Faster Neural Path Planner Based on MPNet

1st Zhirui Dai
Dept. Electrical and Computer Engineering
University of California, San Diego
zhdai@eng.ucsd.edu

***Abstract*—This paper proposes an improved neural motion planner with the Motion Planning Network (MPNet) described in [2]. MPNet is a computationally efficient, learning-based neural planner for solving motion planning problems. MPNet learns a general feature presentation of an environment with obstacles and a sub-optimal heuristic stochastic mapping from the space of environment feature and start position to the space of the next path waypoint. By recursively calling MPNet, a feasible path can be built finally. In [1] and [2], Qureshi et. al show that the C++ version of neural planner based on the batch-training MPNet model can reach the success rate above 90% with 0.03s on Simple 2D test, 0.06s on Complex 3D and 0.53s on Rigid-body-SE2. Some other techniques like active learning are also proposed to further improve the success rate. In this paper, I present an improved neural planner which can reach the similar success rate with a MPNet model trained with fewer epochs, costing at least 50% less running time even under Python implementation.**

I. Introduction

Motion planning is an essential module in a mobile robot system. It aims to find a collision-free and low-cost path connecting a start and a goal in the configuration space. An ideal motion planner should provide an optimal path when it is feasible and report the unfeasibility when it is impossible to reach the destination from the specified location. Besides, the planner should be time-efficient under specific computation resources. Many classical motion planning algorithms have been proposed. Some of them focus on the optimality such as dynamic programming, and label correcting, while others like rapidly-exploring random trees (RRT) and its variant versions aims to find a feasible sub-optimal path faster. However, as the complexity increases, e.g. the increasing of the number of obstacles, or the geometric complexity of obstacles, these classical motion planning algorithms requires more and more memory space and the speed drops dramatically.

Recently, researchers exploit the potential of machine learning and deep learning methods on motion planning and collision checking. MPNet proposed by Qureshi et. al [1], [2] is an impressive motion planning neural network. It receives a vector of point cloud data representing obstacles in the environment and the coordinates of the start location as the input. After compressing the obstacle point cloud data to a compact feature vector, MPNet infers the next feasible waypoint from the obstacle feature vector and the start location. However, the neural planner algorithm proposed by Qureshi et. al does not utilize the MPNet well enough. The drawbacks will be discussed in the next section.

Considering the features of neural network such as parallel computation, and the bottle neck of speed performance, collision detection, I propose a different neural planner algorithm by using MPNet as a tool for fixing path at the very beginning and improving the efficiency of collision checking procedure.

The remaining paper includes the following contents. Section II presents more details of MPNet and some drawbacks of the original neural planning algorithm. Section III shows the algorithm of collision checking, the algorithm of path smoothing, and the algorithm of faster neural path planner (Faster NPP). Section IV shows some benchmark results which shows that Faster NPP can reach a success rate above 95% within 0.015s averaged over all Simple 2D test cases.

II. Related Work

A. MPNet

MPNet consists of an encoder network (ENet) for encoding the environment obstacles and a planning network (PNet) for inferring the next waypoint towards the goal location. (Fig. 1(a)).

ENet takes the information of environment such as point cloud data of obstacles and outputs a compact feature representation of the environment. For different kinds of environment information, ENet can be implemented with different types of neural network. For images of the environment, a convolutional neural network is preferred. For point cloud data, we can simply use multiple fully connected layer to extract the feature of environment z.

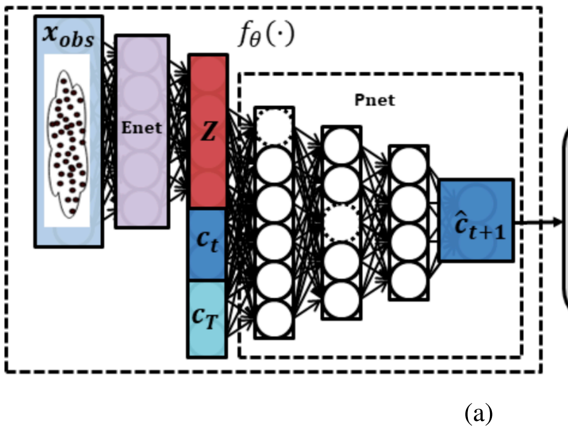
$$z = \text{ENet}(x_{obs}; \theta^e) \tag{1}$$

ENet is an encoder. So, we can either pre-train ENet with a corresponding decoder network or train ENet with PNet together in an end-to-end manner. For the encoder-decoder method, the loss used for training ENet can be the mean square error (MSE) between the input and the rebuilt input.

PNet is a network of multiple fully-connected layers where every linear layer is followed by a PReLU and a Dropout. PNet takes in the environment encoding z, the current/initial location as start c_t, and the goal c_{goal}. Then, PNet predicts the next waypoint $\hat{c}_{t+1}$ to get closer to the goal. $\hat{c}_{t+1}$ is used as the start in the next stage during test time. (Fig. 1(b)).

$$\hat{c}_{t+1} = \text{PNet}(z, c_t, c_{goal}; \theta^p) \tag{2}$$

To train the PNet, the MSE between $\hat{c}_{t+1}$ and ground truth c_{t+1} generated by any classical motion planning algorithm can be used.


x_{obs}
Enet
$f_\theta(\cdot)$
Z
c_t
c_T
Pnet
$\hat{c}_{t+1}$
$l(f_{\theta,t}(x_{obs}, c_t, c_T), \hat{c}_{t+1})$
s.t
$l(f_{\theta,t}, \mathcal{M}) \leq l(f_{\theta,t-1}, \mathcal{M})$
(a)

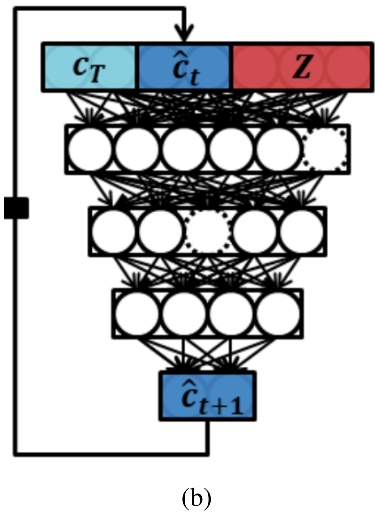


Fig. 1: (a) shows the architecture of MPNet and (b) shows how to recursively use MPNet to plan the path. [2]

In [2], three training methods are proposed, offline batch-learning, continual learning and active continual learning (ACL). ACL is designed for the MPNet to learn from streaming data without forgetting the history. The Faster NPP is compatible with these training methods. So, the offline batch-learning will be used for training in this paper.

Algorithm 1: MPNetPath($c_{init}, c_{goal}, x_{obs}$)

```
z ← ENet($x_{obs}$)
$\sigma$ ← BNP($c_{init}, c_{goal}, z$)
$\sigma$ ← LazyStatesContraction($\sigma$)
if isFeasible($\sigma$) then
    return $\sigma$
else
    for $i \leftarrow 0$ : MAX_TRIES do
        $\sigma$ ← Replan($\sigma, z$)
        $\sigma$ ← LazyStatesContraction($\sigma$)
        if isFeasible($\sigma$) then return $\sigma$;
    return $\varnothing$
```

Algorithm 2: steerTo(σ_i, σ_{i+1})

```
$\beta$ ← STEP_SIZE
for $\alpha \leftarrow 0 : \beta : 1$ do
    $\sigma \leftarrow (1-\alpha)\sigma_i + \alpha\sigma_{i+1}$
    if isInCollision($\sigma$) then return 1 ;
return 0
```

B. Neural Path Planner (NPP)

The algorithm of NPP in [2] uses MPNet to plan the path bi-directionally and then tries to fix some path segments occluded by obstacles. It performs well in time consumption and success rate. Compared with variant RRT algorithm series, MPNet can sample waypoints in the configuration space with much higher efficiency because it samples waypoints in the feasible

Algorithm 3: BNP(c_{init}, c_{goal}, z)

```
$\sigma^a \leftarrow \{c_{init}\}$, $\sigma^b \leftarrow \{c_{goal}\}$
tree ← 0
for $i \leftarrow 0 : N$ do
    if tree == 0 then
        $c_{new}$ ← PNet($z, \sigma^a_{end}, \sigma^b_{end}$)
        $\sigma^a \leftarrow \sigma^a \cup \{c_{new}\}$
    else
        $c_{new}$ ← PNet($z, \sigma^b_{end}, \sigma^a_{end}$)
        $\sigma^b \leftarrow \sigma^b \cup \{c_{new}\}$
    tree ← not tree
    if steerTo($\sigma^a_{end}, \sigma^b_{end}$) then
        $\sigma \leftarrow \sigma^a \cup$ Inverse($\sigma^b$)
        return $\sigma$
return $\varnothing$
```

Algorithm 4: Replan(σ, z)

```
$\sigma_{new} \leftarrow \{\sigma_0\}$
for $i = 0 : size(\sigma) - 1$ do
    if steerTo($\sigma_i$, $\sigma_{i+1}$) then
        $\sigma_{new} \leftarrow \sigma_{new} \cup \{\sigma_{i+1}\}$
    else
        $\sigma'$ ← BNP($\sigma_i, \sigma_{i+1}, z$)
        if $\sigma'$ then
            $\sigma_{new} \leftarrow \sigma_{new} \cup \sigma'[1:\text{end}]$
        else
            return $\varnothing$
return $\sigma_{new}$
```

configuration space with higher probability. Accompanied with the implicit heuristic function learned from the training set, MPNet is also better at finding the next waypoint to get closer to the goal, unlike RRT doing totally random sampling. In a word, planners based on MPNet absorb the merits of A* algorithm and RRT* algorithm. Algorithms (1 to 4) show the baseline version of neural path planner $MPNetPath : NP$ described in [2].

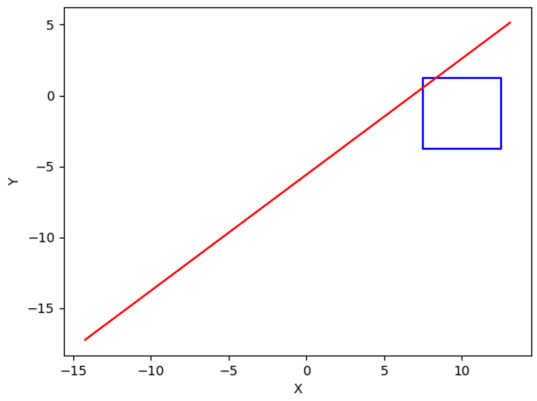


Fig. 2: A 2D case where discrete collision detection algorithm ignores the collision under the step size of 0.01

C. Drawbacks

This planner already achieves impressive success rate and speed as the statement in previous section. But there are still some drawbacks.

- The simplest feasible path $[c_{init}, c_{goal}]$ is not considered in the above algorithms. Such a case does not need to use MPNet to do path planning. However, the Algorithm (1) does the path planning directly without checking the path feasibility. This may result in some failed cases that could have been successful.
- We can see that $Replan$ described in Algorithm (4) fixes the path segments in collision one by one. This serial procedure does not use the parallel feature of MPNet.
- $steerTo$ in Algorithm (2) checks the collision by dividing a path segment into smaller segment, which may ignore some collision beneath the step size. This method is inefficient. There are other collision detection algorithms which can detect the collision between obstacles and a path segment. For example, by wrapping obstacles with polygons, we can use Minkowski difference to detect the collision between a polygon and a line segment. However, these methods are still not scalable or not general enough. With the encoded obstacle feature, a neural network can be used at the frontend for collision detection. And these classical algorithms can be used to guarantee that the path is collision-free.
- The path is not contracted in time so that once the case is very difficult, the path generated by MPNet in the first round could be much longer than average and slows down the subsequent procedure to fix the in-collision path segments. Actually, a path contraction, which is called $smoothPath$ in the next section, can generate a simplified path and a record indicating which segment needs repair. Therefore, the burden of collision checking can be relieved a lot.

III. Faster Neural Path Planner

A. Overview

Considering the above drawbacks, faster neural path planner (Faster NPP) adopts the following improvements:

- Path Repair: use MPNet as a tool to fix the path at the very beginning so that the procedure can become much simpler. The first repair if necessary is to fix the path segment $[c_{init}, c_{goal}]$. Sometimes the straight path from c_{init} to c_{goal} is feasible and therefore it does not require any repair.
- Collision Detection: use collision detection neural network (CDNet) to detect collision in the planning procedure of MPNet and use classical collision detection algorithm to determine if the path needs further repair.
- Path Smoothing: unlike the original NPP that does path contraction and feasibility checking via $LazyStatesContraction$ and $isFeasible$ separately, Faster NPP tries to simplify the path and update the record of path segment feasibility simultaneously in every repair. Once the record shows that all the path segments are feasible, this path is qualified.

The top level of Faster NPP is given in Algorithm (5). Details of sub routines will be presented in the remaining of the paper.

Algorithm 5: FasterNPP($X_{obs}, c_{start}, c_{goal}$)

```
1  $\sigma \leftarrow \{c_{start}, c_{goal}\}$
2  $\sigma, V \leftarrow$ PathSmooth($\sigma, X_{obs}$)
3  $f \leftarrow$ all elements in $V$ are True
4  itr $\leftarrow 0$
5  while not $f$ and itr $<$ MAX_TRIES do
6      itr += 1
7      $C_{start} \leftarrow$ StartsOfBadSegments($\sigma, V$)
8      $C_{end} \leftarrow$ EndsOfBadSegments($\sigma, V$)
9      $\sigma_{new} \leftarrow$ FixWithMPNet($X_{obs}, C_{start}, C_{end}$)
10     $\sigma, V \leftarrow$ PathSmooth($\sigma_{new}, X_{obs}$)
11     $f \leftarrow$ all elements in $V$ are True
```

B. Collision Detection: Minkowski Difference

Theoretically, Minkowski difference is a perfect tool to detect collision between objects. However, its efficiency is very low in the collision detection of complex objects. Hence, some measures are required to simplify the obstacle representation. In 2D, polygon could be used to wrap an object. (Fig. (3)) In 3D, the collision detection between axis-aligned bounding boxes (AABB) and line segments can be divided into three sub-detections by orthogonal projection. (Fig. (4))

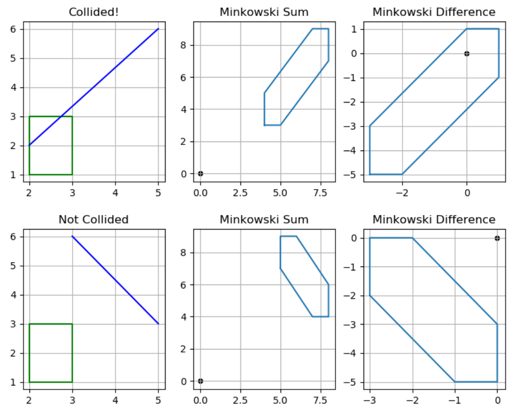


Fig. 3: Minkowski sum and difference between a rectangle and a segment. When collision happens, the Minkowski difference contains the origin.

The definition of Minkowski sum and Minkowski difference are as follows.

Given two sets A and B which represent two objects, the Minkowski sum is

$$A \oplus B = \{a + b \,|a \in A, b \in B\} \tag{3}$$

Similarly, the Minkowski difference is defined as

$$A \ominus B = \{a - b \,|a \in A, b \in B\} \tag{4}$$

To implement Minkowski difference, we can implement Minkowski sun at first. To calculate Minkowski difference, we can mirror b around the origin at first to get the mirrored b'. Then, calculate the Minkowski sun between a and b'.

We also need an efficient method to check if the Minkowski difference contains the origin. Winding number algorithm [3] is a useful tool for detecting if a point is inside a polygon, which can be used for the task.

When the winding number is 0, the specified point P is on the edge of the polygon V or outside it. Otherwise, the point P is inside the polygon V.

Then, the collision detection algorithm based on Minkowski difference is

C. Collision Detection: Neural Network

Neural network is another powerful tool to predict collision efficiently. With features encoded by ENet, a classifier can be trained to classify line segments into two classes, in-collision and collision-free.

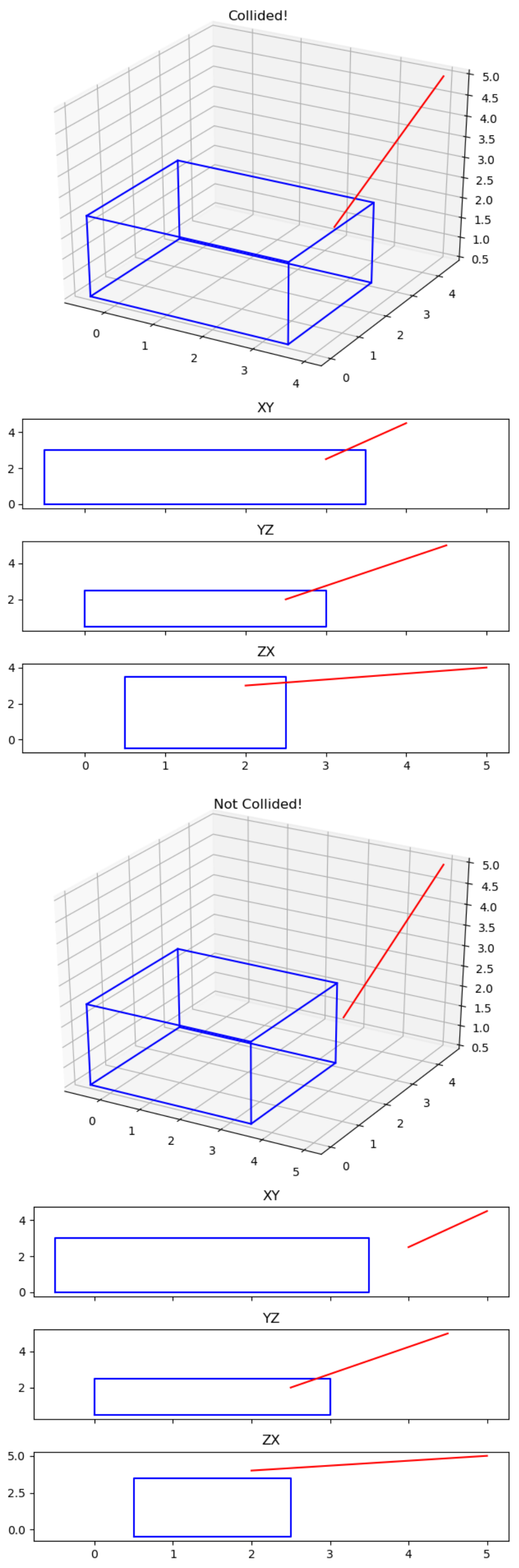


Fig. 4: Collision detection of AABB and line segment can be done by detecting the collision of projections on three 2D planes: XY, YZ and ZX.

Algorithm 6: WindingNumber(P, V)

```
Input  : point P = (x, y) and vertices of a polygon V
Output: the winding number W
1  W ← 0
2  for every edge of the polygon do
3      $v_i$, $v_{i+1}$ ← two ends of the edge
4      $(x_1, y_1) \leftarrow v_i$, $(x_2, y_2) \leftarrow v_{i+1}$
5      if $y_1 < y$ then
6          if $y_2 > y$ and isLeft($v_i, v_{i+1}, P$) then
7              W += 1
8      else if $y_2 \leq y$ and isRight($v_i, v_{i+1}, P$) then
9              W -= 1
10 return W
11 Function isLeft($p_1, p_2, p$):
12     return $[p_1p_2 \times p_1p]_z > 0$
13 Function isRight($p_1, p_2, p$):
14     return $[p_1p_2 \times p_1p]_z < 0$
```

Algorithm 7: isCollided2D(V, L)

```
Input  : vertices of rectangle V, path L=[$c_{start}, c_{end}$]
Output: True if in collision, otherwise False
1  $V'$ ← MinkowskiDifference($V$, $L$)
2  return WindingNumber((0, 0), V') $> 0$
```

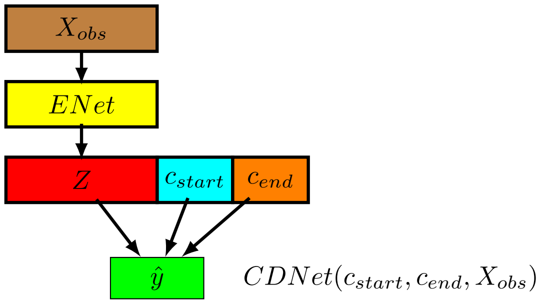


Fig. 5: Collision Detection Network

To train CDNet, I used binary cross entropy as the loss. The dataset for training, validation and test is created based on the environments for MPNet.

The advantage of CDNet is that it allows doing collision detection in parallel, which can boost the performance when the path is complicated.

D. Path Smoothing

Path smoothing is another critical design in Faster NPP. It removes extra waypoints to make the path consist of fewer segments. The simplified path can make the robot motion smoother in practice. What's the most important is that it reduces the number of collision detection in the future path repairs. Hence, it mitigates the affects of the performance bottleneck, collision detection.

Algorithm 8: isCollided3D(V, L)

```
Input  : vertices of rectangle V, path L=[$c_{start}, c_{end}$]
Output: True if in collision, otherwise False
1  $V_{XY}, L_{XY}$ ← XYProjection($V$, $L$)
2  if not isCollided2D($V_{XY}, L_{XY}$) then
3      return False
4  $V_{YZ}, L_{YZ}$ ← YZProjection($V$, $L$)
5  if not isCollided2D($V_{YZ}, L_{YZ}$) then
6      return False
7  $V_{ZX}, L_{ZX}$ ← ZXProjection($V$, $L$)
8  if not isCollided2D($V_{ZX}, L_{ZX}$) then
9      return False
10 return True
```

Algorithm 9: isCollided_CDNet(X_{obs}, L)

```
Input  : obstacle data $X_{obs}$, path $L = [c_1, ..., c_n]$
Output: vector $V$ indicating if any segment is
        in-collision
1  $C_{start} \leftarrow \{c_1, ..., c_{n-1}\}$
2  $C_{end} \leftarrow \{c_2, ..., c_n\}$
3  $V$ ← CDNet($C_{start}, C_{end}, X_{obs}$)
4  return V
```

Algorithm 10: PathSmooth(σ, X_{obs})

```
input  : $\sigma = \{\sigma_1, ..., \sigma_n\}, X_{obs}$
output: smoothed path $\sigma_{new}$ and the vector of path
        segment feasibility $V$
1  $\sigma_{new} \leftarrow \{\sigma_1\}$, $V \leftarrow \varnothing$
2  i ← 0
3  while $i < n - 1$ do
4      $\sigma_{next}$ ← None
5      $i_{next} = i + 1$
6      for $j \leftarrow n - 1 : -1 : i$ do
7          if not isCollided($X_{obs}, [\sigma_i, \sigma_j]$) then
8              $\sigma_{next} \leftarrow \sigma_j$
9              $i_{next} \leftarrow j$
10     if $\sigma_{next}$ is None then
11         $V \leftarrow V \cup \{False\}$
12         $\sigma_{new} \leftarrow \sigma_{new} \cup \{\sigma_{i+1}\}$
13     else
14         $V \leftarrow V \cup \{True\}$
15         $\sigma_{new} \leftarrow \sigma_{new} \cup \{\sigma_{next}\}$
16 return $\sigma_{new}$, $V$
```

IV. Results

A. Dataset

Dataset from [2] is used to train MPNet. S2D is for 2D and R3D is for 3D. Both S2D and R3D contains 110 different environments. For the first 100 environments, the first 4000 paths are used for training and validation and the last 1000 paths are used for test. The left 10 environments are used to test the performance of MPNet and Faster NPP on unseen environments. Each unseen environment contains 2000 paths.

Each Environment in S2D has 7 rectangle obstacles of size 5. And each environment in R3D has 10 box obstacles. The sizes of different boxes vary.

The dataset used to train CDNet is generated based on S2D and R3D. Minkowski difference collision detector is used to generate the ground truth.

B. Train MPNet

To describe a rectangle, we need 4 parameters. In S2D, there are 7 rectangle obstacles. Therefore, the length of the obstacle feature generated by the encoder should be at least 28. Similarly, for R3D, the obstacle feature length should be at least 60.

For S2D, 322167 samples are used for training and 35976 samples are used for validation. As for R3D, the numbers are 323999 and 36000.

MPNet converges after epoch 30 in both S2D and R3D. (Fig. 6)

C. Train and Test CDNet

CDNet converges very fast in either S2D or R3D. The accuracy in validation reaches 100% after only one training epoch. (Fig. (7))

D. Faster NPP: use different Collision Detection Method

The test platform is i7-9700K@5.0GHz and NVIDIA RTX 2070 Super 8GB.

By setting the maximum number of repairs to be 20 and the maximum waypoints used in one repair to be 10 (5×2 because of bi-directional planning), the $FasterNPP : M$ reaches a similar success rate with at least 50% fewer time cost compared with the C++ version of the NPP in [2]. And the $FasterNPP : D$ and the $FasterNPP : C$ are also much faster than the python version of the NPP in [2].

Planner	S2D	R3D
NPP (B) C++ [1]	0.02 (0.02)	0.06 (0.07)
NPP (B) Python	0.26 (0.28)	0.27 (0.17)
Faster NPP: D	0.1052 (0.0963)	0.1433 (0.1262)
Faster NPP: M	0.0099 (0.0097)	0.0070 (0.0060)
Faster NPP: C	0.0274 (0.0286)	0.0810 (0.0574)

TABLE I: Average time cost of successful cases. D: discrete, M: Minkowski, C: CDNet

Planner	S2D	R3D
NPP (B) Python	1.7133 (2.1976)	3.7630 (2.4827)
Faster NPP: D	0.8934 (0.9077)	1.5542 (1.5408)
Faster NPP: M	0.1113 (0.1186)	0.1414 (0.1271)
Faster NPP: C	0.3579 (0.2948)	6.4639 (3.366)

TABLE II: Average time cost of failed cases

By comparing $FasterNPP : D$ with $NPP(B)Python$, we can see that $FasterNPP : D$ is faster even though it also uses discrete collision detection. The path segment feasibility vector and timely path simplification does reduce the burden of collision checking.

Planner	S2D	R3D
NPP (B) Python	0.2894 (0.4288)	1.1252 (0.5478)
Faster NPP: D	0.1400 (0.1356)	0.1862 (0.1624)
Faster NPP: M	0.0141 (0.0138)	0.0121 (0.0100)
Faster NPP: C	0.0513 (0.0482)	0.3360 (0.1766)

TABLE III: Average time cost

Planner	S2D	R3D
NPP (B) [1]	99.30 (98.30)	99.11 (97.76)
NPP (B) Python	97.00 (97.00)	83.00 (77.00)
Faster NPP: D	95.57 (95.15)	96.95 (97.44)
Faster NPP: M	97.13 (95.92)	97.72 (98.22)
Faster NPP: C	92.76 (92.63)	96.05 (96.40)

TABLE IV: Success rate

The $FasterNPP : C$ is not faster than $FasterNPP : M$ in test. This is because the I/O time between CPU and GPU becomes the bottleneck. Besides, another problem of $FasterNPP : C$ is that it requires more repairs to reach higher success rate. The $CDNet$ may mis-classify in-collision and collision-free so that the efficiency of repair is lower than others.

V. Conclusion and Future Work

Faster Neural Path Planner (Faster NPP) is a high-performance path planner based on MPNet and CDNet. The result in this paper is from the test of the Python version of Faster NPP. It is reasonable to expect Faster NPP can become much faster when it is implemented with C++.

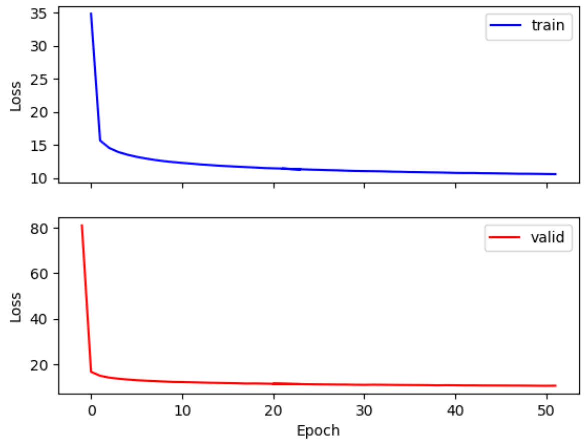
Besides, we should test this planner with more difficult cases and other popular collision detectors to exam its value in real-time application.

References

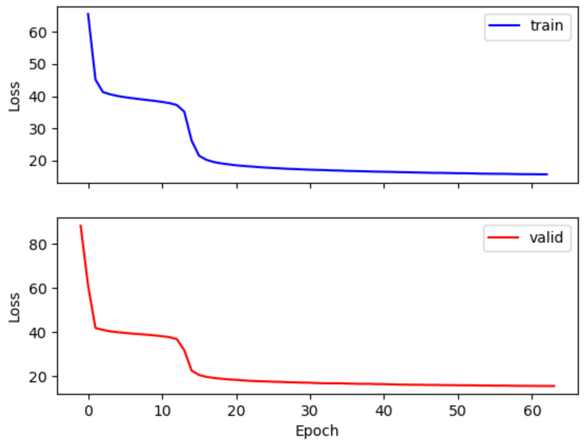
[1] Qureshi, Ahmed H and Yip, Michael C, "Deeply Informed Neural Sampling for Robot Motion Planning", 2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 6582–6588, 2018

[2] Qureshi, Ahmed H and Miao, Yinglong and Simeonov, Anthony and Yip, Michael C, "Motion Planning Networks: Bridging the Gap Between Learning-based and Classical Motion Planners", arXiv preprint arXiv:1907.06013, 2019

[3] Alciatore, D., and Rick Miranda. "A winding number and point-in-polygon algorithm." Glaxo Virtual Anatomy Project Research Report, Department of Mechanical Engineering, Colorado State University (1995).

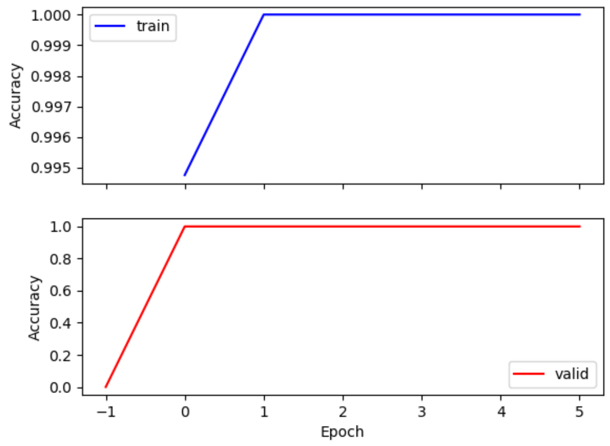


S2D

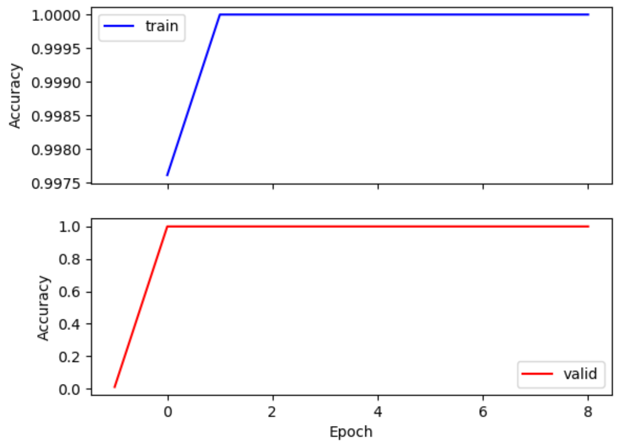


R3D

Fig. 6: Loss curve of MPNet



S2D



R3D

Fig. 7: Accuracy curve of CDNet