot)$ and $P_e(\cdot)$.

According to [67], I-projection is seldom used, as the expectation w.r.t. the distribution $P_\theta(\tau)$ is hard to approximate. On the other hand, $\mathbf{E}_e$ can be approximated by the Monte Carlo sampling method. Notice that minimizing the M-projection is equivalent to maximizing the log-likelihood for parameter θ, i.e.

$$\arg\min_{\theta} D_{KL}(P_e, P_\theta) = \arg\max_{\theta} \mathbf{E}_e[\log P_\theta(\tau)].$$

which is the maximum likelihood principle [66].

After obtaining an idea of the loss function, we classify the important methods in imitation learning area by three categories: sequential state-action model, motion primitive, and state distribution model.

Sequential state-action model

The sequential state-action model uses the MDP formulation, which shows the maximization of the expected log-likelihood of trajectories satisfies:

$$\begin{aligned}\arg\max_{\theta} \mathbf{E}_e[\log P_\theta(\tau)] &= \arg\max_{\theta} \mathbf{E}_e[\log P(x_0)\Pi_t P(x_{t+1}|x_t,u_t)P_\theta(u_t|x_t)] \\ &= \arg\max_{\theta} \mathbf{E}_e[\log P(x_0) + \sum_t \log P(x_{t+1}|x_t,u_t) + \sum_t \log P_\theta(u_t|x_t)] \\ &= \arg\max_{\theta} \mathbf{E}_e[\sum_t \log P_\theta(u_t|x_t)]\end{aligned} \tag{2.2}$$

Thus under the same dynamics and initial state distribution, the problem reduces to minimizing the divergence between conditional distributions of state-action pairs. In RL, this is referred to as policy learning [99]. This category has two sub-categories of methods: direct supervised learning and distribution matching.

The direct supervised learning approach uses the likelihood maximization principle as in equation 2.2. One important algorithm in this direction is [84]. In this direction, the key challenge is that the actual state experienced may not be in the support of the expert data [67]. Several methods might help with this problem, including confidence-based approach [15], dynamical recovery, and interactive data aggregation [84].

The distribution matching method comes from research in RL. This method matches the distribution of state-action pairs, which, under the assumption of MDP, can be defined as [74]

$$\mathscr{D} = \{\rho : \rho \geq 0, \sum_u \rho(x,u) = p_0(x) + \gamma \sum_{x',u} P(x|x',u)\rho(x',u), \forall x\}.$$

In the RL community, this is referred to as occupancy measure, and it uniquely maps to a policy [37]. However, the algorithms do not explicitly achieve distribution matching, but rather through inverse reinforcement learning (IRL). The relation between IRL and occupancy measure is later found out in [37].

From the observation that RL requires a predefined cost function, IRL learns a cost function for which the expert policy is optimal. Hence IRL often uses RL as a subroutine to find the optimal policy for a given cost function. [37] provides a comprehensive framework to analyze the theoretical relation between RL, IRL, and distribution matching. It observes that the relationship between IRL and occupancy matching closely connects to the optimization problem's primal-dual structure. Indeed [37] shows that IRL with constant regularization function is the dual program of occupancy measure matching problem. Moreover, by setting a specific regularization function, [37] gives a formulation of IRL using GAN.

Meanwhile, closely related to this line of research, recent work also uses occupancy measure as a probability distribution to learn the dynamics function, or transition model [106].

Learning Motion Primitive

Motion primitive methods directly parameterize the dynamics function $f_\theta(x, u)$, and match with the expert demonstration by implicitly learning the control trajectory u. Several important methods include DMP [91], ProMP [70], and SEDS [48]. They explicitly model the trajectory as a parameterized control system and obtain the parameters by tracking [91], or through learning using the Gaussian model [70][48]. Because they explicitly model the dynamics function, some methods such as [48] can even obtain theoretical stability properties, which is not possible in other methods.

Learning State Distribution

In the expert trajectories, the state correlation might be strong enough to determine the future state, without information about the controls. If this is true, we may reduce the trajectory learning problem to the state distribution matching problem. A more generalized version is treated as trajectory feature distribution learning in [67]. With the learned state trajectory distribution, the control sequence may be later reconstructed by trajectory optimization method, if the dynamics model is known [67]. This can also benefit the latter process as trajectory

optimization is greatly affected by initialization [67].

This line of research is useful in motion planning to obtain informative waypoints. For instance, sampling-based methods can be accelerated by providing informative samples as in [76][30]. Also closely related to this is the human pose trajectory learning [59][90], as human poses can be considered as states in kinodynamic motion planning problems.

Most work in this direction uses a sequential model. For instance, in [78], a transition model $f_\theta(x_{t+1}|x_0, x_t, x_T, W)$ is constructed to solve the motion planning task from x_0 to x_T in environment W. This model is iteratively used to generate a trajectory from the start point to the goal point. Although this work does not consider kinodynamic constraints, a possible future extension is to use trajectory optimization to connect planned waypoints [107].

The problem of the sequential model is the divergence from the expert distribution, as we introduced before. An analogy to this is the Lyapunov exponent in nonlinear dynamic systems [9]. A small error might lead to exponential divergence in the future when following a trajectory. This problem might be due to repeatedly calling the function as:

$$x_n = f(f(\dots f(f(x_0, u_0), u_1), \dots), u_{n-1})$$

Instead of using a sequential model that might cause chaos as in nonlinear dynamic systems, learning the distribution of the trajectory globally might be more beneficial. This direction is also beneficial as it also applies to situations when the Markov assumption does not hold.

Recent work in [62] uses a global Wasserstein Distance minimization of state visitation distribution between expert demonstrations and the learned policies. [38] uses methods similar to road-map to manually "memorize" the trajectory distribution. In this line of research, a more general study is needed to investigate how to best represent the trajectory distribution in the state space without a sequential model.

2.2.3 Continual Learning

In literature, continual learning is another name for lifelong learning, which has been studied for a long time [81][101][63]. Several theoretical results have done in this direction using linear models [5][72], with applications in [87][11]. The continual learning aims to solve the catastrophic forgetting problem [33], which emerges if naive algorithms are used in the continual learning setting.

Chapter 3

Methods

3.1 Learning-Based Collision Checker

In this chapter, we will explain details about our implementation of the collision checker using deep learning methods. Our collision checker takes the input of two point clouds representing the objects and returns the collision probability. We use recent advances in 3D computer vision community, specifically, PointNet [75], combined with siamese structure [18] to construct the neural network.

3.1.1 PointNet

PointNet is a network that takes advance of point cloud's unordered nature and uses point-wise operations to extract local and global features [75]. In our work, we use the network structure of PointNet as defined in figure 3.1, which is used as a submodule in the siamese structure for data embedding.

3.1.2 Siamese Net

The siamese structure was first introduced in [18], and is widely used in the deep learning community to extract distance information for complicated data manifolds [47]. It, at the same time, solves the data scarcity problem. The basic idea is to use the same network structure and weights for two different inputs to transform them into latent space. Later another network is

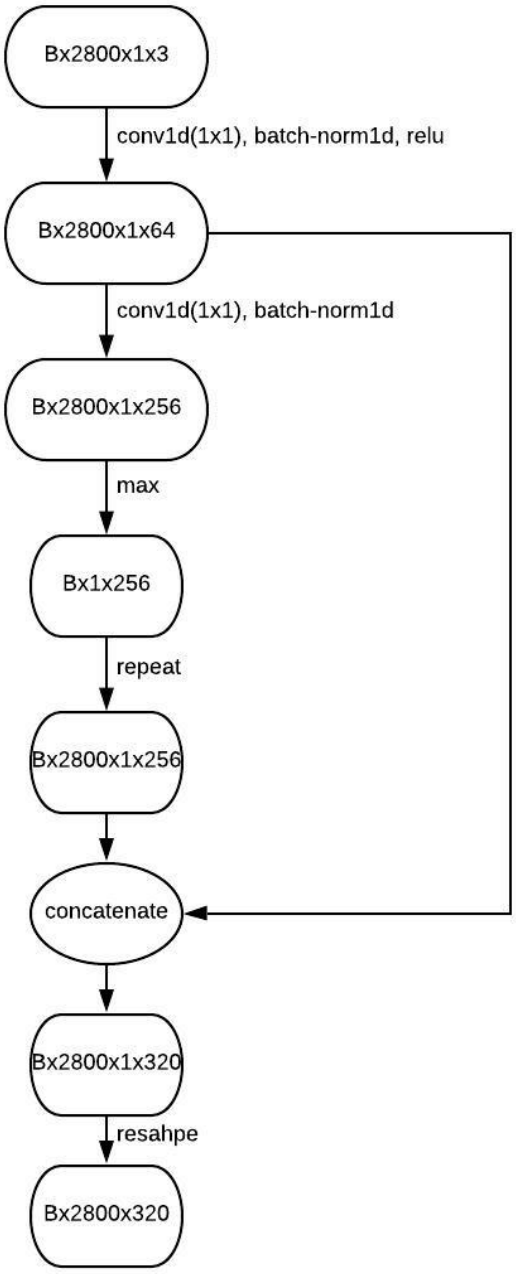


Figure 3.1. PointNet Structure

used to extract distance information from the latent space. In our work, we represent the collision probability as distance in the latent space.

We use the siamese structure as defined in 3.2, taking advantage of the previously defined

PointNet. Specifically, we transform the output of PointNet by a series of fully-connected layers

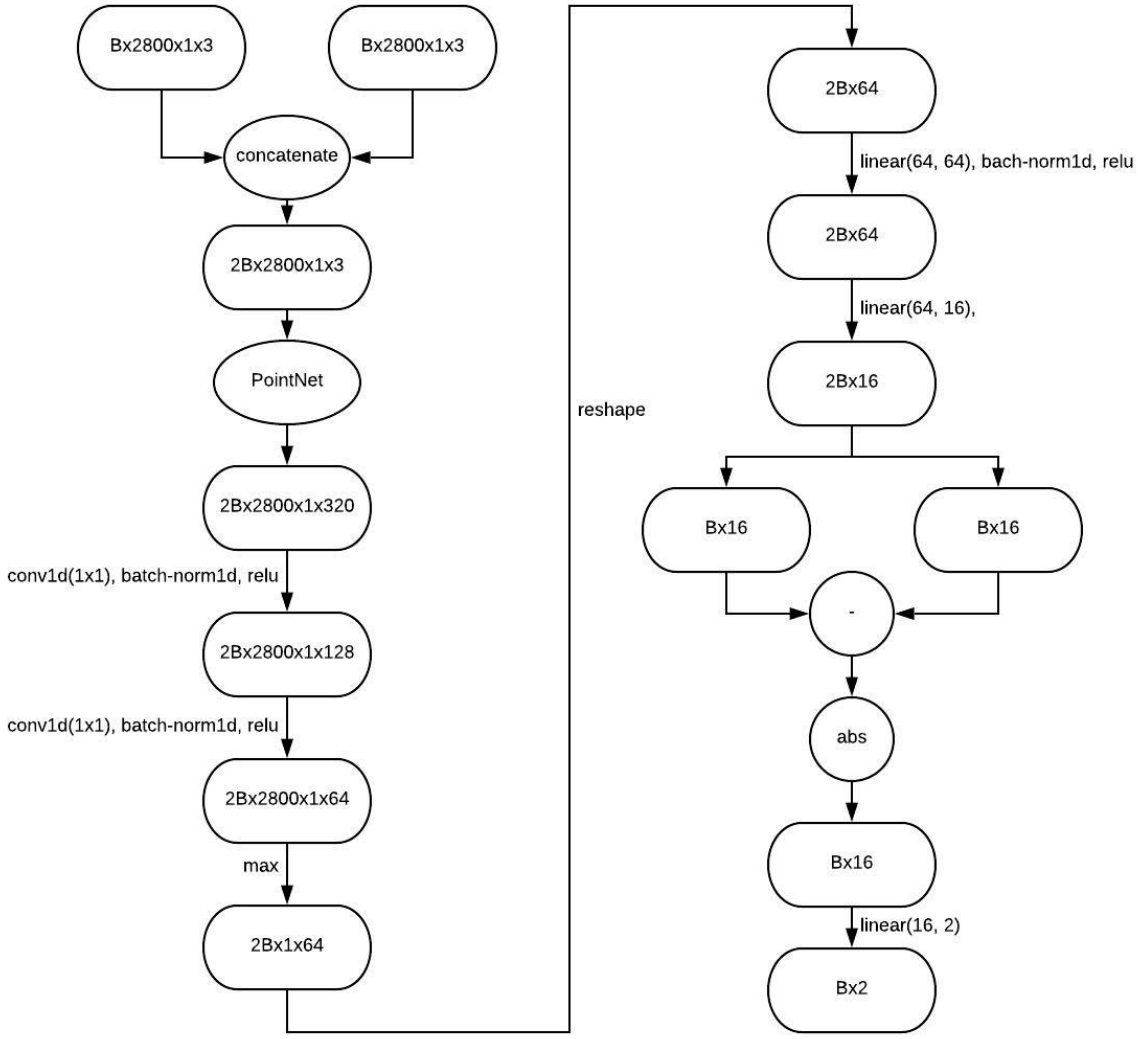


Figure 3.2. Siamese Net Structure

and then obtain the feature space, which is of dimension 16. We then compute the distance of the two feature vectors and obtain the absolute values. This is due to the symmetry of in-collision relation. To be more precise, the output of the network for two inputs (P_1, P_2) and (P_2, P_1) should be the same. After obtaining the absolute values, we further transform them by a fully-connected layer to a vector of dimension 2.

The output can be viewed as specifying the ”probability” of in-collision or collision-free.

To see this, if we further transform the 2d vector using the softmax function, we can ensure that the entries are non-negative and sum up to 1. Hence, by comparing the values of the two entries, we can obtain the decision for in-collision or collision-free. For this task, we use the first entry as the probability of choosing collision-free, and the second entry as the probability of choosing in-collision. We use the cross-entropy loss to measure the difference between neural network outputs and data labels.

3.1.3 Pipeline

Here we describe the pipeline for training and testing the collision checker.

Data Generation

We generate object pairs together with a label indicating if the pair is in collision or not in the format:

$$(P_1, P_2, o)$$

where P_1 and P_2 represent point clouds of the pair of objects. o takes values of 0 or 1, with 0 denoting not in-collision, and 1 denoting in-collision. We split the generated dataset into training dataset, validation dataset and testing dataset.

Training Pipeline

After obtaining the data through the data generation procedure, We use the standard deep learning training method to train the collision checking neural network on the training dataset. We also validate our training procedure on the validation dataset.

Testing Pipeline

After training, we test the trained neural network on the testing dataset, and compare the results with the ground truth.

3.1.4 Experiments and Results

For experiments and results, please refer to Section 4.1 in Chapter 4.

3.2 Learning-Based Motion Planner

In this thesis, we use a learning-based sequential prediction model to learn optimal paths' state distribution. After the state distribution is learned, we use this model as an oracle to produce informative waypoints in a bidirectional and recursive path planning algorithm. We separately give details about the predictive model and motion planning algorithm as follows.

3.2.1 Sequential Prediction Model

Network Structure

Our neural network contains two parts: the encoder network and the planning network. We term this neural network approach as MPNet (Motion Planning Network).

Our encoder network aims at learning useful features of the environment information. Our planning network takes the encoded environment representation, the current state in planning and goal state, and produce the next waypoint. Thus the planning network is iteratively used to generate a path in the state space from start to goal.

For the implementation of the encoder network, we use a fully-connected neural network for simple environments and VoxelNet for more complicated environments [111]. We also compared the implementation using PointNet and VoxelNet. We chose VoxelNet as PointNet suffers from computational time even for a relatively small point cloud in robotic tasks [75][111]. For the VoxelNet structure, we use 3D CNN layers combined with fully-connected layers. Details about the VoxelNet structure are given as in figure 3.3.

For the planning network implementation, we use a fully-connected neural network combined with dropouts to model stochasticity during planning. According to [28], dropout adds Gaussian randomness to the network. Together with the encoder, the entire structure is given as

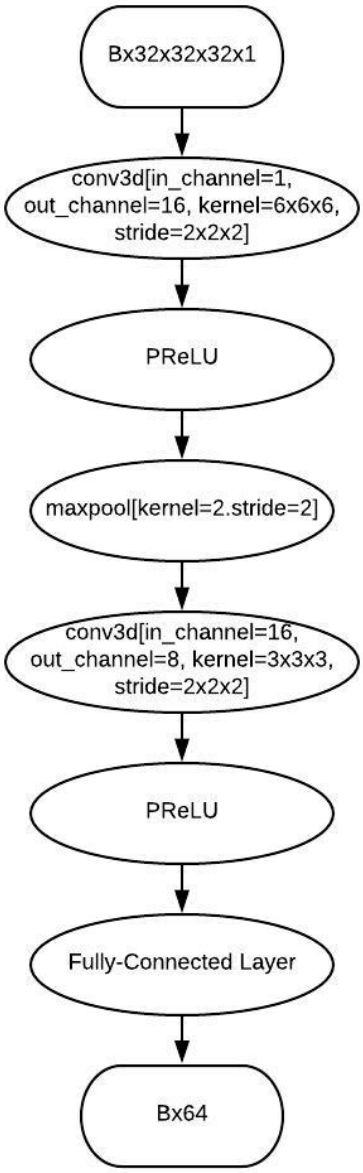


Figure 3.3. Voxel Encoder Structure

in figure 3.4.

3.2.2 Pipeline

For the pipeline of the prediction model, we first generate the paths and point clouds representing obstacles using classical optimal motion planners. Then we preprocess the obstacle

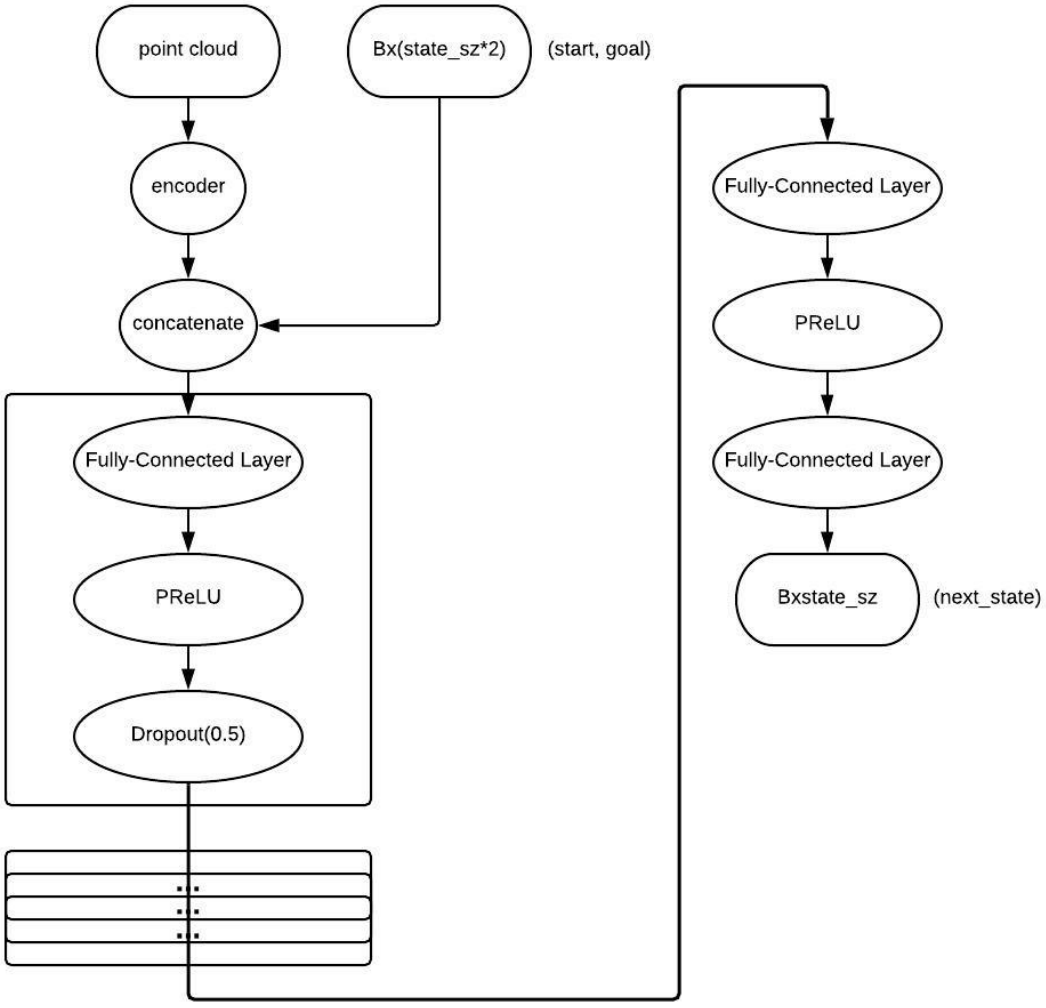


Figure 3.4. Full MPNet Structure

point clouds into voxels if using VoxelNet. Next, we transform the paths and obstacle into the structure of

$$(x_t, x_T, o, x_{t+1})$$

for training, where o denotes the obstacle representation. After training the network on the training dataset, we test the network on the unseen paths.

3.2.3 Motion Planning Algorithm

In this section, we give details about the motion planning algorithms using the trained sequential prediction model. Our idea is to use a bidirectional and recursive path-searching-and-refining method.

In general, we iterate between the forward planning and backward planning using a bidirectional searching method. For the forward planning, we use the backward endpoint as the target goal position and generate the next state leading to it. Similarly, we perform the same for backward planning. We first only use the predictive model to generate a sequence of informative waypoints. If the forward path and backward path are connectable, we will consider the neural planning as successful, and thus check the connectability by verifying collisions on the edges. If a certain part of the planned path is not connectable, we only repeat the planning procedure for the invalid node pairs, until the entire path is collision-free. Meanwhile, we also use a path smoothing method called lazy states contraction (LSC) to reduce the number of nodes and collision checks needed for the path. A detailed description of the algorithm is given in the algorithm 1.

Algorithm 1: neural-based geometric planning(x_0, x_G, O)

$z \leftarrow \text{encoder}(O)$;
$\tau \leftarrow [x_0, x_G]$; `// initialization for the path`
for $i \leftarrow 1$ **to** MAX_ITER **do**
    $\tau \leftarrow \text{replan}(\tau, z)$;
    $\tau \leftarrow \text{lazyStatesContraction}(\tau)$;
    if feasible(τ) **then**
        return τ;
return $\emptyset$; `// plan failed`

3.2.4 Experiments and Results

For experiments and results, please refer to Section 4.2 in Chapter 4.

Algorithm 2: replan(τ, z)

$\tau_{\text{new}} \leftarrow \emptyset$; // initialize the return path
for $i \leftarrow 1$ **to** size(τ) $- 1$ **do**
  if steerTo(τ_i, τ_{i+1}) **then**
    $\tau' \leftarrow \tau_{\text{new}} \cup [\tau_i, \tau_{i+1}]$; // store if connectable
  else
    $\tau' \leftarrow$ neuralPlan(τ_i, τ_{i+1}, z) ; // local plan using neural network
  $\tau_{\text{new}} \leftarrow \tau_{\text{new}} \cup \tau'$;
return τ_{new};

Algorithm 3: neuralPlan(x_0, x_G, z)

$\tau^a \leftarrow [x_0]$, $\tau^b \leftarrow [x_G]$;
$\tau \leftarrow []$;
direction $= 0$; // initialize the return path
for $i \leftarrow 1$ **to** MAX_PLAN **do**
  $x_{\text{new}} \leftarrow$ planNeuralNetwork($z, \tau^a_{\text{end}}, \tau^b_{\text{end}}$);
  $\tau^a \leftarrow \tau^a \cup [x_{\text{new}}]$;
  if steerTo($\tau^a_{\text{end}}, \tau^b_{\text{end}}$) **then**
    $\tau \leftarrow$ concatenate(τ^a, τ^b, direction);
    return τ;
  direction $\leftarrow 1 -$ direction;
  swap(τ^a, τ^b);
return $[x_0, x_G]$; // plan failed

3.3 Life-long learning in Motion Planning

In this chapter, we describe a life-long learning algorithm to make the learning-based motion planner improve in an online fashion as more planning is done.

We first describe a method called Gradient Episodic Memory [63] to make learning achievable from stream-line data. Then we incorporate this continual-learning framework into an active learning module, which trains MPNet only when its plan fails, thus achieving data efficiency.

Algorithm 4: lazyStatesContraction(τ)

```
for i ← 1 to τ.size() − 1 do
    for j ← τ.size() to i + 2 do
        if steerTo(τ[i], τ[j]) then
            /* can connect to later nodes besides adjacent nodes */
            return lazyStatesContraction(τ[1 : i] ∪ τ[j : τ.size()]);
return τ;
```

3.3.1 Continual Learning MPNet

Our method of continual learning builds upon a previous work called Gradient Episodic Memory [63]. The previous work solves the continual learning problem by combining an episodic memory with loss gradient projection. Specifically, an episodic memory M_k is constructed for each task k. Thus the loss on the memory M_k

$$l(M_k;\theta) = \frac{1}{|M_k|}\mathbf{E}_{(x_i,k,y_i)\sim M_k}[l(f_\theta(x_i,k),y_i]$$

can represent the loss on the task k. It thus tries to solve the following optimization problem:

$$\begin{aligned} \min_{\theta} \quad & l(f_\theta(x,t),y) \\ \text{s.t.} \quad & l(M_k;\theta) \leq l(M_k;\theta_{t-1}), \forall k < t \end{aligned} \tag{3.1}$$

where θ_{t-1} denotes the network parameters after training with samples on task $t-1$.

The idea is to ensure the inner product of gradients satisfies

$$\langle g, g_t \rangle = \langle g, \frac{\partial l(f_\theta(x,t),y)}{\partial \theta} \rangle \geq 0, \langle g, g_{M_k} \rangle = \langle g, \frac{\partial l(M_k;\theta)}{\partial \theta} \rangle \geq 0,$$

where g is the gradient used in the gradient descent, g_t and g_{M_k} denote the gradient of loss on current batch and the loss on M_k. Then gradient descent can ensure the loss on the current data and the episodic memory is non-increasing. This is the key step in [63].

Unlike [63], which targets reducing the effect of catastrophic forgetting on previous tasks, we focus on learning with stream-line data. Thus our problem considers the loss on the current batch of data and the loss of past samples. We want to ensure that the overall loss gradually decreases. Hence our problem is a more relaxed version of the continual learning problem. Thus instead of constructing an episodic memory for each task, we only construct the memory for the past data. Thus our problem becomes:

$$\begin{aligned} \min_{\theta} \quad & l(f_\theta(x_t), y_t) \\ \text{s.t.} \quad & l(M;\theta) \leq l(M;\theta_{t-1}) \end{aligned} \tag{3.2}$$

Here (x_t, y_t) denotes the data sample at time t, and θ_t denotes the network after trained on the t-th sample. Thus by using the gradient projection method, we need to ensure the gradient g used in gradient descent satisfies:

$$\langle g, g_t \rangle = \langle g, \frac{\partial l(f_\theta(x_t), y_t)}{\partial \theta} \rangle \geq 0, \langle g, g_M \rangle = \langle g, \frac{\partial l(M;\theta)}{\partial \theta} \rangle \geq 0,$$

To make sure this condition is valid, we follow a similar method as in [63] to project the gradient of loss on the current batch to the previous data:

$$\begin{aligned} \min_{g} \quad & \frac{1}{2} \|g_t - g\|_2^2 \\ \text{s.t.} \quad & \langle g, g_m \rangle \geq 0 \end{aligned} \tag{3.3}$$

This can be solved by quadratic programming [63].

Moreover, we also investigate several approaches to implement the episodic memory. Specifically, by following a previous paper [40], we experiment with sampling methods by surprise maximization, reward maximization, coverage maximization, and global distribution matching. As a result, we use the global distribution matching method to construct the memory. For details, please see chapter 4. We use a global distribution matching method called reservoir

sampling, which can obtain uniform sampling from a stream of samples in an online fashion. The algorithm for reservoir sampling is described in algorithm 5.

Algorithm 5: reservoirSample(i (data number), d (data), M)

```
if i ≤ M.capacity then
    M ← M ∪ [d]
else
    if uniformSample(0,1) ≤ M.capacity / i then
        /* with probability M.capacity / i, keep the new data          */
        k ← randomInteger(1, M.capacity) ;  // obtain the memory index to
         replace with the new data
        M[k] ← d;
```

Meanwhile, we also use rehearsal to improve the performance in the continual learning setting, by following [82]. Specifically, we keep a separate memory B that stores a subset of the data seen, and trains the network periodically on a uniform sampled batch from this memory. We also treat the replay batch as new data seen and use the gradient projection method for training. However, We do not store the replay batch again into the episodic memory. Since in our problems, the memory is enough to hold the entire training data, we use B to store all the data seen.

With these defined, our algorithm for continual learning is described in algorithm 6. We term this learning-based method as CMPNet.

3.3.2 Active Learning MPNet

After introducing CMPNet, we further extend it to an active learning setting, thus making the algorithm more practical. Our active-learning MPNet (ACMPNet) considers an online-learning scenario where the classical motion planners can be used as a backup. Our idea is that after some training samples, MPNet has encapsulated some trajectories, but is still uncertain in other cases. To distinguish learned paths from paths that have not been learned, we use the criterion of success during neural planning. If motion planning with MPNet is successful, we

Algorithm 6: continualLearningUpdate(t (training iteration), (x_0, x_G, O) (planning problem), θ (network parameters), M (episodic memory), B (rehersal memory))

Predefined parameters: r (replay period), N_B (replay batch size);
$\tau \leftarrow \text{getExpertDemonstration}(x_0, x_G, O)$;
$D_t \leftarrow \text{transformToTrainingData}(\tau, \text{O})$;
$\text{reservoirSample}(\text{t}, \text{D}_\text{t}, \text{M})$;
$B \leftarrow B \cup D_t$;
$g_t \leftarrow \mathbf{E}_{(x,y)\sim D_t} \frac{\partial l(f_\theta(x),y)}{\theta}$;
$g_M \leftarrow \mathbf{E}_{(x,y)\sim M} \frac{\partial l(f_\theta(x),y)}{\theta}$;
$g \leftarrow \text{QPProjection}(g_t, g_M)$; `// quadratic programming for Equation 3.3`
update parameters θ using g;
`/* rehersal */`
if $(B.\text{size}() > N_B$ *and* $(t\%r == 0)$ **then**
    $b \leftarrow \text{uniformSampleBatch}(B)$;
    $g_t \leftarrow \mathbf{E}_{(x,y)\sim b} \frac{\partial l(f_\theta(x),y)}{\theta}$;
    $g_M \leftarrow \mathbf{E}_{(x,y)\sim M} \frac{\partial l(f_\theta(x),y)}{\theta}$;
    $g \leftarrow \text{QPProjection}(g_t, g_M)$; `// quadratic programming for Equation 3.3`
    update parameters θ using g;

assume the MPNet has already learned enough about the environment embedding and the path distribution. Otherwise, we need to train MPNet.

With this idea in our ACMPNet algorithm, we call the classical motion planners when MPNet fails, and only use the failure paths to train MPNet. We thus expect an improvement in data efficiency, as only part of the data is needed. Meanwhile, this algorithm is a prototype for a realistic online planning setting. The algorithm is described in algorithm 7.

3.3.3 Experiments and Results

For experiments and results, please refer to Section 4.2 in Chapter 4.

Algorithm 7: activeLearningUpdate(t (training iteration), (x_0, x_G, O) (planning problem), θ (network parameters), M (episodic memory), B (rehersal memory))

Predefined parameters: r (replay period), N_B (replay batch size), N_C (number of iterations to pretrain MPNet);
`/* pretrain MPNet */`
if $t \leq N_C$ **then**
    continualLearningUpdate$(t, (x_0, x_G, O), \theta, M, B)$;
    return;
`/* active learning */`
$\tau \leftarrow$ neuralBasedPlanning(x_0, x_G, O) ; `// using MPnet to plan first`
if $\tau == \emptyset$ **then**
    `/* neural planning failed. Use expert demonstration */`
    continualLearningUpdate$(t, (x_0, x_G, O), \theta, M, B)$;

3.4 Learning-Based Kinodynamic Motion Planning

3.4.1 Adding Kinodynamic Constraints

We further extend the learning-based motion planning algorithm by adding kinodynamic constraints specified by

$$\dot{x_t} = f(x_t, u_t).$$

In our problems, we use linear discretization to discretize this differential constraint into

$$x_{t+1} = x_t + dt * \dot{x_t}.$$

with some specified dt. Hence the algorithm needs to plan both a state trajectory and a control trajectory. To incorporate the previously introduced sequential motion planner in kinodynamic motion planning tasks, we use insights from [107]. We change the steering function to a BVP solver implemented by the cross-entropy method (CEM) [10].

A BVP solver tries to obtain the intermediate state trajectory and control trajectory given specified start and goal states, through optimization method. Specifically, it considers the

following optimization problem:

$$\begin{aligned} \min_{x,u} \quad & J(x,u;W) \\ \text{s.t.} \quad & \dot{x}(t) = f(x(t),u(t)), \forall t \text{ (motion model)} \\ & x(t) \in \mathscr{X}, u(t) \in \mathscr{U} \text{ (state boundary and control boundary)} \\ & x(0) = x_0 \text{ (start constraints)} \\ & x(T) \leq x_T \text{ (goal constraints)} \\ & \text{other constraints such as collision constraints} \end{aligned} \tag{3.4}$$

This is similar to the problem as defined in section 1.2.1. However, BVP has smaller horizons, and approximate optimization solvers can often find solutions fast. Algorithm 8 describes our implementation of the BVP solver using CEM. Specifically, it iteratively estimates the Gaussian distribution that minimizes the distance between the endpoint and the goal.

We obtain a line-search planning algorithm (MPC-MPNetLine) and a tree-based planning algorithm (MPC-MPNetTree) for the kinodynamic motion planning task. In both algorithms, we maintain a search tree recording the history of planning. The differences between these two algorithms lie in the orders for the expansion of nodes.

3.4.2 Line-Search Planning

We adapted the geometric line-based motion planning algorithm to the kinodynamics motion planning by changing it to unidirectional planning and removing the lazy vertex contraction. These changes come from the challenges of kinodynamic motion planning.

Firstly, in kinodynamic motion planning, simple discretization methods will cause the dynamic propagation to be asymmetric concerning the direction. For example, forward propagation

$$x_t + dt * f(x_t, u_t)$$

Algorithm 8: MPC(x_0, x_G)

initialize the distribution of controls $\mu_u[0..NT]$, $\sigma_u[0..NT]$, and time $\mu_t[0..NT]$, $\sigma_t[0..NT]$;
for $iter \leftarrow 1$ **to** MAX_ITER $+ 1$ **do**
    /* sample N control and time trajectories */
    for $i \leftarrow 1$ **to** N **do**
        for $j \leftarrow 1$ **to** NT **do**
            $u[i,j] \leftarrow \mathcal{N}(\mu_u[j], \sigma_u[j]))$;
            $t[i,j] \leftarrow \mathcal{N}(\mu_t[j], \sigma_t[j]))$;
    /* evaluate the samples to obtain loss */
    for $i \leftarrow 1$ **to** N **do**
        $x[i] \leftarrow$propagate$(u[i], t[i])$;
        if *IsInCollision*$(x[i])$ **then**
            $l[i] \leftarrow \infty$;
        else
            $l[i] \leftarrow d(x[i,-1], x_G)$;
    /* select top-k of loss, and update the distribution */
    $\phi_l \leftarrow$ topKIndices$(-l)$; // obtain the k indices that obtain the lowest k losses
    /* update the statistics */
    for $j \leftarrow 1$ **to** NT **do**
        $\mu_u[j] =$mean$(u[\phi_l, j])$;
        $\sigma_u[j] =$std$(u[\phi_l, j])$;
        $\mu_t[j] =$mean$(t[\phi_l, j])$;
        $\sigma_t[j] =$std$(t[\phi_l, j])$;
/* return the trajectory with the lowest cost */
return $x[0], u[0], t[0]$;

and backward propagation

$$x_{t+1} - dt * f(x_{t+1}, u_{t+1})$$

will create two different trajectories. Meanwhile, due to the differential motion constraints, a small change in the state space might be unattainable. Hence we cannot guarantee successful connections between any two points. Thus naive bidirectional method and lazy vertex contraction may not be valid. The issues are evident by using the analysis of the controllability of each point in the state space [34]. In literature, these problems are bypassed by defining a valid

region around each node [100][13][55]. However, as controllability cannot be guaranteed, this bypassing might cause problems during the path's real executions. Due to this consideration, we do not use the bidirectional method as in geometric motion planning. The details of the algorithm are described in 9.

Algorithm 9: MPC-MPNetLine(x_0, x_G, O, d, V_G)

$z \leftarrow$ encoder(O);
$x \leftarrow x_0$;
/* initialize the search tree */
T $\leftarrow \{x_0\}$;
for $i \leftarrow 1$ **to** MAX_ITER **do**
    /* obtain waypoints from MPNet */
    $x_{\text{new}} \leftarrow$ planNeuralNetwork(z, x, x_G);
    $\tau_{\text{local}} \leftarrow$ steerTo(x, x_{new});
    /* if local steering is unsuccessful, then reset the state to a random point in the tree */
    if $\tau_{local} == \emptyset$ **then**
        $x \leftarrow$ randomNode(T);
    else
        /* if local steering is successful, then set the state as the endpoint of the local trajectory */
        addToTree(τ_{local}, T);
        if $d(x, x_G) \leq \varepsilon_G * V_G$ **then**
            /* if the state is near goal region, then reset to start */
            $x \leftarrow x_0$;
        else
            $x \leftarrow \tau_{\text{local}}[-1]$;
    if $d(\text{T}, x_G) \leq V_G$ **then**
        return solution path from T;

In this algorithm, we also added tricks to make it work more efficiently in practice. One example trick is periodic goal sampling. These tricks are standard in motion planning literature [56].

Algorithm 10: steerTo(x_t, x_{t+1})

```
/* a general function that returns a local trajectory connecting
   two nodes.  Here we show implementation using BVP solver.   */
x_local, u_local, t_local ← CEM(x_t, x_{t+1});
/* BVP solution might be approximate and does not follow system
   dynamics.  Hence we propagate again using the solution to
   find the path according to the dynamics                      */
τ ← [x_t];
for i ← i to u_local.size() do
    N_dt = round(t_local[i]/dt);
    for j ← 1 to N_dt do
        x_new ← propagate(τ_end, u_local[i], dt);
        if feasible(x_new) then
            τ ← τ ∪ [x_new];
        else
            /* collision happens, propagation ends here        */
            return τ;
return τ;
```

3.4.3 Tree-based Planning

From the observation that randomness affects line-based planning, we use tree-based planning to improve the sample efficiency. Our tree-based algorithm takes insights from [58]. We generalize the algorithm to algorithm 11.

Notice that we can use parallel methods to implement MPC, neural networks, and sampling-based methods. Hence our MPC-MPNetLine and MPC-MPNetTree methods can both be accelerated by vectorization and running on GPU. We obtain a CUDA implementation of the MPC-MPNetTree algorithm to further accelerate it by evaluating a batch of samples at once [89].

3.4.4 Experiments and Results

For experiments and results, please refer to Section 4.3 in Chapter 4.

Algorithm 11: neural-based kinodynamic tree-search planning with BVP solver(x_0, x_G, O, d, V_G)

```
T ← {x_0};                                   // initialize the tree structure
z ← encoder(O);
for i ← 1 to MAX_ITER do
    x_t ← randomSample();
    x_t ← nearestNeighbor(x_t, T);
    x_{t+1} ← planNeuralNetwork(z, x_t, x_G);
    τ_local ← steerTo(x_t, x_{t+1});
    addToTree(τ_local, T);
    if d(T, x_G) ≤ V_G then
        return solution path from T;
return ∅;
```

3.5 Chapter Acknowledgement

Section 3.2 and Section 3.3 in Chapter 3, and Section 4.2 in Chapter 4, in full, are a reprint of the material with minor modifications as it appears in Motion Planning Networks: Bridging the Gap Between Learning-Based and Classical Motion Planners 2020. Qureshi, Ahmed H.; Miao, Yinglong; Simeonov, Anthony; Yip, Michael C., IEEE Transactions on Robotics, 2020. The thesis author was the co-author of the published paper.

Section 3.4 in Chapter 3, and Section 4.3 in Chapter 4, in full is currently being prepared for submission for publication of the material. Authors include Li, Linjun; Miao, Yinglong; Qureshi, Ahmed H.; Yip, Michael C.. The thesis author was the shared first author of this material.

Chapter 4

Experiments and Results

In this chapter, we present our experimental results for the algorithms in the previous chapters.

For all our experiments, we run on a system with 3.40GHz×8 Intel Core i7 processors, 32 GB RAM and GeForce GTX 1080 GPUs.

4.1 Learning-Based Collision Checker

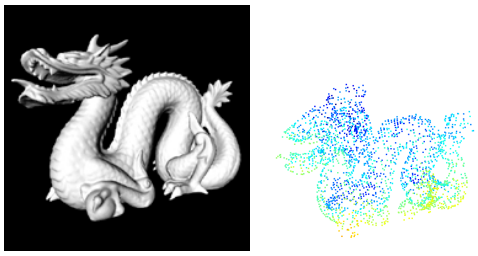
In this section, we present the results of our learning-based collision checker on two sets of data, namely, simple geometry dataset and complex shape dataset. For each dataset, we randomly place the objects and generate labels for collision checking.

4.1.1 Data Generation

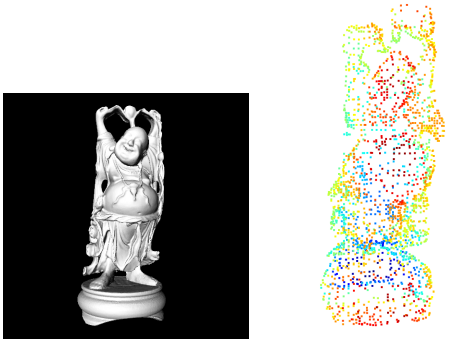
We generate two sets of data, simple geometry and complex shapes. For simple geometry, we generate two types of shapes, boxes, and balls. We firstly generate point clouds for standard shapes, and then randomly change the size and location within specific ranges. After obtaining the transformed point clouds, we obtain the meshes from them and then check collisions using the meshes. This gives us the labels for a pair of shapes. Examples are shown in the figure 4.1.

For complex geometry, we obtain shapes including the dragon, happy buddha, horse, and bunny from the Large Geometric Models Archive [102]. Figure 4.2 shows pictures for each of these models.

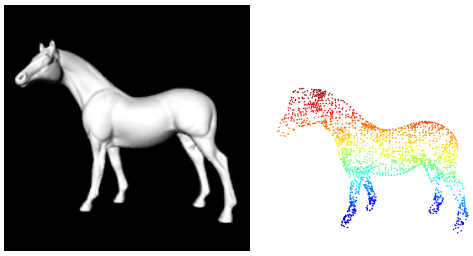
Figure 4.1. Simple Shapes for Collision Checking



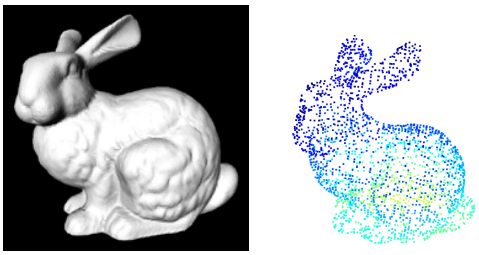
(a) Dragon Mesh and Point Cloud



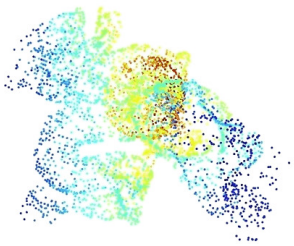
(b) Happy Buddha Mesh and Point Cloud



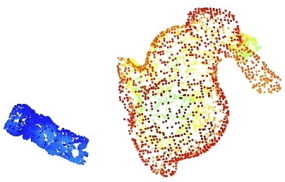
(c) Horse Mesh and Point Cloud



(d) Bunny Mesh and Point Cloud



(e) In-Collision Case for Dragon and Happy Buddha



(f) Collision-Free Case for Bunny and Happy Buddha

Figure 4.2. Complex Shapes for Collision Checking

We use similar generation procedures as simple geometric data, and randomly select a pair of shapes for inputs.

By following the above procedures, we generate data in the format:

$$(P_1, P_2, o)$$

where P_1 and P_2 represent point clouds of the pair of objects. o takes values of 0 or 1, with 0 denoting not in-collision, and 1 denoting in-collision.

We use pyntcloud to convert point clouds into meshes, and use Klamp't (Kris' Locomotion and Manipulation Planning Toolbox) packages to check collisions between meshes.

4.1.2 Training and Testing

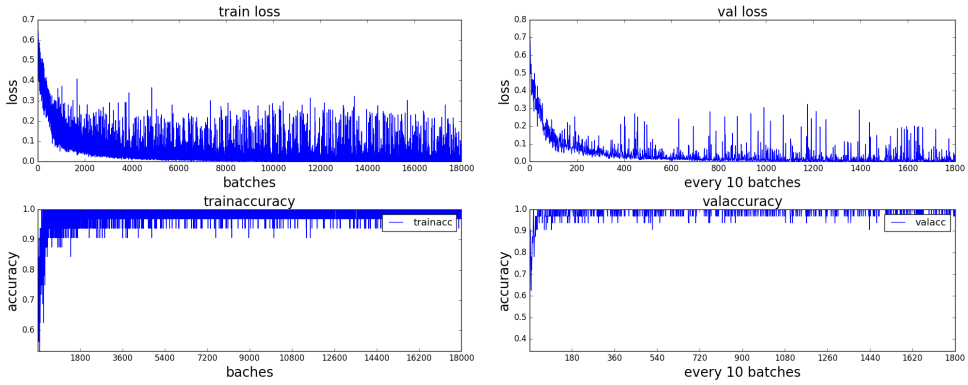
After obtaining the data through the data generation procedure, we separately train a model for the simple and complex dataset. We use the standard deep learning training method to obtain the model.

On the simple geometry dataset, we have a total of 80000 pairs of objects. Each object is represented using 2800 points. We use 57600 pairs as the training dataset, 8000 pairs as validation dataset, and 14400 pairs as the test dataset. We train the neural network for a total of 10 epochs, with a batch size of 32. Meanwhile, we use Adam optimizer [49] with learning rate 0.001, and gamma parameter 0.5.

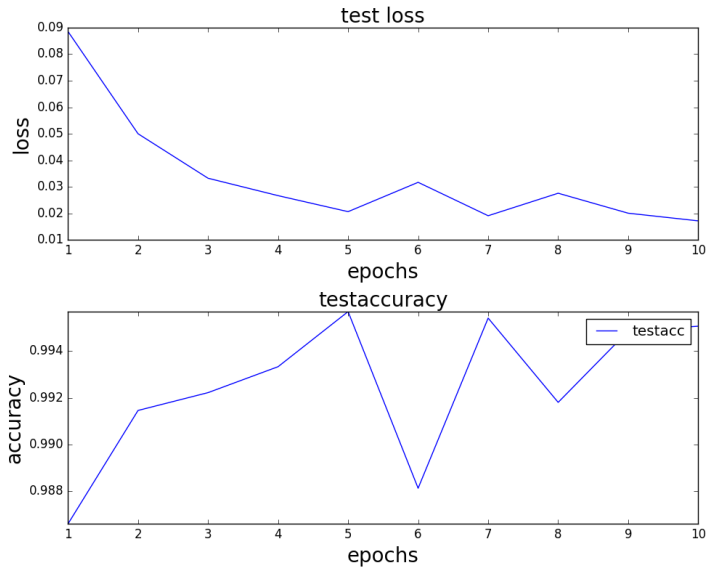
After training separately on the simple and complex dataset, we test the trained model on them. We do not cross test the models as deep learning assumes training and testing dataset comes from similar distributions.

4.1.3 Results and Analysis

Here we present the experimental results of the learning-based collision checker on the previously mentioned dataset. Figure 4.3 is the evolution of loss and accuracy for training, validation, and testing stages. For complex geometry dataset, we use the same setting, and figure 4.4 shows the results.



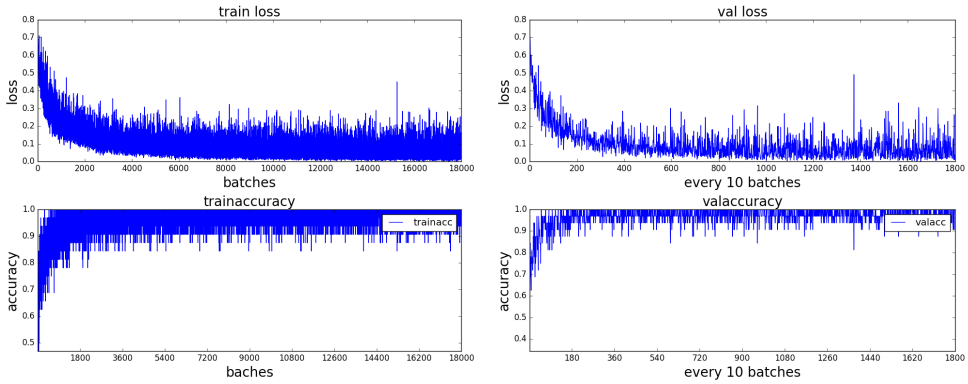
(a) loss and accuracy during training and validation on simple geometry



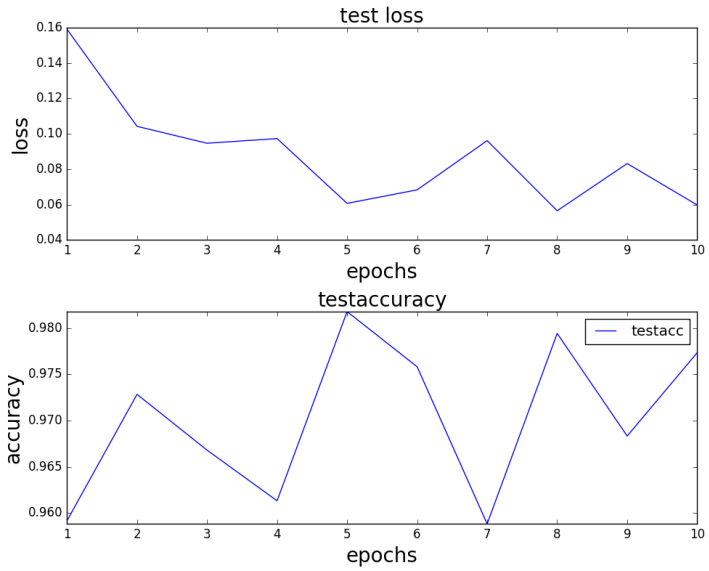
(b) loss and accuracy during testing on simple geometry

Figure 4.3. Results on Simple Geometry

Both of these results show the testing accuracy of more than 96%, which shows that the learning-based approach is promising. We use the same neural network structures for both simple geometry and complex geometry dataset. Hence we only need to record the average time for neural network propagation to represent the collision checking time. The average time for collision checking for both cases is 0.0088s. Since preprocessing is not needed in our approach, this result is much faster than the classical methods introduced before.



(a) loss and accuracy during training and validation on complex geometry



(b) loss and accuracy during testing on complex geometry

Figure 4.4. Results on Complex Geometry

4.2 Learning-Based Motion Planning

In this section, we present the results of our learning-based motion planner on six different environments, namely, 2D point-mass with simple and complex environments (S2D and C2D), 3D point-mass (C3D), SE2, SE3, and 7DOF manipulation planning. For all of the environments, we create different obstacles and plan with collision avoidance constraints.

4.2.1 Environment Setting

In the S2D environment, we randomly generate 7 square boxes with size 5 in random positions. Meanwhile, we randomly pick start and goal positions for planning and restrict the planning region to be 40x40. We generate 110 different obstacle configurations. Within each configuration, we generate 6000 paths with different start and goal positions using classical planners. The obstacles are represented by point clouds with size 1400x2. Figure 4.5 shows two planned paths of MPNet in two S2D scenes.

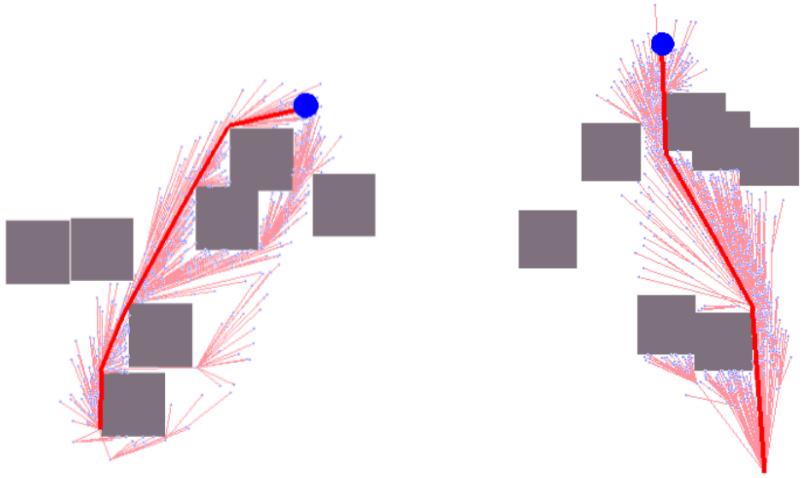


Figure 4.5. S2D Planning Scene

In the C2D environment, we randomly generate seven rectangles with different sizes in random positions. Like the S2D environment, we randomly pick start and goal positions and restrict the planning region to 40x40. We use the same data generation methods as in the S2D environment. Figure 4.6 shows two planning instances in two C2D scenes.

In the C3D environment, we randomly generate ten rectangular cuboids with different sizes in random positions. Similar to before, we randomly pick start and goal positions and restrict the planning region to be 40x40x40. We also use the same data generation methods as before. The point clouds representing obstacles have shape 1400x3. Figure 4.7 shows two planned paths in two C3D scenes.

In the SE2 environment, we use the S2D environment setting, but consider planning for a

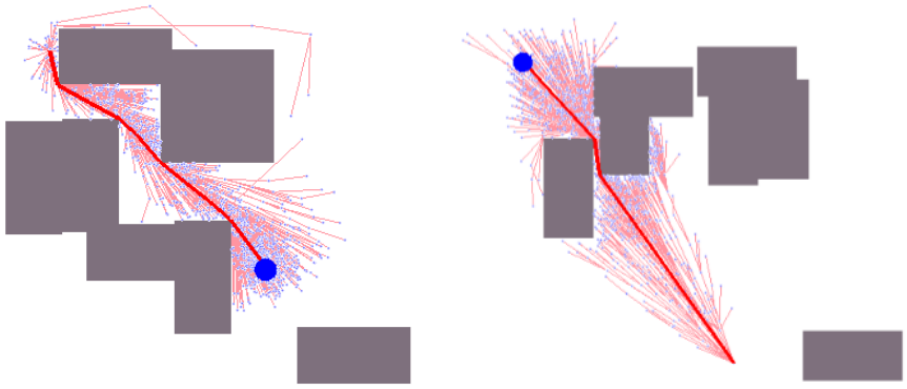


Figure 4.6. C2D Planning Scene

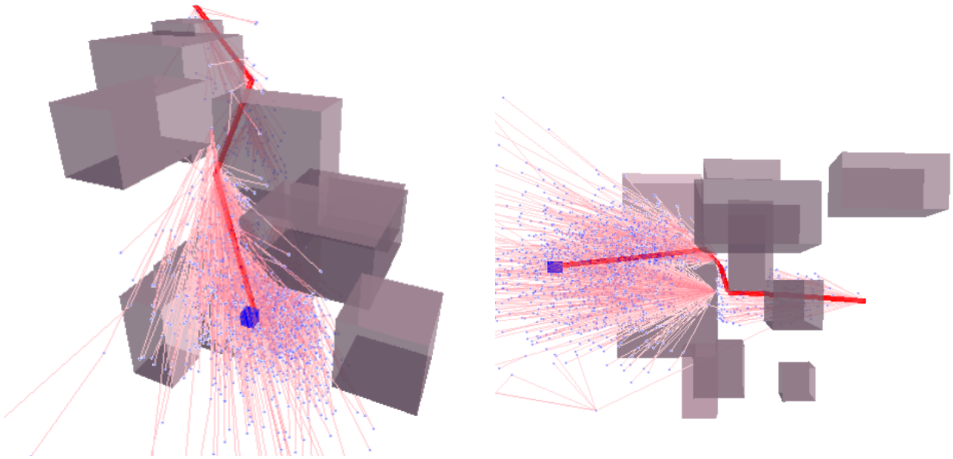


Figure 4.7. C3D Planning Scene

rectangle robot with size 2x5. Due to the shape of the robot, we need to consider its position and orientation during planning. Hence this planning problem has DOF of three. The data generation method is the same as above. Figure 4.8 shows two paths in two SE2 scenes.

In the SE3 environment, we consider a planning scene for a chair robot in a home-like environment with complicated obstacles and narrow passages [97]. As the robot position and orientation need to be considered in the 3D space, the search space is of DOF 7. We use one obstacle configuration, and generate 2696 paths with random start and goal positions. We obtain the point clouds from the OMPL [97] mesh representation, with a shape of 150008x3. Two example paths are shown in Figure 4.9.

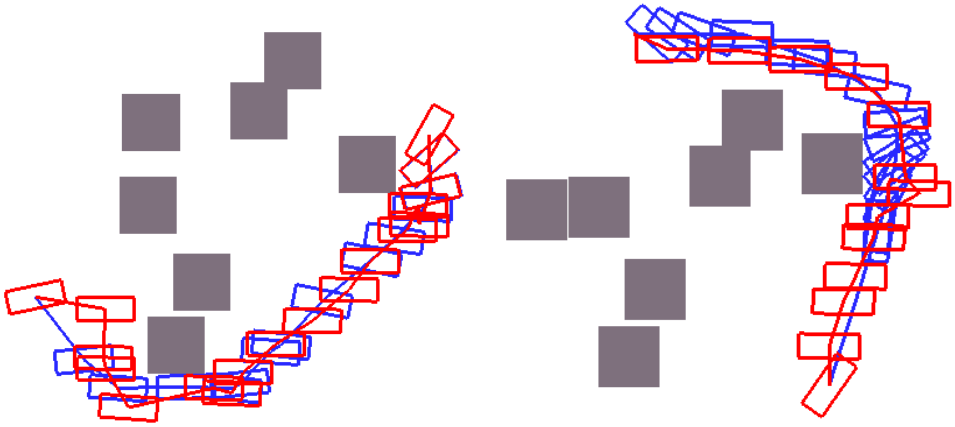


Figure 4.8. R2D Planning Scene

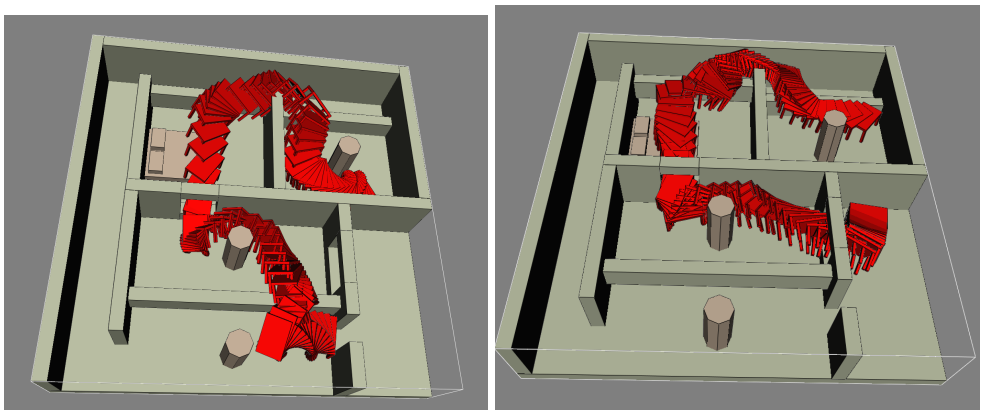


Figure 4.9. R2D Planning Scene

In the manipulation task, we consider planning for a Baxter robot in an environment with two L-shaped tables where five different objects are randomly placed. We consider the reaching task for one arm of the robot. In total, we generate ten different environments, each with 1000 paths generated by classical planners. We obtain the point cloud information by using a KINECT depth camera in the simulated environment. The point clouds are captured by using the ROS packages of PCL [86]. The size of the point cloud is 3x5351. Four real-world scenes for the Baxter environments are shown in Figure 4.10.

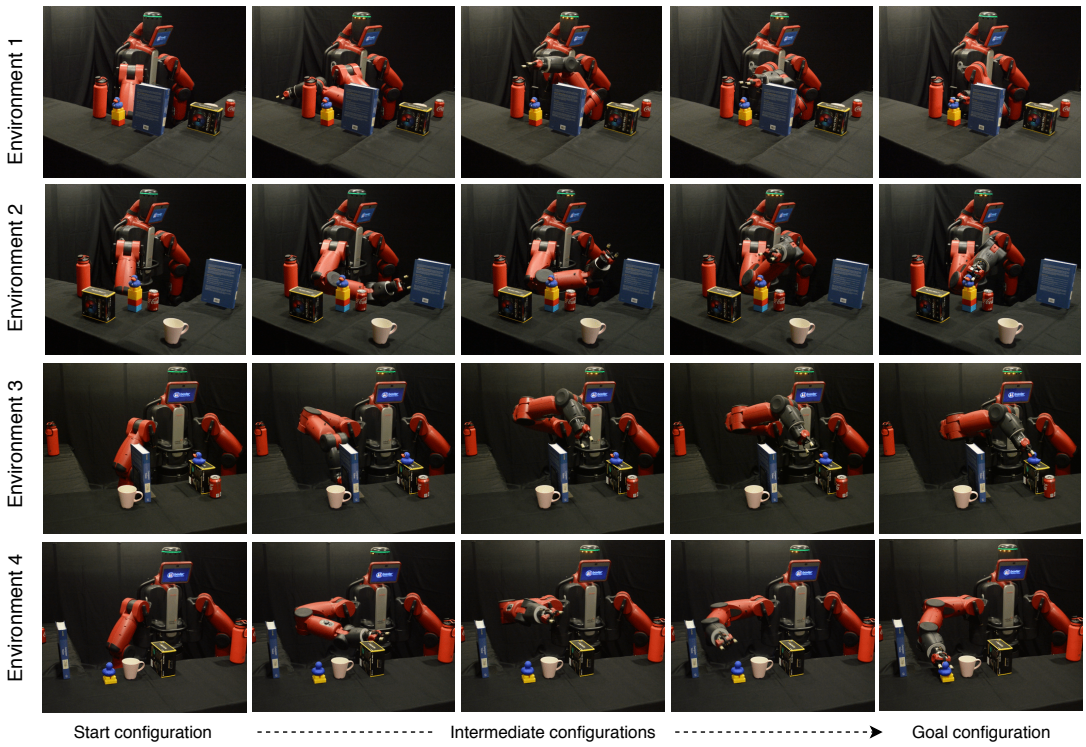


Figure 4.10. Baxter Planning Scene in Real World

4.2.2 Training and Testing

Here we specify details about the training and testing phases. For training, we convert the generated training trajectories into input-target pairs as specified before, and follow the general deep learning training procedure. For the testing phase, we consider two separate cases to evaluate the performance of both the encoder and the planner network. Individually, we consider testing on seen environments where obstacle settings are the same as the training data, and on unseen environments where the obstacles are not present in the training data. We use the seen environments to test the planner network's performance and the unseen environments to test the performance of the encoder and planner networks.

Here we specify the data size for training and testing phases. In the S2D, C2D, C3D, SE2 planning problems, we use 100 environments, each with 4000 paths for training. For the seen testing environments, we use the same environments as in training, each with 200 paths. Additionally, we also use ten unseen environments, each with 2000 paths for testing.

Table 4.1. Performance accuracies of all MPNet variants in the four environments on both test datasets, seen and unseen (shown inside brackets).

Methods	Environments			
	S2D	C2D	C3D	SE2
MPNet	99.30 (98.30)	99.71 (98.80)	99.11 (97.76)	94.21 (95.18)
CMPNet	93.33 (93.18)	83.44 (83.78)	89.88 (90.86)	83.77 (86.18)
ACMPNet	96.70 (97.83)	84.36 (84.08)	96.60 (95.28)	87.08 (87.64)

Table 4.2. Mean computation times with standard deviations are presented for MPNet (all variations), Informed-RRT* and BIT* on test datasets.

Methods	Environments			
	S2D	C2D	C3D	SE2
Informed-RRT*	1.06 ± 0.33 (1.10 ± 0.09)	1.60 ± 0.47 (1.49 ± 0.16)	2.99 ± 0.82 (2.76 ± 0.20)	15.58 ± 2.85 (14.80 ± 2.83)
BIT*	0.59 ± 0.28 (0.65 ± 0.30)	1.79 ± 1.35 (1.61 ± 0.53)	0.19 ± 0.12 (0.20 ± 0.04)	7.16 ± 1.95 (6.52 ± 1.65)
MPNet	0.02 ± 0.00 (0.02 ± 0.00)	0.04 ± 0.00 (0.04 ± 0.01)	0.06 ± 0.01 (0.07 ± 0.01)	0.38 ± 0.04 (0.37 ± 0.02)
CMPNet	0.02 ± 0.00 (0.02 ± 0.00)	0.05 ± 0.01 (0.05 ± 0.01)	0.07 ± 0.01 (0.08 ± 0.01)	0.41 ± 0.08 (0.39 ± 0.07)
ACMPNet	0.03 ± 0.01 (0.03 ± 0.01)	0.06 ± 0.01 (0.06 ± 0.01)	0.08 ± 0.01 (0.08 ± 0.01)	0.53 ± 0.12 (0.42 ± 0.08)

For the other two planning problems, we only consider seen environments for testing. For the SE3 planning problem, we use 2196 paths for training and 500 paths for testing. For the Baxter planning problem, we use ten environments, each with 900 paths for training and 100 paths for testing.

4.2.3 Performance Analysis

Here we show the experimental results of our previously introduced algorithms, specifically the barefoot neural-based motion planner (termed as MPNet), the continual learning motion planner (termed as CMPNet), and the active learning motion planner (termed as ACMPNet). We test MPNet and CMPNet on all problems and ACMPNet only on S2D, C2D, C3D, and SE2 problems.

Meanwhile, we compare these algorithms with classical planners, including RRT*, Informed-RRT*[29], and BIT*[30]. We let Informed-RRT* and BIT* run until the path quality in terms of Euclidean cost is within the range of 5% of the MPNet planned path. We show the comparison of these planners in terms of computational time and path quality.

The comparison of each method is shown in table 4.1 and 4.2. Table 4.3 shows the

Table 4.3. Computation time, path cost/length, and success rate comparison of different methods in Baxter environment.

Methods	Baxter		
	Time	Path cost	Success rate(%)
BIT* (Initial Path)	0.94 ± 0.20	13.91 ± 0.60	83.0
BIT* ($\pm$40% MPNet cost)	9.20 ± 7.61	10.78 ± 0.31	56.0
MPNet	0.59 ± 0.08	7.86 ± 0.20	87.8
CMPNet	0.81 ± 0.08	6.98 ± 0.18	78.6

results for the Baxter planning problem.

For the SE3 planning problem, MPNet achieves a success rate of around 85%. The mean computation times for finding an initial path solution are 0.96s, 1.61s, and 2.84s for MPNet, CMPNet, and BIT*, respectively. Meanwhile, the mean path costs for MPNet, CMPNet, and BIT* are 457.8, 512.73, and 836.85. Furthermore, we notice that BIT* takes up to several minutes to find a path of similar cost as MPNet's solution.

comparison against classical planners

As we observe in the results, neural planners achieve significant speedup compared with classical planners for finding paths with similar optimality. Table 4.2 shows the comparison of computational time in different environments. From the figure, we can observe that the computational speedup is consistent in different obstacle settings. It can also be observed from the table that, as the planning task becomes more complicated with higher dimensions, the speedup becomes more evident.

comparison of neural planners

As shown in the results, MPNet has the best performance compared with the other two algorithms in terms of accuracy and time. However, CMPNet and ACMPNet require only one training epoch in simple problems, and ten epochs in Baxter and SE3 planning problems to achieve acceptable results. On the other hand, MPNet requires 100 training epochs. We also observe that CMPNet and ACMPNet can achieve similar performance if given a similar number of training epochs as MPNet.

Meanwhile, we also observe the sample efficiency of ACMPNet compared with MPNet. We compare the number of paths MPNet and ACMPNet are trained on for each of the environments. ACMPNet achieves a significant reduction of training samples needed to achieve similar performance, as shown in figure 4.11.

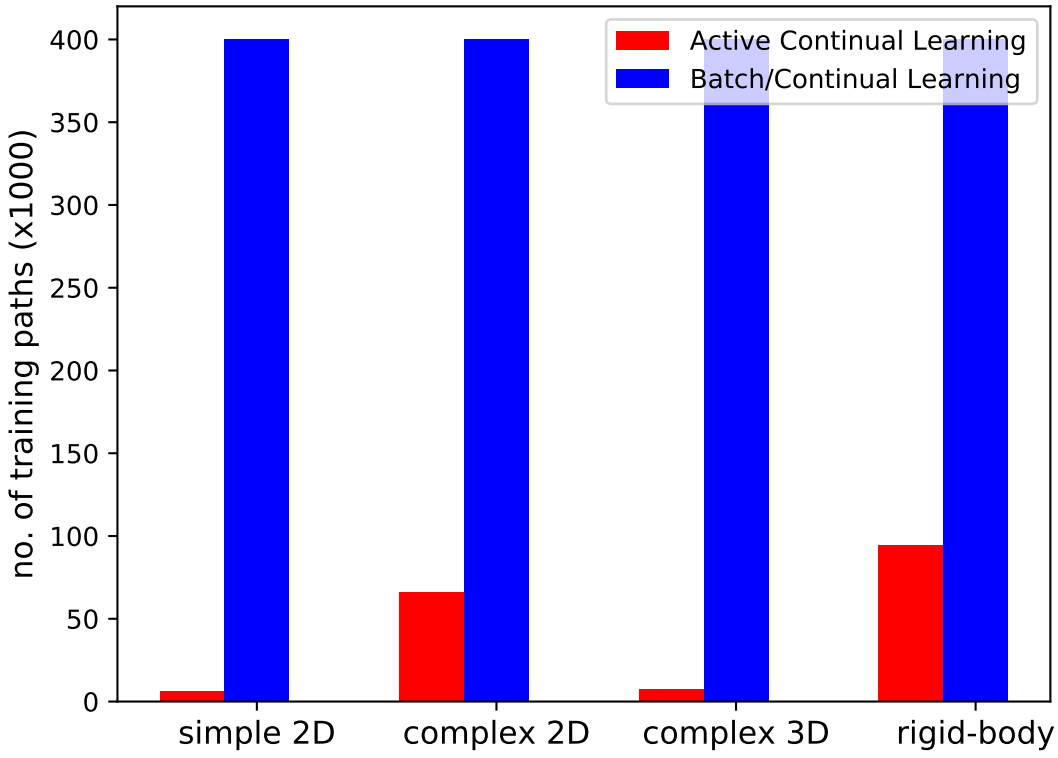


Figure 4.11. Number of training paths required for MPNet/CMPNet vs. ACMPNet

Sample selection in CMPNet and ACMPNet

We also compare different sample selection methods for continual learning. We follow the definitions in [82] and use four methods for selecting samples. Specifically, these are surprise maximization, reward (in our case, it is training loss) maximization, coverage maximization, and global distribution matching. The surprise maximization and reward maximization prefer samples with higher prediction loss or lower prediction loss, respectively. The coverage maximization maintains a memory of k-nearest neighbors. The global distribution matching uses the reservoir sampling technique as discussed before to obtain a uniform sampled subset for representing the

global distribution.

We evaluate these four methods on the S2D dataset and report the results in 4.12. It can be seen that the global distribution matching method obtains the best results, slightly better than the reward maximization method and the coverage maximization method. The explanation of this result might be that the violation of the distribution may lead to deep learning algorithms' inconsistent behaviors.

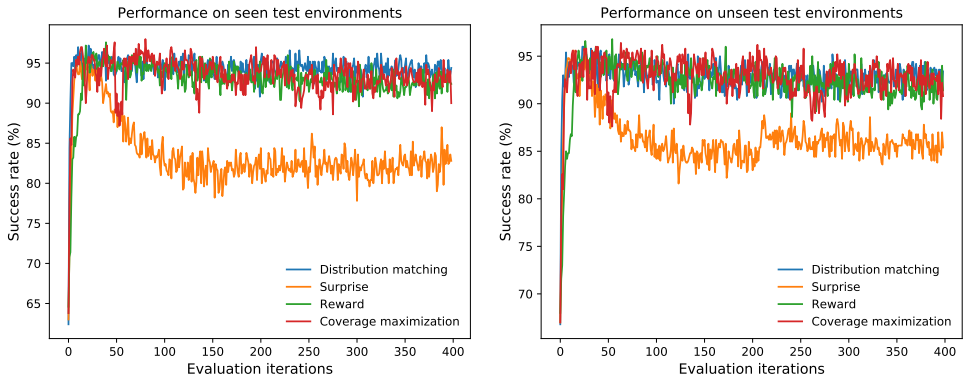


Figure 4.12. Impact of sample selection methods for episode memory on the performance of CMPNet/ACMPNet in S2D

4.2.4 Discussion

Although our algorithms achieve improvement compared with classical planning algorithms, they have several directions for improvement. Firstly, we did not utilize advanced features in deep learning but just representing stochasticity by dropout. A combination with generative methods such as GAN [32] and VAE [23] might better model stochasticity explicitly. Meanwhile, our algorithms are general solutions to motion planning tasks, adding domain knowledge when applying them to specific tasks.

4.3 Learning-Based Kinodynamic Motion Planning

In this section, we present the results of the extension of the motion planning algorithm to kinodynamic planning tasks. Different from the previous problem, the robots need to satisfy

additional kinodynamic motion constraints. We benchmark our algorithm against the state-of-the-art SST (sparseRRT) algorithm [58] in a double-pendulum environment with obstacles. Figure 4.13 shows an example of the planning scene.

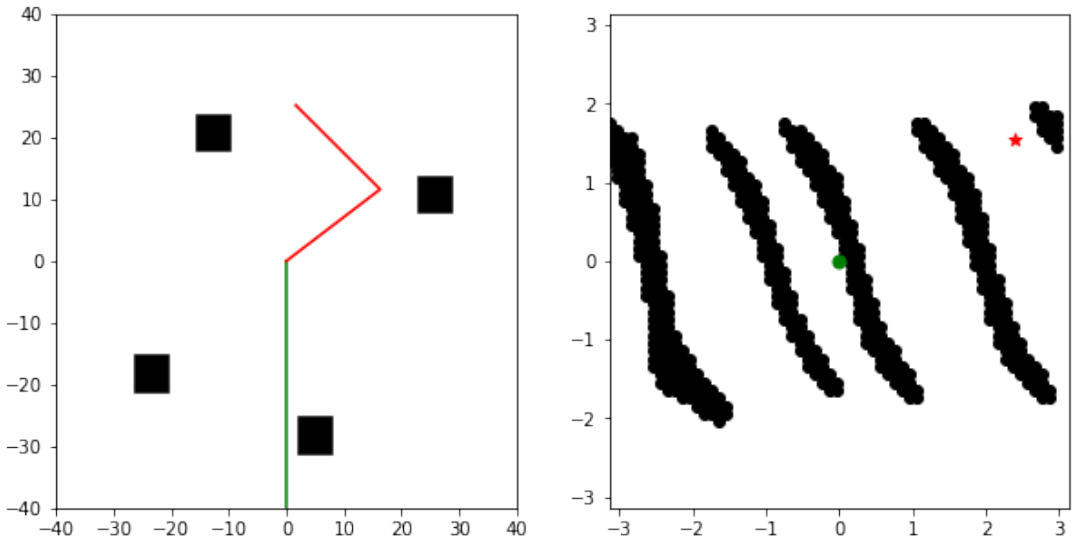


Figure 4.13. Double-Pendulum Problem with Obstacles. Green: start position. Red: goal position.

4.3.1 Environment Setting

In the double-pendulum environment, the state space is of four dimensions, represented by:

$$s = [\theta_0, \theta_1, \dot{\theta}_0, \dot{\theta}_1].$$

For simplicity, we use linear interpolation with fixed time step $dt = 0.02s$ for propagation. The dynamics thus satisfies:

$$s(t+1) = s(t) + f(s(t), u(t))dt.$$

Here $f(s(t), u(t))$ denotes the derivative given the state and control. The detailed equation is given in [95].

The endpoints of the first pole are given by:

$$[0,0],[L\cos(\theta_0)-\frac{\pi}{2},L\sin(\theta_0)-\frac{\pi}{2}].$$

The last endpoint of the second pole is given by:

$$[L(\cos(\theta_0-\frac{\pi}{2})+\cos(\theta_0+\theta_1-\frac{\pi}{2})),L(\sin(\theta_0-\frac{\pi}{2})+\sin(\theta_0+\theta_1-\frac{\pi}{2}))].$$

Here L denotes the length of the pole, which is 20 in our problem setting. The cost in this problem is the time spent on the trajectory.

We make this classical problem more complicated by adding four obstacles, and randomly place the obstacles while making sure the problem is solvable and complicated enough. Meanwhile, we generate start and goal states by fixing the start state to $(0,0,0,0)$ and randomly place the goal states so that the end position is above the height of one pole.

We generate trajectories for training and testing from these randomly generated planning problems using the SST algorithm [58]. After letting the SST algorithm run for 300000 iterations, we can obtain an optimized trajectory suitable for training the MPNet. In total, we generated ten environments, each with 1000 trajectories.

4.3.2 Training and Testing

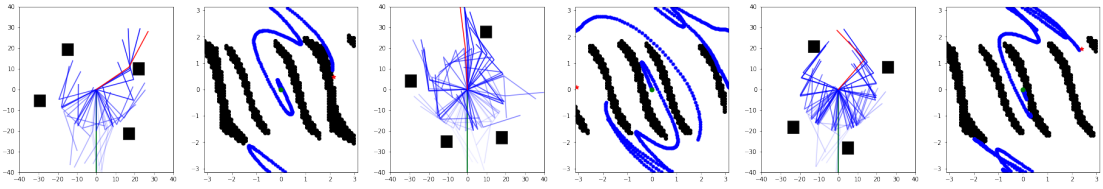
Here we specify details about the training and testing phases. For training, we directly use the waypoints generated by SST and follow the same steps as in the geometric MPNet. The propagation duration between adjacent points ranges from 0.1s to 2s. We use the ten environments, each with 900 trajectories for training. During testing, we use the remaining 100 trajectories in each environment for evaluation.

4.3.3 Performance Analysis

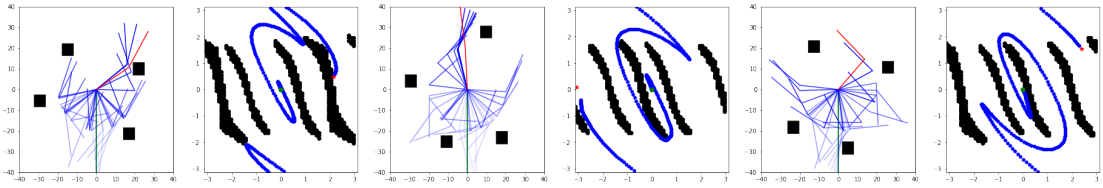
Here we show the experimental results of our previously introduced learning-based kinodynamic planning algorithms. As introduced before, our algorithm can be accelerated by vectorization and implementation using GPU. During experiments, we implemented the CUDA version of the MPC-MPNetTree algorithm, which is faster than the CPU version.

We compare the performances against SST. For comparison of time, we rerun SST on the testing dataset by setting a stopping threshold of cost. If the cost reaches 120% of the data cost, we will terminate SST and use it for comparison. For our learning-based kinodynamic planning algorithms, we will terminate once a trajectory is found. Meanwhile, we consider planning that takes more than 40s as a failure. However, our planning algorithm still results in a high success rate even when we limit the computational time. Specifically, the success rate of MPC-MPNetLine is 90.0%, and the success rate of MPC-MPNetTree is 96.5%.

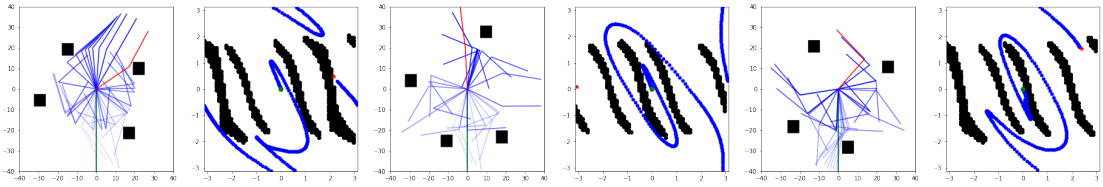
Figure 4.14 shows some sample planned trajectories using different methods. Figure



(a) Sample Planned Trajectories of MPC-MPNetLine



(b) Sample Planned Trajectories of MPC-MPNetTree



(c) Sample Planned Trajectories of SST

Figure 4.14. Sample Planned Trajectories of each Method

4.15 shows the box plots of computational time for each method.

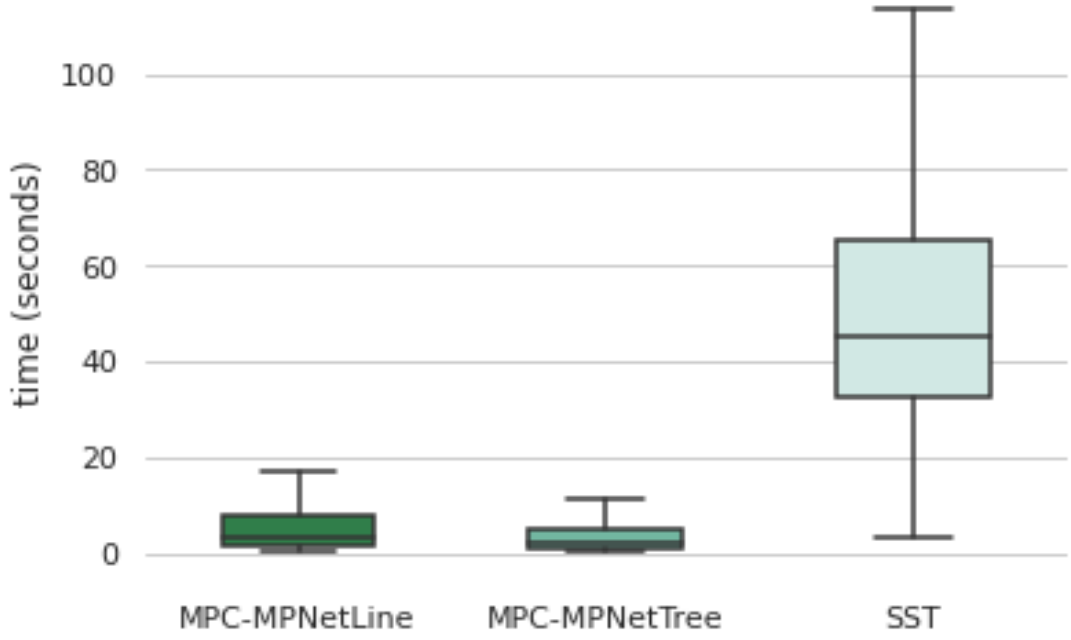


Figure 4.15. Box Plot of Computational Time for Different Kinodynamic Motion Planning Methods.

Meanwhile, to evaluate the optimality of trajectories, we compare the three methods with a baseline of the RRT algorithm. We run the baseline RRT algorithm for a fixed time of 200s. Figure 4.16 shows the box plots of trajectory cost for each method.

As shown in the figures, our learning-based approach can find a more optimized solution than the RRT method with a much shorter computational time. Our learning-based approach is much faster than SST with more than 90% of paths within two times of the SST solution cost.

4.3.4 Discussion

In experiments, we show that learning-based algorithms achieve improvement in computational time for finding paths with 2x cost in kinodynamic motion planning. This result can be further improved with system speedup, such as parallelization and usage of GPU. The cost can also be further optimized by using SST algorithm after the initial solution is found by our

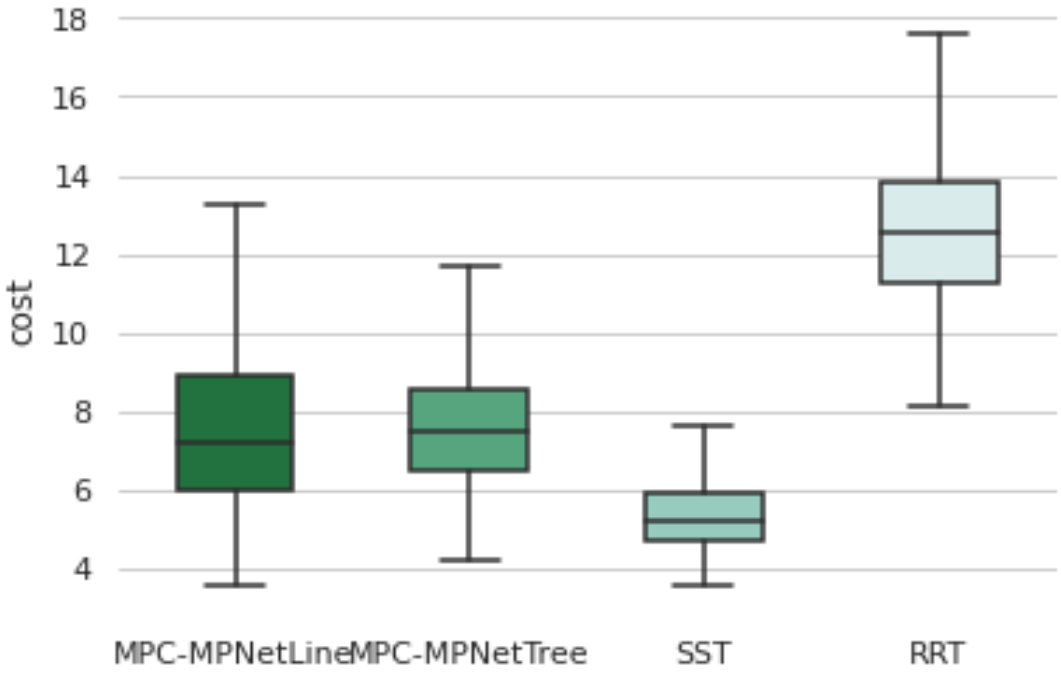


Figure 4.16. Box Plot of Cost for Different Kinodynamic Motion Planning Methods.

algorithm.

4.4 Chapter Acknowledgement

Section 3.2 and Section 3.3 in Chapter 3, and Section 4.2 in Chapter 4, in full, are a reprint of the material with minor modifications as it appears in Motion Planning Networks: Bridging the Gap Between Learning-Based and Classical Motion Planners 2020. Qureshi, Ahmed H.; Miao, Yinglong; Simeonov, Anthony; Yip, Michael C., IEEE Transactions on Robotics, 2020. The thesis author was the co-author of the published paper.

Section 3.4 in Chapter 3, and Section 4.3 in Chapter 4, in full is currently being prepared for submission for publication of the material. Authors include Li, Linjun; Miao, Yinglong; Qureshi, Ahmed H.; Yip, Michael C.. The thesis author was the shared first author of this material.

Chapter 5

Conclusion and Future Work

Traditional planning algorithms achieve theoretical optimality guarantees and also achieve decent results in practical problems. However, for complicated tasks, these algorithms require much computational time, which is not suitable for real-time planning in robotics.

By utilizing recent advances in deep learning, past planning instances can be better utilized to accelerate the planning pipeline. Specifically, by learning from past successful trajectories, neural networks can learn a sampling distribution where samples are near the solution of good-quality paths. This experience can be generalized to problem instances with similar environment settings.

In this thesis, we show out motion planning algorithms using neural samplers in simple geometric tasks and complicated kinodynamic motion planning tasks. We show the success of this method by benchmarking with traditional motion planners. Specifically, it achieves a high success rate while requiring much less computational time. Thus, in practice, we can benefit by learning from planned instances and using neural samplers to accelerate the planning pipeline. Meanwhile, we offer an active learning pipeline to combine the merits of traditional motion planners and learning-based motion planners. Specifically, we only use traditional motion planners when learning-based motion planners fail. This method results in a system that can dynamically improve and achieves life-long learning.

At the same time, we also investigated another essential component in the motion planning

pipeline, collision checking. We show how this component can also be accelerated by deep learning. We showed a high success rate with low computational time without preprocessing the point cloud inputs.

By combining both components, future directions can work on building an end-to-end motion planning pipeline. This direction may also generalize to partially-observable planning problems and dynamic environments.

It would also be interesting to see how deep learning can help with more practical and complicated tasks such as hierarchical planning in manipulation and locomotion problems. Future projects can work on the real deployment of these learning-based motion planning methods.

Meanwhile, more compact representation of the trajectory will also be beneficial. The combination with generative neural networks will be an interesting direction.

Furthermore, our approach belongs to imitation learning, where expert demonstrations are required for training. Another direction is self-supervised methods such as reinforcement learning. These methods can learn without demonstrations but might require more time to converge. A hybrid method might be another direction to combine the merits of both fields.

Bibliography

[1] Anders Adamson and Marc Alexa. Approximating and intersecting surfaces from points. In *Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, pages 230–239, 2003.

[2] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein gan. *arXiv preprint arXiv:1701.07875*, 2017.

[3] Karl J Astrom. Optimal control of markov processes with incomplete state information. *Journal of mathematical analysis and applications*, 10(1):174–205, 1965.

[4] Mukund Balasubramanian, Eric L Schwartz, Joshua B Tenenbaum, Vin de Silva, and John C Langford. The isomap algorithm and topological stability. *Science*, 295(5552):7–7, 2002.

[5] Jonathan Baxter. A model of inductive bias learning. *Journal of artificial intelligence research*, 12:149–198, 2000.

[6] Richard Bellman. A markovian decision process. *Journal of mathematics and mechanics*, pages 679–684, 1957.

[7] Dmitry Berenson, Pieter Abbeel, and Ken Goldberg. A robot path planning framework that learns from experience. In *2012 IEEE International Conference on Robotics and Automation*, pages 3671–3678. IEEE, 2012.

[8] Dimitri P Bertsekas, Dimitri P Bertsekas, Dimitri P Bertsekas, and Dimitri P Bertsekas. *Dynamic programming and optimal control*, volume 1. Athena scientific Belmont, MA, 1995.

[9] Geoff Boeing. Visual analysis of nonlinear dynamical systems: chaos, fractals, self-similarity and the limits of prediction. *Systems*, 4(4):37, 2016.

[10] Zdravko I Botev, Dirk P Kroese, Reuven Y Rubinstein, and Pierre L'Ecuyer. The cross-entropy method for optimization. In *Handbook of statistics*, volume 31, pages 35–59. Elsevier, 2013.

[11] Andrew Carlson, Justin Betteridge, Bryan Kisiel, Burr Settles, Estevam R Hruschka, and Tom M Mitchell. Toward an architecture for never-ending language learning. In *Twenty-Fourth AAAI conference on artificial intelligence*, 2010.

[12] Ishani Chatterjee, Maxim Likhachev, Ashwin Khadke, and Manuela Veloso. Speeding up search-based motion planning via conservative heuristics. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 29, pages 674–679, 2019.

[13] Peng Cheng, Emilio Frazzoli, and Steven LaValle. Improving the performance of sampling-based motion planning with symmetry-based gap reduction. *IEEE Transactions on Robotics*, 24(2):488–494, 2008.

[14] Shiing-Shen Chern, Wei-huan Chen, and Kai Shue Lam. *Lectures on differential geometry*, volume 1. World Scientific Publishing Company, 1999.

[15] Sonia Chernova and Manuela Veloso. Interactive policy learning through confidence-based autonomy. *Journal of Artificial Intelligence Research*, 34:1–25, 2009.

[16] Hao-Tien Lewis Chiang, Jasmine Hsu, Marek Fiser, Lydia Tapia, and Aleksandra Faust. Rl-rrt: Kinodynamic motion planning via learning reachability estimators from rl policies. *IEEE Robotics and Automation Letters*, 4(4):4298–4305, 2019.

[17] F Chin and Cao An Wang. Optimal algorithms for the intersection and the minimum distance problems between planar polygons. *IEEE Transactions on Computers*, 32(12):1203–1207, 1983.

[18] Sumit Chopra, Raia Hadsell, and Yann LeCun. Learning a similarity metric discriminatively, with application to face verification. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 1, pages 539–546. IEEE, 2005.

[19] Jonathan D Cohen, Ming C Lin, Dinesh Manocha, and Madhav Ponamgi. I-collide: An interactive and exact collision detection system for large-scale environments. In *Proceedings of the 1995 symposium on Interactive 3D graphics*, pages 189–ff, 1995.

[20] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[21] Nikhil Das, Naman Gupta, and Michael Yip. Fastron: An online learning-based model and active learning strategy for proxy collision detection. *arXiv preprint arXiv:1709.02316*, 2017.

[22] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[23] Carl Doersch. Tutorial on variational autoencoders. *arXiv preprint arXiv:1606.05908*, 2016.

[24] Bruce Donald, Patrick Xavier, John Canny, and John Reif. Kinodynamic motion planning. *Journal of the ACM (JACM)*, 40(5):1048–1066, 1993.

[25] Hugh Durrant-Whyte and Tim Bailey. Simultaneous localization and mapping: part i. *IEEE robotics & automation magazine*, 13(2):99–110, 2006.

[26] Eugene A Feinberg and Adam Shwartz. *Handbook of Markov decision processes: methods and applications*, volume 40. Springer Science & Business Media, 2012.

[27] Mauro Figueiredo, João Oliveira, Bruno Araújo, and João Pereira. An efficient collision detection algorithm for point cloud models.

[28] Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*, pages 1050–1059, 2016.

[29] Jonathan D Gammell, Siddhartha S Srinivasa, and Timothy D Barfoot. Informed rrt*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2997–3004. IEEE, 2014.

[30] Jonathan D Gammell, Siddhartha S Srinivasa, and Timothy D Barfoot. Batch informed trees (bit*): Sampling-based optimal planning via the heuristically guided search of implicit random geometric graphs. In *2015 IEEE international conference on robotics and automation (ICRA)*, pages 3067–3074. IEEE, 2015.

[31] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[32] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[33] Ian J Goodfellow, Mehdi Mirza, Da Xiao, Aaron Courville, and Yoshua Bengio. An empirical investigation of catastrophic forgetting in gradient-based neural networks. *arXiv preprint arXiv:1312.6211*, 2013.

[34] Wassim M Haddad and VijaySekhar Chellaboina. *Nonlinear dynamical systems and control: a Lyapunov-based approach*. Princeton university press, 2011.

[35] Bin He, Shuai Wang, and Yongjia Liu. Underactuated robotics: a review. *International Journal of Advanced Robotic Systems*, 16(4):1729881419862164, 2019.

[36] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[37] Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning. In *Advances in neural information processing systems*, pages 4565–4573, 2016.

[38] Zhiao Huang, Fangchen Liu, and Hao Su. Mapping state space using landmarks for universal goal reaching. In *Advances in Neural Information Processing Systems*, pages 1942–1952, 2019.

[39] Piotr Indyk, Jirı Matoušek, and Anastasios Sidiropoulos. Low-distortion embeddings of finite metric spaces. *Handbook of discrete and computational geometry*, 37:46, 2004.

[40] David Isele and Akansel Cosgun. Selective experience replay for lifelong learning. In *Thirty-second AAAI conference on artificial intelligence*, 2018.

[41] Fahad Islam, Venkatraman Narayanan, and Maxim Likhachev. A*-connect: Bounded suboptimal bidirectional heuristic search. In *2016 IEEE International Conference On Robotics and Automation (ICRA)*, pages 2752–2758. IEEE, 2016.

[42] Lucas Janson, Edward Schmerling, Ashley Clark, and Marco Pavone. Fast marching tree: A fast marching sampling-based method for optimal motion planning in many dimensions. *The International journal of robotics research*, 34(7):883–921, 2015.

[43] Maximilian Jaritz, Jiayuan Gu, and Hao Su. Multi-view pointnet for 3d scene understanding. In *Proceedings of the IEEE International Conference on Computer Vision Workshops*, pages 0–0, 2019.

[44] Bolesław Kacewicz. Complexity of nonlinear two-point boundary-value problems. *Journal of Complexity*, 18(3):702–738, 2002.

[45] Sertac Karaman and Emilio Frazzoli. Sampling-based algorithms for optimal motion planning. *The international journal of robotics research*, 30(7):846–894, 2011.

[46] Kenji Kawaguchi and Jiaoyang Huang. Gradient descent finds global minima for generalizable deep neural networks of practical sizes. In *2019 57th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 92–99. IEEE, 2019.

[47] Mahmut Kaya and Hasan Şakir Bilge. Deep metric learning: A survey. *Symmetry*, 11(9):1066, 2019.

[48] S Mohammad Khansari-Zadeh and Aude Billard. Learning stable nonlinear dynamical systems with gaussian mixture models. *IEEE Transactions on Robotics*, 27(5):943–957, 2011.

[49] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[50] Donald E Kirk. *Optimal control theory: an introduction*. Courier Corporation, 2004.

[51] Jan Klein and Gabriel Zachmann. Point cloud collision detection. In *Computer Graphics Forum*, volume 23, pages 567–576. Wiley Online Library, 2004.

[52] Jan Klein and Gabriel Zachmann. Interpolation search for point cloud intersection. 2005.

[53] Tobias Kunz, Andrea Thomaz, and Henrik Christensen. Hierarchical rejection sampling for informed kinodynamic planning in high-dimensional spaces. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 89–96. IEEE, 2016.

[54] Olivier Lablée. *Spectral theory in Riemannian geometry*. 2015.

[55] Florent Lamiraux, Etienne Ferré, and Erwan Vallée. Kinodynamic motion planning: Connecting exploration trees using trajectory optimization methods. In *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA'04. 2004*, volume 4, pages 3987–3992. IEEE, 2004.

[56] Steven M LaValle. *Planning algorithms*. Cambridge university press, 2006.

[57] Frank L Lewis, Draguna Vrabie, and Vassilis L Syrmos. *Optimal control*. John Wiley & Sons, 2012.

[58] Yanbo Li, Zakary Littlefield, and Kostas E Bekris. Sparse methods for efficient asymptotically optimal kinodynamic planning. In *Algorithmic foundations of robotics XI*, pages 263–282. Springer, 2015.

[59] Jiahao Lin and Gim Hee Lee. Trajectory space factorization for deep video-based 3d human pose estimation. *arXiv preprint arXiv:1908.08289*, 2019.

[60] Ming Lin and Stefan Gottschalk. Collision detection between geometric models: A survey. In *Proc. of IMA conference on mathematics of surfaces*, volume 1, pages 602–608, 1998.

[61] Zakary Littlefield and Kostas E Bekris. Efficient and asymptotically optimal kinodynamic motion planning via dominance-informed regions. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1–9. IEEE, 2018.

[62] Fangchen Liu, Zhan Ling, Tongzhou Mu, and Hao Su. State alignment-based imitation learning. *arXiv preprint arXiv:1911.10947*, 2019.

[63] David Lopez-Paz and Marc'Aurelio Ranzato. Gradient episodic memory for continual learning. In *Advances in neural information processing systems*, pages 6467–6476, 2017.

[64] Olvi L Mangasarian. Sufficient conditions for the optimal control of nonlinear systems. *SIAM Journal on control*, 4(1):139–152, 1966.

[65] David M Mount. Geometric intersection. In *Handbook of Discrete and Computational Geometry, chapter 33*. Citeseer, 1997.

[66] Ryuie Nishii. Maximum likelihood principle and model selection when the true model is unspecified. *Journal of Multivariate analysis*, 27(2):392–403, 1988.

[67] Takayuki Osa, Joni Pajarinen, Gerhard Neumann, J Andrew Bagnell, Pieter Abbeel, and Jan Peters. An algorithmic perspective on imitation learning. *arXiv preprint arXiv:1811.06711*, 2018.

[68] Brian Paden, Valerio Varricchio, and Emilio Frazzoli. Verification and synthesis of admissible heuristics for kinodynamic motion planning. *IEEE Robotics and Automation Letters*, 2(2):648–655, 2017.

[69] Jia Pan, Sachin Chitta, and Dinesh Manocha. Probabilistic collision detection between noisy point clouds using robust classification. In *Robotics Research*, pages 77–94. Springer, 2017.

[70] Alexandros Paraschos, Christian Daniel, Jan R Peters, and Gerhard Neumann. Probabilistic movement primitives. In *Advances in neural information processing systems*, pages 2616–2624, 2013.

[71] Xue Bin Peng, Glen Berseth, KangKang Yin, and Michiel Van De Panne. Deeploco: Dynamic locomotion skills using hierarchical deep reinforcement learning. *ACM Transactions on Graphics (TOG)*, 36(4):1–13, 2017.

[72] Anastasia Pentina and Ruth Urner. Lifelong learning with weighted majority votes. In *Advances in Neural Information Processing Systems*, pages 3612–3620, 2016.

[73] Yunchen Pu, Zhe Gan, Ricardo Henao, Xin Yuan, Chunyuan Li, Andrew Stevens, and Lawrence Carin. Variational autoencoder for deep learning of images, labels and captions. In *Advances in neural information processing systems*, pages 2352–2360, 2016.

[74] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[75] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 652–660, 2017.

[76] Ahmed H Qureshi and Michael C Yip. Deeply informed neural sampling for robot motion planning. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 6582–6588. IEEE, 2018.

[77] Ahmed Hussain Qureshi and Yasar Ayaz. Intelligent bidirectional rapidly-exploring random trees for optimal motion planning in complex cluttered environments. *Robotics and Autonomous Systems*, 68:1–11, 2015.

[78] Ahmed Hussain Qureshi, Yinglong Miao, Anthony Simeonov, and Michael C Yip. Motion planning networks: Bridging the gap between learning-based and classical motion planners. *IEEE Transactions on Robotics*, 2020.

[79] Aravind Rajeswaran, Vikash Kumar, Abhishek Gupta, Giulia Vezzani, John Schulman, Emanuel Todorov, and Sergey Levine. Learning complex dexterous manipulation with deep reinforcement learning and demonstrations. *arXiv preprint arXiv:1709.10087*, 2017.

[80] John H Reif. Complexity of the generalized mover's problem. Technical report, HARVARD UNIV CAMBRIDGE MA AIKEN COMPUTATION LAB, 1985.

[81] Mark Bishop Ring. *Continual learning in reinforcement environments*. PhD thesis, University of Texas at Austin Austin, Texas 78712, 1994.

[82] David Rolnick, Arun Ahuja, Jonathan Schwarz, Timothy Lillicrap, and Gregory Wayne. Experience replay for continual learning. In *Advances in Neural Information Processing Systems*, pages 350–360, 2019.

[83] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[84] Stephane Ross and J Andrew Bagnell. Reinforcement and imitation learning via interactive no-regret learning. *arXiv preprint arXiv:1406.5979*, 2014.

[85] Stuart Russell and Peter Norvig. Artificial intelligence: a modern approach. 2002.

[86] Radu Bogdan Rusu and Steve Cousins. 3d is here: Point cloud library (pcl). In *2011 IEEE international conference on robotics and automation*, pages 1–4. IEEE, 2011.

[87] Paul Ruvolo and Eric Eaton. Ella: An efficient lifelong learning algorithm. In *International Conference on Machine Learning*, pages 507–515, 2013.

[88] Basak Sakcak, Luca Bascetta, Gianni Ferretti, and Maria Prandini. Sampling-based optimal kinodynamic planning with motion primitives. *Autonomous Robots*, 43(7):1715–1732, 2019.

[89] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[90] Nikolaos Sarafianos, Bogdan Boteanu, Bogdan Ionescu, and Ioannis A Kakadiaris. 3d human pose estimation: A review of the literature and analysis of covariates. *Computer Vision and Image Understanding*, 152:1–20, 2016.

[91] Stefan Schaal, Jan Peters, Jun Nakanishi, and Auke Ijspeert. Learning movement primitives. In *Robotics research. the eleventh international symposium*, pages 561–572. Springer, 2005.

[92] Jacob T Schwartz and Micha Sharir. On the "piano movers" problem. ii. general techniques for computing topological properties of real algebraic manifolds. *Advances in applied Mathematics*, 4(3):298–351, 1983.

[93] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[94] Aravind Sivaramakrishnan, Zakary Littlefield, and Kostas E Bekris. Towards learning efficient maneuver sets for kinodynamic motion planning. *arXiv preprint arXiv:1907.07876*, 2019.

[95] Mark W Spong. Underactuated mechanical systems. In *Control problems in robotics and automation*, pages 135–150. Springer, 1998.

[96] SS Srinivasa, Timothy D Barfoot, and JD Gammell. Batch informed trees (bit*): Informed asymptotically optimal anytime search. *The International Journal of Robotics Research*, 39(5), 2020.

[97] Ioan A Sucan, Mark Moll, and Lydia E Kavraki. The open motion planning library. *IEEE Robotics & Automation Magazine*, 19(4):72–82, 2012.

[98] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[99] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

[100] Russ Tedrake, Ian R Manchester, Mark Tobenkin, and John W Roberts. Lqr-trees: Feedback motion planning via sums-of-squares verification. *The International Journal of Robotics Research*, 29(8):1038–1052, 2010.

[101] Sebastian Thrun. A lifelong learning perspective for mobile robot control. In *Intelligent robots and systems*, pages 201–214. Elsevier, 1995.

[102] Greg Turk and Brendan Mullins. Large geometric models archive, 2003.

[103] René Weller. A brief overview of collision detection. In *New Geometric Data Structures for Collision Detection and Haptics*, pages 9–46. Springer, 2013.

[104] Svante Wold, Kim Esbensen, and Paul Geladi. Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52, 1987.

[105] Muiris Woulfe, Michael Manzke, and John Laizoliana Dingliana. Hardware accelerated broad phase collision detection for realtime simulations. 2007.

[106] Yueh-Hua Wu, Ting-Han Fan, Peter J Ramadge, and Hao Su. Model imitation for model-based reinforcement learning. *arXiv preprint arXiv:1909.11821*, 2019.

[107] Christopher Xie, Jur van den Berg, Sachin Patil, and Pieter Abbeel. Toward asymptotically optimal motion planning for kinodynamic systems using a two-point boundary value problem solver. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4187–4194. IEEE, 2015.

[108] Daqing Yi, Rohan Thakker, Cole Gulino, Oren Salzman, and Siddhartha Srinivasa. Generalizing informed sampling for asymptotically-optimal sampling-based kinodynamic planning via markov chain monte carlo. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7063–7070. IEEE, 2018.

[109] Gabriel Zachmann. Optimizing the collision detection pipeline. In *Proc. of the First International Game Technology Conference (GTEC)*, volume 2, 2001.

[110] Han Zhang, Ian Goodfellow, Dimitris Metaxas, and Augustus Odena. Self-attention generative adversarial networks. In *International Conference on Machine Learning*, pages 7354–7363, 2019.

[111] Yin Zhou and Oncel Tuzel. Voxelnet: End-to-end learning for point cloud based 3d object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4490–4499, 2018.