Motion Planning Networks

Preprint · June 2018

CITATIONS
0

READS
486

3 authors:

Ahmed H. Qureshi
Purdue University
64 PUBLICATIONS **1,465** CITATIONS
SEE PROFILE

Mayur Bency
University of California, San Diego
5 PUBLICATIONS **286** CITATIONS
SEE PROFILE

Michael C. Yip
University of California, San Diego
152 PUBLICATIONS **3,229** CITATIONS
SEE PROFILE

Motion Planning Networks

Ahmed H. Qureshi, Mayur J. Bency and Michael C. Yip
Department of Electrical and Computer Engineering
University of California San Diego
9500 Gilman Dr, La Jolla, CA 92093
Email: {a1qureshi, mbency, yip}@ucsd.edu

***Abstract*—Fast and efficient motion planning algorithms are crucial for many state-of-the-art robotics applications such as self-driving cars. Existing motion planning methods such as RRT*, A*, and D*, become ineffective as their computational complexity increases exponentially with the dimensionality of the motion planning problem. To address this issue, we present a neural network-based novel planning algorithm which generates end-to-end collision-free paths irrespective of the obstacles' geometry. The proposed method, called MPNet (Motion Planning Network), comprises of a Contractive Autoencoder which encodes the given workspaces directly from a point cloud measurement, and a deep feedforward neural network which takes the workspace encoding, start and goal configuration, and generates end-to-end feasible motion trajectories for the robot to follow. We evaluate MPNet on multiple planning problems such as planning of a point-mass robot, rigid-body, and 7 DOF Baxter robot manipulators in various 2D and 3D environments. The results show that MPNet is not only consistently computationally efficient in all 2D and 3D environments but also show remarkable generalization to completely unseen environments. The results also show that computation time of MPNet consistently remains less than 1 second which is significantly lower than existing state-of-the-art motion planning algorithms. Furthermore, through transfer learning, the MPNet trained in one scenario (e.g., indoor living places) can also quickly adapt to new scenarios (e.g., factory floors) with a little amount of data.**

I. Introduction

Robotic motion planning aims to compute a collision-free path for a robot given initial and goal configurations [14]. Algorithms which solve the motion planning problem have applications spanning fields of autonomous driving [16], robotic surgery [26], aerial robotics [4], underwater robotics [27], humanoid robotics [3], and even space exploration [24]. As motion planning algorithms are necessary for solving a variety of complicated, high-dimensional problems, there arises a critical, unmet need for computationally tractable, real-time algorithms.

The quest for developing fast, computationally efficient methods has led to the development of various sampling-based planning algorithms such as Rapidly-exploring Random Trees (RRT) [13], optimal Rapidly-exploring Random Trees (RRT*) [10], and Potentially guided-RRT* (P-RRT*) [18]. Despite previous efforts to design fast, efficient planning algorithms, the current state-of-the-art struggles to offer methods which scale to the high-dimensional setting that is common in many real-world applications. In this paper, we propose a Deep Neural Network (DNN) based iterative motion planning algorithm, called MPNet (Motion Planning Networks).

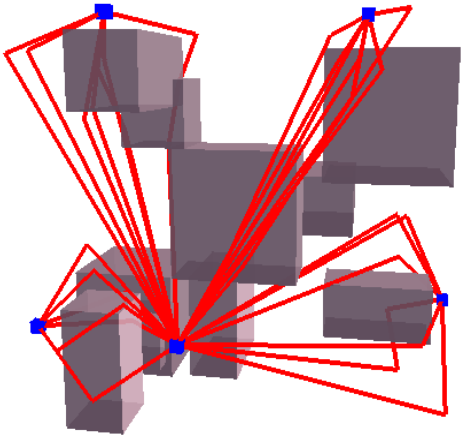


Fig. 1: MPNet planning motions in a 3D cluttered environment for a fixed start and multiple goal configurations (presented as blue boxes). For each start and goal pair, MPNet can generate multiple collision-free paths (shown in red) in a finite time.

MPNet consists of two components: an obstacle-space encoder and a path generator. We use a Contractive Autoencoder [19] to encode a point cloud of the obstacles into a latent space. The path generator is a deep feed-forward neural network which is trained to predict the robot configuration at time step $t+1$ given the robot configuration at time t, goal configuration, and the latent-space encoding of the obstacle space. Once trained, MPNet can be used in conjunction with our novel bi-directional iterative algorithm to generate feasible trajectories.

We validate our model by testing it on a large dataset of complex, cluttered 2D and 3D environments. To highlight our algorithm's computational robustness, we also test our proposed algorithm on the rigid body motion planning problem. As neural networks do not provide theoretical guarantees on their performance, we also propose a hybrid algorithm which combines MPNet with any existing classical planning algorithm, in our case RRT*, which demonstrate a 100% success rate consistently over all tested environments while retaining the computational gains.

Our results indicate that MPNet generalizes very well, not only to unseen start and goal configurations within workspaces which were used in training, but also to new workspaces which the algorithm has never seen. Furthermore, we show that

MPNet can utilize transfer learning to adapt a model trained on one type of environment to an entirely different environment with a small amount of data. The code for MPNet is available online including the training and testing dataset[1] with the hope that our methods will be used as a testbed for future research into this problem.

II. Related Work

Research into developing Neural Network-based motion planners first gained traction in the early 1990s. However, due to the lack of available data as well as the complexity in training DNNs, interest in this field faded during the early 2000s [20]. However, recent developments in Deep Learning (DL) have allowed researchers to apply various DL architectures to robotic control and planning.

Glasius et al. [7] proposed a Hopfield network based method for path generation in dynamic environments. However, this method required the robot to have faster dynamics than the obstacles. The method in [25] extends [7] by removing this limiting restriction. Inspired by biological neurons [9], the algorithm [25] exploits a shutting model [8] to formulate neural network for navigation in dynamic environments. However, the above-mentioned methods do not scale well to high-dimensional problems since, like grid-based planning methods, they can only operate on discretized environments. And, it has already been proven that grid-based methods such as A* and its variants [12] do not scale well to high-dimensional problems [10].

Another active area of research withing robotic control and planning is Deep Reinforcement Learning (RL). For instance, [15] shows how to train a robot to learn visuomotor policies to perform various tasks such as screwing a bottle cap, or inserting a peg. Although RL is a promising framework, it extensively relies on the exploration thus makes it difficult to train for many real-world robotic applications. A recent work, Value Iteration Networks (VIN) [22] emulate value iteration by leveraging recurrent convolutional neural networks and max-pooling. However, in addition to limitations inherited from underlying RL framework, VIN dependency on convolutional neural networks limits its application to 2D, fixed grid-size mazes only.

A recent and relevant method is the Lightning Framework [1], which is composed of two modules. The first module performs path planning from scratch using traditional motion planning methods. The second module maintains a lookup table which caches old paths generated by the first module. For new planning problems, the Lightning Framework retrieves the closest path, in term of start and goal positions, from a lookup table and repairs it using a traditional motion planner from the first module. This approach demonstrates superior performance in high-dimensional spaces when compared to conventional planning methods. However, not only are lookup tables memory inefficient, they also are incapable of generalizing to new environments where the location of obstacles are different from the examples stored in the lookup table.

[1]The supplementary material including project videos are available at https://sites.google.com/view/mpnet/home.

III. Problem Definition

This section describes the notations used in this paper and formally defines the motion planning problems addressed by the proposed method.

Let Q be an ordered list of length $N \in \mathbb{N}$, then a sequence $\{q_i = Q(i)\}_{i \in N}$ is a mapping from $i \in \mathbb{N}$ to the i-th element of Q. Moreover, for the algorithms described in this paper, $Q(\text{end})$ and $Q.\text{length}()$ give the last element and the number of elements in a set Q, respectively. Let $X \subset \mathbb{R}^d$ be a given state space, where $d \in \mathbb{N}_{\geq 2}$ is the dimensionality of the state space. The obstacle and obstacle-free state spaces are defined as $X_{\text{obs}} \subset X$ and $X_{\text{free}} = X \backslash X_{\text{obs}}$, respectively. Let the initial state be $x_{\text{init}} \in X_{\text{free}}$, and goal region be $X_{\text{goal}} \subset X_{\text{free}}$. Let an ordered list τ be a path having non-negative and non-zero scalar length. A solution path τ to the motion planning problem is feasible if it connects x_{init} and $x \in X_{\text{goal}}$, i.e. $\tau(0) = x_{\text{init}}$ and $\tau(\text{end}) \in X_{\text{goal}}$, and lies entirely in the obstacle-free space X_{free}.

The proposed work addresses the feasibility problem of motion planning i.e.,

Problem (Feasible Motion Planning): *Given a triplet* $\{X, X_{\text{free}}, X_{\text{obs}}\}$*, an initial state* x_{init} *and a goal region* $X_{\text{goal}} \subset X_{\text{free}}$*, find a path solution* $\tau \in X_{\text{free}}$ *such that* $\tau(0) = x_{\text{init}}$ *and* $\tau(\text{end}) \in X_{\text{goal}}$.

IV. MPNet: A Neural Motion Planner

This section introduces our proposed model, MPNet (see Fig. 2). MPNet is a neural network based motion planner comprised of two phases. The first phase corresponds to the offline training of the neural models. The second corresponds to the online path generation.

A. Offline Training

Our proposed method uses two neural models to solve the motion planning problem. The first model is a Contractive AutoEncoder (CAE) [19] which embeds the obstacle point cloud, corresponding to a point cloud representing X_{obs}, into a latent space (see Fig. 2(a)). The second model is a feed-forward deep neural network which learns to do motion planning for the given obstacle embedding from the CAE, as well as the start and goal configurations (see Fig. 2(b)).

1) Contractive AutoEncoder: A Contractive AutoEncoder is used to embed obstacles point cloud into an invariant and robust feature space $Z \in \mathbb{R}^m$, where $m \in \mathbb{N}$ is the dimensionality of the feature space. Let $f(\boldsymbol{x}_{\mathbf{obs}}; \boldsymbol{\theta}^e)$ be an encoding function, parameterized by $\boldsymbol{\theta}^e$, which encodes the input vector $\boldsymbol{x}_{\mathbf{obs}} \in X_{\text{obs}}$ into the latent space Z. A decoding function $g(f(\boldsymbol{x}_{\mathbf{obs}}); \boldsymbol{\theta}^d)$, with parameters $\boldsymbol{\theta}^d$, decodes the feature space $Z := f(\boldsymbol{x}_{\mathbf{obs}})$ back to obstacles space $\hat{\boldsymbol{x}}_{\text{obs}} \in X_{\text{obs}}$. The objective function for the CAE is

$$L_{\text{CAE}}(\boldsymbol{\theta}^e, \boldsymbol{\theta}^d) = \frac{1}{N_{\text{obs}}} \sum_{\boldsymbol{x} \in D_{\text{obs}}} ||\boldsymbol{x} - g(f(\boldsymbol{x}))||^2 + \lambda \sum_{ij} (\theta^e_{ij})^2 \tag{1}$$

where λ is a penalizing coefficient, and D_{obs} is a dataset of point clouds $\boldsymbol{x_{\text{obs}}} \in X_{\text{obs}}$ from $N_{\text{obs}} \in \mathbb{N}$ different workspaces. The penalizing term forces the feature space $f(\boldsymbol{x_{\text{obs}}})$ to be contractive in the neighborhood of the training data which results in an invariant and robust feature learning [19].

2) Deep Multi-Layer Perceptron (DMLP): We use a feed-forward deep neural network, parameterized by $\boldsymbol{\theta}$, to perform motion planning. Given the obstacles encoding Z, current state x_t and the goal state x_T, DMLP predicts the next state $\hat{x}_{t+1} \in X_{\text{free}}$ which would lead a robot closer to the goal region i.e.,

$$\hat{x}_{t+1} = \text{DMLP}((x_t, x_T, Z); \boldsymbol{\theta}) \tag{2}$$

To train DMLP, we use RRT* [10] to generate feasible, near-optimal paths in various environments. The paths given by RRT* are a tuple, $\tau^* = \{x_0, x_1, \cdots, x_T\}$, of feasible states that connect the start and goal configurations so that the connected path lies entirely in X_{free}. The training objective for the DMLP is to minimize the mean-squared-error (MSE) loss between the predicted states $\hat{x}_{t+1}$ and the actual states x_{t+1} given by the RRT*. The training loss for the DMLP is formalized as follow:

$$L_{\text{mlp}}(\boldsymbol{\theta}) = \frac{1}{N_p} \sum_{j}^{\hat{N}} \sum_{i=0}^{T-1} ||\hat{x}_{j,i+1} - x_{j,i+1}||^2, \tag{3}$$

where $N_p \in \mathbb{N}$ is the averaging term corresponding to the total number of paths, $\hat{N} \in \mathbb{N}$, in the training dataset times the path lengths.

B. Online Path Planning

The online phase exploits the neural models from the offline phase to do motion planning in cluttered and complex environments. The overall flow of information between encoder $f(\boldsymbol{x_{\text{obs}}})$ and the DMLP is shown in Fig. 2(c). To generate end-to-end feasible paths connecting the start and goal states, we propose a novel incremental bidirectional path generation heuristic. Algorithm 1 presents the overall path generation procedure. The rest of the section describes various functions used by Algorithm 1 and the overall execution of the proposed method.

1) Obstacles Encoder $f(\boldsymbol{x_{\text{obs}}})$: The encoder function $f(\boldsymbol{x_{\text{obs}}})$, trained during the offline phase, is used to encode the obstacles point cloud $\boldsymbol{x_{\text{obs}}} \in X_{\text{obs}}$ into a latent space $Z \in \mathbb{R}^m$.

2) DMLP: The DMLP is a feed-forward neural network from the offline phase which takes Z, current state x_t, goal state x_T and predicts the next state of the robot $\hat{x}_{t+1}$. To inculcate stochasticity into the DMLP, some of the hidden units in each hidden layer of the DMLP were dropped out with a probability $p : [0, 1] \in \mathbb{R}$. The merit of adding the stochasticity during the online path generation is formally discussed in the discussion section.

Algorithm 1: MPNet($x_{\text{init}}, x_{\text{goal}}, \boldsymbol{x_{\text{obs}}}$)

```
Z ← f(x_obs)
τ ← NeuralPlanner(x_init, x_goal, Z);
if τ then
    τ ← LazyStatesContraction(τ)
    if IsFeasible(τ) then
        return τ
    else
        τ_new ← Replanning(τ, Z)
        τ_new ← LazyStatesContraction(τ_new)
        if IsFeasible(τ_new) then
            return τ_new
        return ∅
```

Algorithm 2: NeuralPlanner($x_{\text{start}}, x_{\text{end}}, Z$)

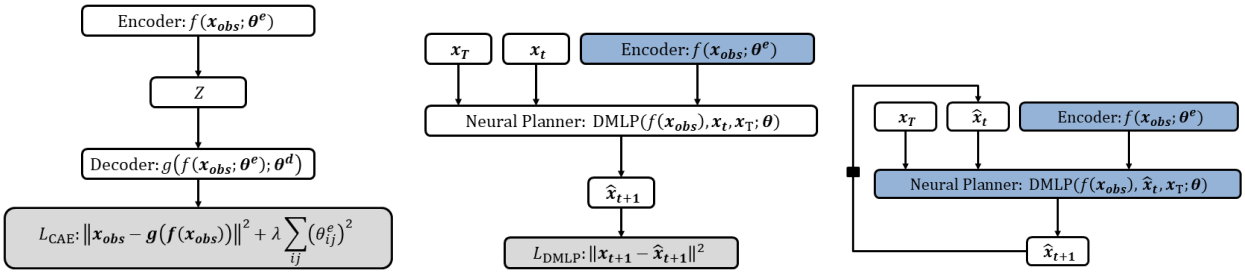
```
τ^a ← {x_start}; τ^b ← {x_end};
τ ← ∅;
Reached ← False;
for i ← 0 to N do
    x_new ← DMLP(Z, τ^a(end), τ^b(end))
    τ^a ← τ^a ∪ {x_new}
    Connect ← steerTo(τ^a(end), τ^b(end))
    if Connect then
        τ ← concatenate(τ^a, τ^b)
        return τ
    SWAP(τ^a, τ^b)
return ∅
```

3) Lazy States Contraction (LSC): Given a path $\tau = \{x_0, x_1, \cdots, x_T\}$, the LSC algorithm connects the directly connectable non-consecutive states, i.e., x_i and $x_{>i+1}$, and removes the intermediate/lazy states. This process is also often known as smoothing or shortcutting. The term contraction is coined as it is used in graph theory literature [17].

4) Steering: The steerTo function takes two states as an input and checks either a straight trajectory connecting the given two states lies entirely in collision-free space X_{free} or not. The steering is done from x_1 to x_2 in small, discrete steps and can be summarized as $\tau(\delta) = (1 - \delta)x_1 + \delta x_2; \delta \in [0, 1]$.

5) isFeasible: Given a path $\tau = \{x_0, x_1, \cdots, x_T\}$, this procedure checks either the end-to-end path, formed by connecting the consecutive states in τ, lies entirely in X_{free} or not. The output is a boolean which is set to TRUE if the path τ is completely collision-free otherwise to FALSE.

6) Neural Planner: This is an incremental bidirectional DMLP based path generation heuristic, see Algorithm 2 for the outline. It takes the obstacles representation, Z, as well as the start and goal states as an input, and outputs a path connecting the two given states. The sets τ^a and τ^b correspond to the



(a) Offline: Contractive Autoencoder (b) Offline: Deep Multi-layer Perceptron (c) Online: Neural Planner

Fig. 2: The offline and online phases of MPNet. The grey shaded blocks indicate the training objectives. The sky-blue and white blocks represent frozen and non-frozen modules, respectively. The frozen modules do not undergo any training.

Algorithm 3: Replanning(τ, Z)

```
1 for i ← 0 to τ.length() do
2     if steerTo(τ_i, τ_{i+1}) then
3         τ_new ← τ_new ∪ {τ(i), τ(i+1)}
4     else
5         τ_mini ← Replanner(τ_i, τ_{i+1}, Z);
6         if τ_mini then
7             τ_new ← τ_new ∪ τ_mini
8         else
9             return ∅
```

paths generated from the start and goal states, respectively. The algorithm starts with τ^a, it generates a new state x_{new} from start towards the goal (Line 5), and checks if a path from start τ^a is connectable to the path from a goal τ^b (Line 7). If paths are connectable, an end-to-end path τ is returned by concatenating τ^a and τ^b. However, if paths are not connectable, the roles of τ^a and τ^b are swapped (Line 11) and the whole procedure is repeated again. The swap function enables the bidirectional generation of paths i.e., if at any iteration i, path τ^a is extended then in the next iteration $i+1$, path τ^b will be extended. This way, two trajectories τ^a and τ^b march towards each other which makes this path generation heuristic greedy and fast.

7) Replanning: This procedure is outlined in the Algorithm 3. It iterates over all the consecutive states x_i and x_{i+1} in a given path $\tau = \{x_0, x_1, \cdots, x_T\}$, and checks if they are connectable or not, where $i = [0, T-1] \subset \mathbb{N}$. If any consecutive states are found not connectable, a new path is generated between those states using one of the following replanning methods (Line 5).

a) Neural Replanning: Given a start and goal states together with obstacle space encoding Z, this method recursively finds a new path between the two given states. To do so, it starts by finding a coarse path between the given states and then if required, it replans on a finer level by calling itself over the non-connectable consecutive states of the new path. This recursive neural replanning is performed for the fixed number of steps to limit the algorithm within the computational bounds.

b) Hybrid Replanning: This heuristic combines the neural replanning with the classical motion planning methods. It performs the neural replanning for the fixed number of steps. The resulting new path is tested for feasibility. If a path is not feasible, the non-connectable states in the new path are then connected using a classical motion planner.

Algorithm 1 outlines the overall procedure which exploits all the aforementioned heuristics to generate the feasible end-to-end paths. It starts by finding a coarse path τ connecting start and goal (Line 2). If a valid path, τ, is found, the lazy states in the path are removed (Line 4) and the feasibility tests are performed (Line 5). If a path is feasible, it is returned as a path solution otherwise, a replanning is done on a finer level to repair the segments of the coarse path which does not lie entirely in the obstacle-free space (Line 8). The replanning method returns a new feasible path if one exists. This new path is returned as a path solution after lazy states contraction (Lines 9-11).

V. Implementation Details

This section gives the implementation details of MPNet. The proposed neural models, CAE and DMLP, are implemented in PyTorch[2]. The path generation heuristic and the classical motion planner (RRT*) are implemented in Python. The system used for training and testing has 3.40GHz$\times$ 8 Intel Core i7 processor with 32 GB RAM and GeForce GTX 1080 GPU. The remaining section explains different modules that lead to MPNet.

A. Data Collection

To generate different 2D and 3D workspaces, a number of quadrilateral blocks were placed in the operating region of 40×40 and $40 \times 40 \times 40$, respectively. The positions of these blocks were randomly sampled without replacement from the operating region. Each random placement of the obstacle blocks leads to a different workspace. To generate

[2]pytorch.org

different feasible start and goal states within each of the generated workspaces, a list of $n \in \mathbb{N}$ states were randomly sampled from the obstacle-free space. A pair of states from the list were selected randomly, without replacement, to form a start and goal pair. These start and goal pairs were then used to generate the feasible paths (using RRT*) for training and testing. By following the aforementioned procedure, 110 different workspaces were generated for each presented case i.e., simple 2D (s2D), rigid-body (rigid), complex 2D (c2D) and 3D (c3D) (see next section). In each of the workspaces, 5000 collision-free paths were generated using RRT*. The training dataset comprised of 100 workspaces with 4000 paths in each workspace. For testing, two types of test datasets were created to evaluate the proposed method. The first test dataset comprised of already seen 100 workspaces with 200 unseen start and goal configurations in each workspace. The second test dataset comprised of completely unseen 10 workspaces where each contained 2000 unseen start and goal configurations. In the case of Baxter, we do not include the environment encoding as we only consider a single environment. Therefore, only start and goal configurations are sampled to compute training trajectories (50,000) from obstacle-free space to train DMLP for Baxter[3].

B. Models Architecture

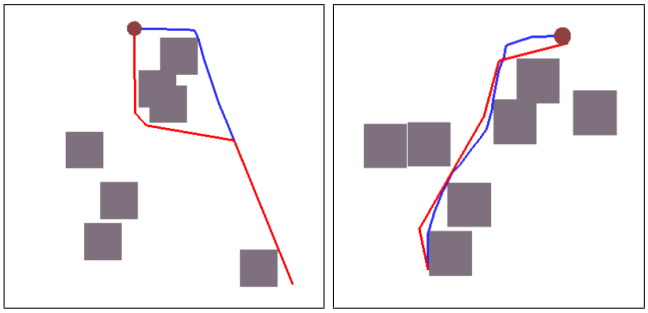
1) Contractive AutoEncoder (CAE): The encoding function $f(\boldsymbol{x_{\text{obs}}})$ and decoding function $g(f(\boldsymbol{x_{\text{obs}}}))$ consist of three linear layers and one output layer, where each linear layer is followed by the Parametric Rectified Linear Unit (PReLU) [23]. Since the structure of the decoder is simply the inverse of encoding unit, we only describe the structure of the encoder.

The input to the encoder is a vector of point clouds of size $1400 \times d$ where 1400 are the points along each dimension, and $d \in \mathbb{N}_{\geq 2}$ is the dimension of a workspace. For 2D workspaces, the layers 1, 2 and 3 transforms the input vector 1400×2 to 512, 256 and 128 hidden units, respectively. The output layer takes the 128 units as an input and outputs 28 units. Hence, the obstacle representation $Z \in \mathbb{R}^m$ is a vector of size $m = 28$. These 28 units are then taken by the decoder to reconstruct the 1400×2 space. For the 3D workspaces, the layers 1,2 and 3 maps the input vector 1400×3 to 786, 512, and 256 hidden units. The output layer transforms the input 256 units to output 60 units, i.e., $Z \in \mathbb{R}^{60}$.

2) Deep Multi-layer Perceptron (DMLP): The DMLP is a 12-layer DNN. For point-mass and rigid-body cases, the input is given by concatenating the obstacles' representation Z, start x_{init} and goal x_{goal} configurations. The configurations for the 2D point-mass robot, 3D point-mass robot and rigid-body have dimensions 2, 3 and 3 respectively. For Baxter robot, we do not include obstacles encoding, and only the start and goal configurations are concatenated to form an input for DMLP. The dimensionality of Baxter's arm configuration space is 7.

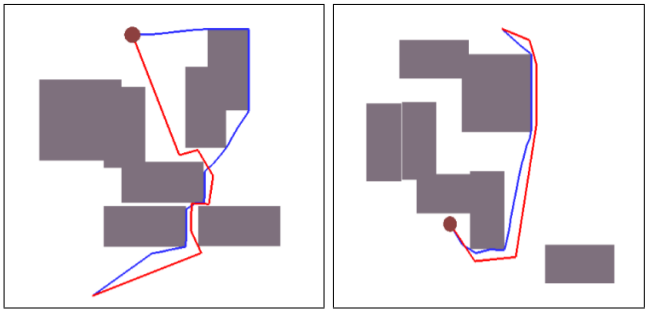
Each of the first nine layers is a sandwich of a linear layer, Parametric Rectified Linear Unit (PReLU) [23] and Dropout(p) [21]. Layers one to nine transforms the input vectors to 1280, 1024, 896, 768, 512, 384, 256, 256 and 128 hidden units, respectively. The tenth and eleventh layers do not use Dropout and transform the inputs to 64 and 32 units, respectively. The output layer takes the 32 units from the 11-th layer and transform them to the dimensions of robot configuration space.

[3]www.rethinkrobotics.com/baxter



(a) $t_{\text{R}} = 5.4s, t_{\text{MP}} = 0.13s$ (b) $t_{\text{R}} = 6.2s, t_{\text{MP}} = 0.19s$

Fig. 3: MPNet (Red) and RRT* (Blue) planning paths in simple 2D environments (s2D).



(a) $t_{\text{R}} = 5.4s, t_{\text{MP}} = 0.13s$ (b) $t_{\text{R}} = 6.2s, t_{\text{MP}} = 0.19s$

Fig. 4: MPNet (Red) and RRT* (Blue) planning motions in complex 2D environments (c2D).

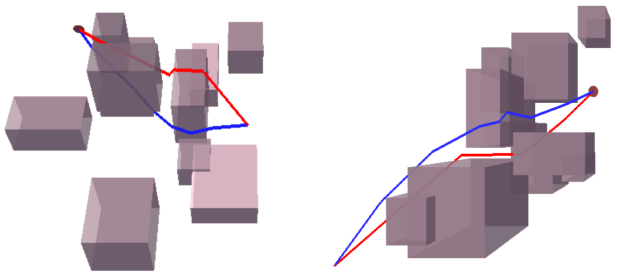
Environment	Accuracy (%)		
	MPNet: NR		MPNet: HR
	Seen-X_{obs}	Unseen-X_{obs}	Seen/Unseen-X_{obs}
s2D: Simple 2D	99.3	98.3	100
c2D: Complex 2D	99.7	98.8	100
c3D: Complex 3D	99.1	97.7	100
rigid: Rigid-body	98.2	97.1	100

TABLE I: Presents the mean accuracy of MPNet with Neural-Replanning (NR) and Hybrid-Replanning (HR) on two test datasets. The test datasets seen-X_{obs} and unseen-X_{obs} evaluate MPNet to unseen start and goal states in already seen workspaces and entirely unseen workspaces, respectively.

C. Hyper-parameters

To train the neural models CAE and DMLP, the Adagrad [2] optimizer was used with the learning rate of 0.1. The Dropout probability p and penalizing term λ were set to 0.5

and 10^{-3}, respectively. To train CAE, $N_{\text{obs}} = 30,000$ different workspaces were generated randomly by the procedure described in section V-A. One of the test datasets contained the 10 workspaces which were neither seen by the CAE nor by the DMLP. For RRT* motion planner, the step size for tree extension was set to 0.9 and 0.01 for the rigid body and point-mass robot case, respectively. The constant gamma, defining rewiring ball radius, was set to 1.6 (see [10] for details on RRT* hyper-parameters). However, for data generation on Baxter, we use OMPL's[4] RRT* and ROS[5] with their default parameters setting.



(a) $t_{\text{R}} = 5.3s, t_{\text{MP}} = 0.44s$ (b) $t_{\text{R}} = 6.9s, t_{\text{MP}} = 0.50s$

Fig. 5: MPNet (Red) and RRT* (Blue) planning paths in complex 3D environments (c3D).

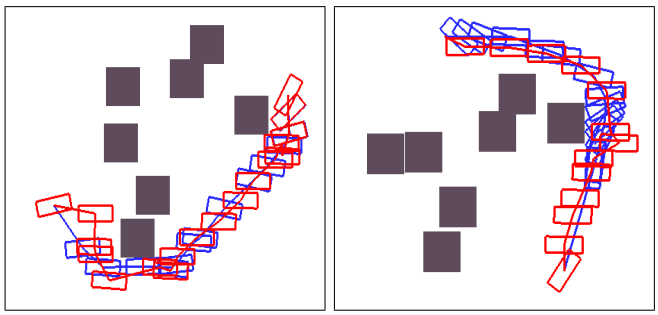
VI. Results

This section presents the results of MPNet for the motion planning of a point-mass robot and a rigid-body in the 2D and 3D environments. Table I summarizes the mean accuracy of MPNet with Neural-Replanning (MPNet: NR) and Hybrid-Replanning (MPNet: HR) in two test cases. The first test case consists of new unseen start and goal configurations but seen obstacle positions i.e., seen-X_{obs}. The second test case corresponds to the completely new workspaces where obstacle locations were not seen by MPNet during training i.e., unseen-X_{obs}. Our performance measure corresponds to the percentage of planning problems successfully solved by MPNet. Since Dropout adds stochasticity to MPNet, we use 20 forward passes through MPNet to calculate the mean performance. The mean accuracy of MPNet: HR was 100% for all test cases whereas for MPNet: NP, the mean accuracy was about 97% and the standard deviation for all cases was roughly 0.4%.

Figs. 3-6 show different example scenarios where MPNet and RRT* provided successful paths. The red and blue colored trajectories indicate the paths generated by MPNet and RRT*, respectively. The goal region is indicated as a brown colored disk. The mean computational time for the MPNet and RRT* is denoted as t_{MP} and t_{R}, respectively. We see that MPNet is able to compute near-optimal paths for both point-mass and rigid-body robot in considerably less time than RRT*.

[4]ompl.kavrakilab.org
[5]www.ros.org

Table II presents the time comparison of MPNet with neural replanning and hybrid replanning against state-of-the-art sampling-based motion planning methods i.e., Informed-RRT* [5] and BIT* [6] over the two above-mentioned test cases. We report the mean times with standard deviation of all algorithms in a given problem. It can be seen that in all test cases, the mean computation time of MPNet with neural and hybrid replanning remained around 1 second. However, the mean computation time of Informed-RRT* and BIT* increases significantly as the dimensionality of planning problem is increased. Note that, on average, MPNet is about 40 and 20 times faster than Informed-RRT* and BIT*, respectively, in all test cases and consistently demonstrates low computational time irrespective of the dimensionality of the planning problem.



(a) $t_{\text{R}} = 43.3, t_{\text{MP}} = 0.33s$ (b) $t_{\text{R}} = 37.3s, t_{\text{MP}} = 0.34s$

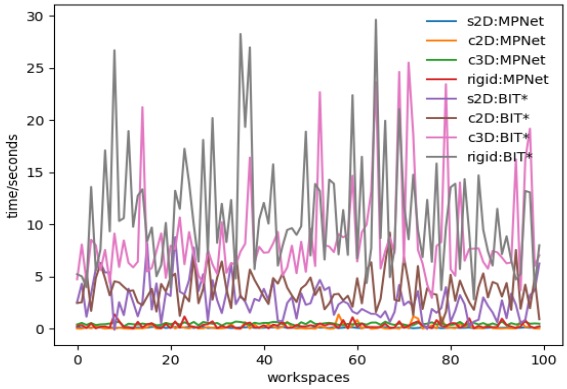
Fig. 6: Rigid body motion planning by MPNet(Red) and RRT*(Blue) in 2D environments (rigid).

From experiments presented so far, it is evident that BIT* outperforms Informed-RRT*, therefore, in the following experiments only MPNet and BIT* are compared. Fig. 7 compares the mean computation time of MPNet with neural replanning and BIT* in two test cases. The first test case contains 100 seen environments i.e., seen-Xobs, with 200 unseen start and goal configurations in each of the environments. The second test case presents the comparison over 10 unseen environments with 2000 new start and goal configurations in each of the environments. It can be seen that the mean computation time of MPNet stays around 1 second irrespective of the planning problem dimensionality. Furthermore, the mean computational time of BIT* not only fluctuates but also increases significantly in the rigid-body planning problem.

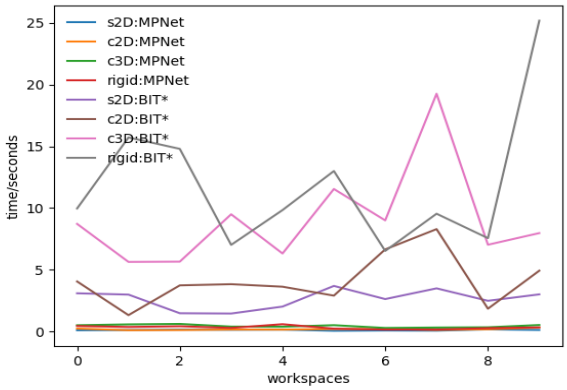
Finally, Fig. 8 shows MPNet planning motions for a dual-arm 7 DOF Baxter robot for a given start and goal configurations. In Fig. 8, the robotic manipulators are at the start configurations, and the shadowed regions indicate the path followed by both manipulators to reach the target objects (shown as green and yellow rigid-bodies). In this problem, MPNet and BIT* have mean computation time of 0.7 and 68.9 seconds, respectively, which makes MPNet around 100 times faster than BIT*.

Environment	Test case	MPNet (NR)	MPNet (HR)	Informed-RRT*	BIT*	$\frac{\text{BIT} : t_{\text{mean}}}{\text{MPNet(NR)} : t_{\text{mean}}}$
Simple 2D	Seen X_{obs}	0.11 ± 0.037	0.19 ± 0.14	5.36 ± 0.34	2.71 ± 1.72	24.64
	Unseen X_{obs}	0.11 ± 0.038	0.34 ± 0.21	5.39 ± 0.18	2.63 ± 0.75	23.91
Complex 2D	Seen X_{obs}	0.17 ± 0.058	0.61 ± 0.35	6.18 ± 1.63	3.77 ± 1.62	22.17
	Unseen X_{obs}	0.18 ± 0.27	0.68 ± 0.41	6.31 ± 0.85	4.12 ± 1.99	22.89
Complex 3D	Seen X_{obs}	0.48 ± 0.10	0.34 ± 0.14	14.92 ± 5.39	8.57 ± 4.65	17.85
	Unseen X_{obs}	0.44 ± 0.107	0.55 ± 0.22	15.54 ± 2.25	8.86 ± 3.83	20.14
Rigid	Seen X_{obs}	0.32 ± 0.28	1.92 ± 1.30	30.25 ± 27.59	11.10 ± 5.59	34.69
	Unseen X_{obs}	0.33 ± 0.13	1.98 ± 1.85	30.38 ± 12.34	11.91 ± 5.34	36.09

TABLE II: Time comparison of MPNet (NR: Neural Replanning; HR: Hybrid Replanning), Informed-RRT* and BIT* on two test datasets.



(a) Test-case 1: seen-X_{obs}



(b) Test-case 2: unseen-X_{obs}

Fig. 7: Computational time comparison of MPNet and RRT* on test datasets. The plots show MPNet is more consistent and faster than RRT* in all test cases.

VII. Discussion

A. Stochasticity through Dropout

The standard practice in Deep Learning is to turn off the Dropout [21] during test time or online execution. However, for our method, Dropout is advantageous for online path generation as we have observed that doing so significantly improves the performance of our model.

Dropout is applied layer-wise to a neural network and it drops each unit in the hidden layer with a probability $p \in [0, 1]$. The resulting neural network is a thinned network and is essentially different from the actual neural model [21].

Note that, in the neural replanning phase, MPNet iterates over the non-connectable consecutive states of the coarse path to do motion planning on a finer level and thus, produces a new path. The replanning procedure is called recursively on each of its own newly generated paths until a feasible solution is found or a loop limit is reached. Dropout adds stochasticity to the DMLP which implies that on each replanning step, the DMLP would generate different paths from previous replanning steps. This phenomenon is evident from Fig. 1 where the DMLP generated different paths for a fixed start and goal configurations. These perturbations in generated paths for fixed start and goal help in recovery from the failure. Thus, adding Dropout increases the overall performance of MPNet.

B. Transfer Learning

Transfer learning is a research which investigates the methods that allow the transfer of knowledge gained while solving one problem to a different but related problem. Neural networks are known for their catastrophic forgetting [11] but with newer developments, a neural network trained in one problem can be fine-tuned with a small amount of dataset to adapt to a new but related problem. We investigate the feature transferring skills of MPNet trained in simple 2D cases to complex 2-D cases. Fig. 9 shows the impact of the size of training dataset as well as the number of top training/non-frozen layers within DMLP on the performance of DMLP in the new workspaces. In Fig. 9, the horizontal axis indicates the portion of the complex 2D training dataset used to fine tune DMLP from simple 2D cases. It can be seen that for small datasets e.g., 5%, fewer non-frozen layers gives better performance whereas for larger datasets e.g., 40%, end-to-end training is beneficial.

C. Completeness

In the proposed method, a coarse path is computed by a neural network. If a coarse path is found to be not fully connectable, a re-planning heuristic is executed to repair the non-connectable path segments to provide an end-to-end collision-free path. The completeness guarantees for the

Fig. 8: MPNet planning motions for a dual-arm 7 DOF Baxeter robot. The robot is shown to be at initial position and the shadowed profile shows the path taken by the robot arms to reach the objects. The mean computational times of MPNet and BIT* are 0.70 and 68.9 seconds, respectively, which makes MPNet about 100 times faster than BIT*.

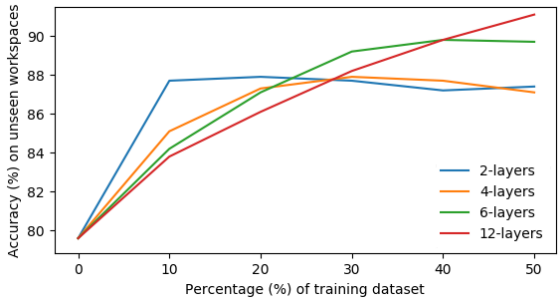


Fig. 9: Transferring skills of MPNet from simple 2D workspaces to complex 2D workspaces. The plot shows deeper networks (e.g., 12-layers) transfer better to new environments as the dataset size increases.

proposed method depends on the underline replanning heuristic. The classical motion planner based replanning methods are presented to guarantee the completeness of the proposed method. If the classical motion planner is A* MPNet is guaranteed to be complete. However, since, we use RRT*, our proposed method inherits the probabilistic completeness of RRTs and RRT* [10] while retaining the computational gains.

D. Computational Complexity

This section formally highlights the computational complexity of the proposed method. Neural networks are known to have online execution complexity of $O(1)$. Therefore, the execution of lines 1-2 of Algorithm 1 will have a complexity no greater than $O(1)$. The lazy state contraction (LSC) heuristic is a simple path smoothing technique which can be executed in a fixed number of iteration as a feasible trajectory have to have a finite length. Also, note that the LSC is not an essential component of the proposed method. Its inclusion helps to generate near-optimal paths. The computational complexity of the replanning heuristic depends on the motion planner used for replanning. The two methods are proposed for replanning i.e., the neural replanning and hybrid replanning methods. Since the neural replanner is executed for fixed number of steps, the complexity is $O(1)$. For the classical motion planner, we use RRT* which has $O(nlogn)$ complexity, where n is the number of samples in the tree [10]. Hence, for hybrid replanning, we can conclude that the proposed method has a worst case complexity of $O(nlogn)$ and a best case complexity of $O(1)$. Note that, the proposed MPNet with neural replanning is able to compute collision-free paths for more than 97% of the cases (see Table I). Therefore, it can be said that MPNet will be operating with $O(1)$ most of the time except for nearly 3% cases where the RRT* needs to be executed. Moreover, note that for those 3% cases, only a small segment of a path, given by MPNet, which does not lie in the obstacle-free space is repaired through RRT* (see section IV). This execution of RRT* on small segments of a global path reduces the complicated problem to a simple planning problem which makes the RRT* execution computationally acceptable and practically much less than $O(nlogn)$.

VIII. Conclusions and Future Work

In this paper, we present a fast and efficient Neural Motion Planner called MPNet. MPNet consists of a Contractive AutoEncoder that takes the point cloud of a robot's surrounding to encode them into an invariant feature space, and a feedforward neural network that takes the environment encoding, and start and goal robotic configurations to output a collision-free feasible path connecting the given configurations. The proposed method (1) plans motions irrespective of the obstacles geometry, (2) demonstrates mean execution time of about 1 second in different 2D and 3D workspaces, (3) generalizes to new unseen obstacle locations, (4) adapts very quickly with a small dataset to different working situations e.g., from indoor living places to factory floors, and (5) has completeness guarantees.

In our future work, we plan to extend MPNet to dynamic environments as its great generalization to the new obstacle locations makes it the best candidate for time-varying motion planning problems. Another interesting extension would be to add neural attention on the workspace encoding. We recognize through unfreezing the Encoder block in Fig. 2(b), this is achieved without any change to MPNet, i.e., by training the MPNet end-to-end in Fig. 2(b), an encoder can be realized that can attend to the regions which are crucial to planning a path for any given start and goal states.

References

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

[2] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[3] Paul Fitzpatrick, Kensuke Harada, Charles C Kemp, Yoshio Matsumoto, Kazuhito Yokoi, and Eiichi Yoshida. Humanoids. In *Springer Handbook of Robotics*, pages 1789–1818. Springer, 2016.

[4] Emilio Frazzoli, Munther A Dahleh, and Eric Feron. Real-time motion planning for agile autonomous vehicles. *Journal of guidance, control, and dynamics*, 25(1): 116–129, 2002.

[5] 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 *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, pages 2997–3004. IEEE, 2014.

[6] 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 *Robotics and Automation (ICRA), 2015 IEEE International Conference on*, pages 3067–3074. IEEE, 2015.

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

[8] Stephen Grossberg. Nonlinear neural networks: Principles, mechanisms, and architectures. *Neural networks*, 1 (1):17–61, 1988.

[9] Alan L Hodgkin and Andrew F Huxley. A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of physiology*, 117(4):500–544, 1952.

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

[11] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the National Academy of Sciences*, 114(13):3521–3526, 2017.

[12] Sven Koenig, Maxim Likhachev, Yaxin Liu, and David Furcy. Incremental heuristic search in ai. *AI Magazine*, 25(2):99, 2004.

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

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

[15] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *Journal of Machine Learning Research*, 17(39):1–40, 2016.

[16] Tomás Lozano-Perez. *Autonomous robot vehicles*. Springer Science & Business Media, 2012.

[17] S Muthukrishnan and Gopal Pandurangan. The bin-covering technique for thresholding random geometric graph properties. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 989–998. Society for Industrial and Applied Mathematics, 2005.

[18] Ahmed Hussain Qureshi and Yasar Ayaz. Potential functions based sampling heuristic for optimal path planning. *Autonomous Robots*, 40(6):1079–1093, 2016.

[19] Salah Rifai, Pascal Vincent, Xavier Muller, Xavier Glorot, and Yoshua Bengio. Contractive auto-encoders: Explicit invariance during feature extraction. In *Proceedings of the 28th International Conference on International Conference on Machine Learning*, pages 833–840. Omnipress, 2011.

[20] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[21] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research*, 15(1):1929–1958, 2014.

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

[23] Ludovic Trottier, Philippe Giguère, and Brahim Chaib-draa. Parametric exponential linear unit for deep convolutional neural networks. *arXiv preprint arXiv:1605.09332*, 2016.

[24] Richard Volpe. Rover functional autonomy development for the mars mobile science laboratory. In *Proceedings of the 2003 IEEE Aerospace Conference*, volume 2, pages 643–652, 2003.

[25] Simon X Yang and Max Meng. An efficient neural network approach to dynamic robot motion planning. *Neural Networks*, 13(2):143–148, 2000.

[26] Michael Yip and Nikhil Das. Robot autonomy for surgery. *arXiv preprint arXiv:1707.03080*, 2017.

[27] Junku Yuh. Design and control of autonomous underwater robots: A survey. *Autonomous Robots*, 8(1):7–24, 2000.