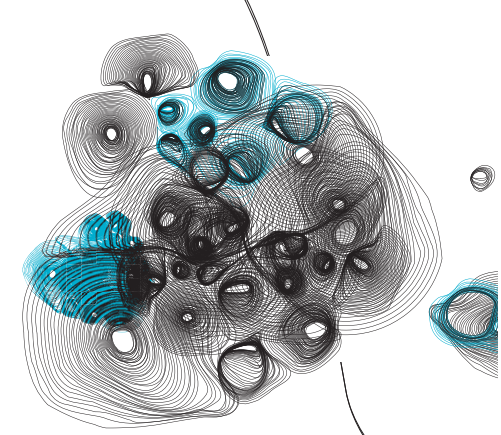


DEVELOPING ADAPTIVE OFFLINE ROBOT PROGRAMMING BASED ON 3D SENSING

INTERNSHIP REPORT (191211208)

Nimish Shah
Digitally signed by Nimish Shah
Date: 2019.12.16 13:04:07 +01'00'

Nimish Shrenik Shah, *s2088894*
M-EE, Robotics and Mechatronics

University of Twente, the Netherlands

Fraunhofer-Institute für Produktionstechnik und Automatisierung IPA, Germany

Supervised by:
dr.ir. F. van der Heijden, EEMCS, Robotics and Mechatronics
Christian Landgraf, M. Sc., Robotics and Assistive Systems

Christian Landgraf
Digital unterschrieben von Christian Landgraf
Datum: 2019.12.18 17:34:35 +01'00'

UNIVERSITY OF TWENTE.

Fraunhofer IPA

DECEMBER 16, 2019

Contents

List of Figures

List of Algorithms and Acronyms

ENet Encoder Network

GPU Graphics Processing Unit

MPNet Motion Planning Network

PNet Planning Network

PRM Probabilistic Road Map

RRT Rapidly-exploring Random Tree

SMP Sampling-based Motion Planner

Chapter 1

Introduction

Today, in the era of industrialization the importance of robots in industry, be it small-scale or large-scale has dramatically increased. With the advent of current technologies robots are becoming more common in our day-to-day life as well as production and manufacturing. But, still problem of planning the complete, efficient and collision-free path remains the area of active research in the field of mobile and articulated robotics.

Many classical algorithms exist to solve the problem of path planning but, they are, computationally expensive and struggle to upscale with increased degrees of freedom, as search space explodes exponentially.

The swift rise in computational capabilities of both hardware and software, such as Graphics Processing Unit (GPU), parallel computing, new open software platforms and availability of massive labelled data sets allows us to exploit computational power and available data to research modern motion planning techniques such as *Neural Planning*. These techniques take advantage of machine learning algorithms to dynamically reduce the time required to plan near-optimal paths [1], [2].

In this report, MPNet—a motion planning network, the technique employed by Qureshi et al. is described, the majority of references for these algorithms are taken from [3], [1] and [2]. This network receives environment information as point-clouds, as well as the robot's initial and desired goal configurations and recursively calls itself to bidirectionally generate connectable paths. MPNet is implemented using *Tensorflow*–an end-to-end open-source machine learning platform with *Python*.

1.1 Motion Planning

The problem of motion planning is central to robotics and artificial intelligence, be it a semi-autonomous robot or a fully-autonomous vehicle. It aims to find a collision-free, low-cost path connecting start and goal configuration for an agent.

An ideal motion planning algorithm for solving real-world problem should offer following key features: [2]

1. **completeness and optimality guarantee:** a solution will be found if one exists and the solution will be globally optimal

2. **computational efficiency:** finding a solution in either real-time or in sub-second times or better while being memory efficient

3. **insensitivity to environment complexity:** the algorithm is effective and efficient in finding solutions regardless of the constraints of the environment

1.2 Organization

This report is organized as follows. Chapter 2 shortly reviews the traditional and state-of-the-art algorithms in motion planning. Chapter **??** provides a basic foundation of deep learning keeping TensorFlow as an framework in mind. Chapter 4 formally introduces and describes the problem statement. Chapter 5 describes notations required to outline network architecture, propositions, approaches to model the network and frameworks used. Chapter 6 outlines results and chapters 7 and 8 briefly compares the results with results provided in the literature reviewed, provides recommendations and sets the stage for further research respectively. The Chapter 9 gives very short information about side tasks performed during the internship. These tasks are not explained in detail as they are not main focus of this internship.

Chapter 2

Motion Planning Algorithms

This chapter shortly reviews the classical and modern (neural) motion planning algorithms.

2.1 Classical Algorithms

Years of research in the field of motion planning has developed a variety of methods, leveraging concepts from interdisciplinary subjects. This has led to developments of methods such as *Combinatorial Planning*, *Control-based Methods*, *Potential Fields* and *Sampling-based Motion Planners (SMPs)*. This subsection provides a brief introduction to these methods.

2.1.1 Combinatorial Planning

Combinatorial approaches to motion planning find paths through the continuous configuration space without resorting to approximations. Due to this property, they are alternatively referred to as exact algorithms [4]. This class of techniques involves algorithms such as visibility graphs, Voronoi diagrams, exact cell decomposition, approximate cell decomposition etc.

2.1.2 Control-based Methods

These methods model equations model the equations of motion, system dynamics and take advantage of control theory to navigate a system along desired trajectory. These approaches also employ feedback loops for error minimization and feed-forward techniques based upon system and environment knowledge to improve performance. Hence, using these methods navigation can be done in online manner but due to complex dynamics and/or cluttered environments valid motions are heavily restricted [5].

2.1.3 Potential Fields Methods

The robot, represented as a point in $\mathcal{X}$, the configuration space, is modelled as a particle under the influence of a artificial potential field U. This field superimposes *repulsive forces* from obstacles

and *attractive force* from goal. Hence, potential function can be written as

$$U(q) = U_{\text{att}}(q) + U_{\text{rep}}(q)$$
$$\vec{F}(q) = -\vec{\nabla}U(q)$$

After finding force $\vec{F}(q)$, *gradient descent* is performed in the direction of $\vec{F}(q)$. This method has drawbacks such as getting stuck in the local minima, representational cost etc. Further, different policies can be employed for local-minima-free navigation.

2.1.4 Sampling-based Motion Planners

The main idea is to avoid the explicit construction of $\mathcal{X}_{\text{obs}}$, and instead conduct a search that probes the C-space with a sampling scheme. This probing is enabled by a collision detection module, which the motion planning algorithm considers as a *black box*. This enables the development of planning algorithms that are independent of the particular geometric models. This general philosophy (fig. 2.1) has been very successful in recent years for solving problems from robotics, manufacturing, and biological applications that involve thousands and even millions of geometric primitives. Such problems would be practically impossible to solve using techniques that explicitly represent $\mathcal{X}_{\text{obs}}$ [4]. Furthermore, as mentioned, these methods are *probabilistically complete*.

In short, SMP avoids the explicit representation of the state space and instead maintains an implicit representation of the $\mathcal{X}$ by three basic parts of the algorithm viz., *sampling*, *steering* and *collision checking*. This representation is global and becomes more accurate as number of samples are added.

The motivation behind randomized path planning is a limitation of search-based path planning to lower-dimensional configuration spaces due to the very high computational requirement. Hence, the trade-off between search based and randomized methods is that they are incomplete, but will find a solution with any probability given sufficient running time. These methods converge quickly in practice and are simple enough to yield consistent behaviour [6].

Randomized path planning algorithms are designed in two contexts:

1. **Single Query Planning:** it is assumed that a single path planning problem must be solved quickly, without any preprocessing.
2. **Multiple Query Planning:** it is assumed that many path planning problems will be solved for the same environment. Hence, it is logical to store it in a suitable data structure to allow fast path planning queries.

Some examples of methods of this class are RRT, RRT* and Probabilistic Road Maps (PRMs). We take the liberty to discuss RRT and RRT* in detail as it is used for *expert demonstration* and replanning stages of the MPNet algorithm developed by Qureshi et al.. These algorithms are chosen for expert demonstration due to their probabilistic completeness.

2.1.5 RRT

It is a single query planning method introduced as an efficient data-structure and a sampling strategy to quickly search non-convex, high dimensional spaces with both algebraic (due to

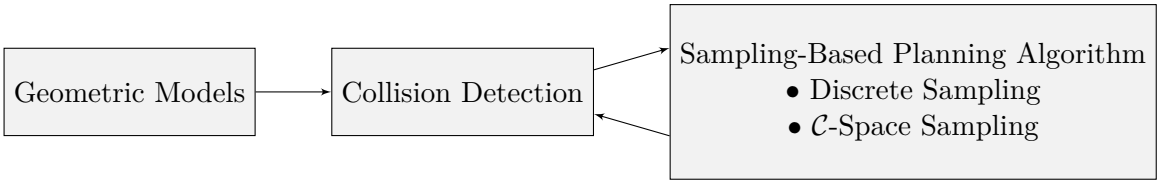


Figure 2.1: Sampling-based planning philosophy uses collision detection as *black-box* that separates the motion planning from the particular geometric and kinematic models.

obstacles) and differential (due to non-holonomy and robot dynamics) constraints by randomly building a tree.

In its basic version, the algorithm incrementally builds a tree of feasible trajectories, rooted at the initial condition. An outline of the algorithm is given in Algorithm 2.1. The algorithm is initialized with a graph that includes the initial state as its single vertex, and no edges. At each iteration, a point $q_{rand} \in \mathcal{C}_{free}$ is sampled. An attempt is made to connect the nearest vertex q_{near} in the tree to the new sample. If such a connection is successful, q_{rand} is added to the vertex set, and $(q_{near}; q_{rand})$ is added to the edge set. In the original version of this algorithm, the iteration is stopped as soon as the tree contains a node in the goal region [7].

Algorithm 2.1: Algorithm for RRT; adapted from [8]

Input: q_0, K, Δt
Output: τ

```
τ.init(q_0);
for k = 1 to K do
    q_rand ← RANDOM_STATE();
    q_near ← NEAREST_NEIGHBOUR(q_rand, τ);
    u ← SELECT_INPUT(q_rand, q_near);
    q_new ← NEW_STATE(q_near, u, Δt);
    τ.add_vertex(q_new);
    τ.add_edge(q_near, q_new, u);
end
```

2.1.6 RRT* Algorithm

It is an optimal variant of the RRT which asymptotically guarantees to find the shortest path if one exists. This is done by maintaining tree structure rather than graph structure in the memory which is memory efficient and allows relatively easy adaptations for motion planning with differential constraints, or coping with modelling errors.

Variety of cost functions can be used to design optimal RRT* algorithm on a case-to-case basis.

Furthermore, biasing a sample distribution towards the unexplored portions and high probability path finding space with heuristics can improve the performance of such single-query methods. In the case of RRT*, we can avoid the formation of cycles, which removes redundant edges.

2.2 State of the Art

Deep Learning, in which deep neural networks (DNNs), i.e., networks with serveral layers, take advantage of huge datasets to progressively extract higher-level features from the raw input. Deep learning has changed the entire landscape over the past few years. Every day, there are more applications that rely on deep learning techniques in fields as diverse as healthcare, finance, human resources, retail, earthquake detection, and self-driving cars. As for existing applications, the results have been steadily improving.

Qureshi and Yip in their work [3], [1] and [2] outline the importance of cross-fertilisation of machine-learning algorithms with motion planning techniques to solve planning problems. According to them, these hybrid algorithms provides all key features of an ideal planner mentioned in section 1.1. Qureshi et al. 2 review many so called *neural planning* algorithms developed over the period.

MPNet

Here, we briefly introduce the MPNet neural motion planner developed in [2]. In depth description is available in Chapter 5.

MPNet is a deep neural network-based bidirectional iterative planning algorithm that comprises two modules, an encoder network and planning network. The encoder network takes point-cloud data of the environment, such as the raw output of a depth camera or LIDAR, and embeds them into a latent space. The planning network takes the environment encoding, robot's current and goal state, and outputs a next state of the robot that would lead it closer to the goal region [2].

One of the significant feature of the path planned by MPNet is a near optimal path with minimal to no branching. Three learning strategies are mentioned in the literate. We are interested in second and third strategy.

Learning Strategies:

1. **offline batch learning**—assumes the availability of all training data and no expert demonstration, hence, no learning from innovative experience provided by expert demonstration
2. **continual learning**—assumes that the expert demonstrations come in streams and the global training data distribution is unknown , uses episodic memory
3. **active continual learning**—incorporates MPNet into the learning process and asks for expert demonstrations only when needed

For mathematical details and implementation [9] can be referred.

Chapter 3

Deep Learning: A Primer

Deep learning is a specialized form of machine learning. A machine learning workflow starts with relevant features being manually extracted from images. The features are then used to create a model that categorizes the objects in the image. With a deep learning workflow, relevant features are automatically extracted from images. In addition, deep learning performs "end-to-end learning" – where a network is given raw data and a task to perform, such as classification, and it learns how to do this automatically.

Another key difference is deep learning algorithms scale with data, whereas shallow learning converges. Shallow learning refers to machine learning methods that plateau at a certain level of performance when you add more examples and training data to the network.

A key advantage of deep learning networks is that they often continue to improve as the size of your data increases.

3.1 Neural Networks

Neural Networks are inspired by biological concept of neuron (perceptron), which is a non-linear primary functional unit of the nervous system. Such non-linear neurons can be combined in the layers, with or without same activation. This Multi-Layer Perceptron structure can be use to learn complex functions. Number of such layers defines the length of the network. On this basis network can be classified as *shallow* or *deep*.

Deep vs. Shallow Networks

Shallow neural network consists of a single hidden layer and as name suggests deep neural network consists of more than one hidden layers. With multiple layers, each layer can create more and more abstract features/concepts. As the we move forward through the layers i.e., deeper and deeper, the features become more abstract, from simple line detectors to various types of moving objects.

3.2 Layers

Layer is the set of neuron at a same level. Layers can be of different types depending upon operations they perform. Following subsection discusses different layers made available by Keras.

Dense

Neurons in this layer are fully or densely connected to the neurons in the next layer. This layer implements the operation $\boldsymbol{y} = g(\boldsymbol{x}) + b$, where $g(\cdot)$ is an activation function and b is a bias.

Activation

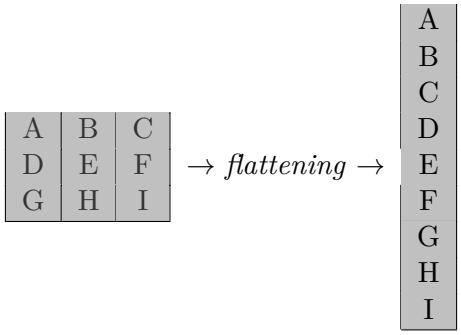
This layer applies an activation function $g(\cdot)$ to an output. Hence, the dimensions of the output of the layer is same as the input of the layer. Different types of activations are defined in section 3.3.

Dropout

This layer applies Dropout to the input. Dropout consists in randomly setting a fraction rate of input units to 0 at each update during training time, which helps prevent overfitting. It is important to note that dropout is not applied during testing or prediction phase.

Flatten

This layer flattens the input and does not affect the batch size. This can be thought as unrolling the multi-dimensional input into single dimension. Following figure shows the flattening process.



Convolutional

These layers perform convolution operation on the input. These layers can be used to perform 1D convolutions such as temporal, 2D convolutions such as spatial convolution over images etc. In Keras layers such as `Conv1D`, `Conv2D` or even `Conv3D` to perform spatial convolution over volumes are available.

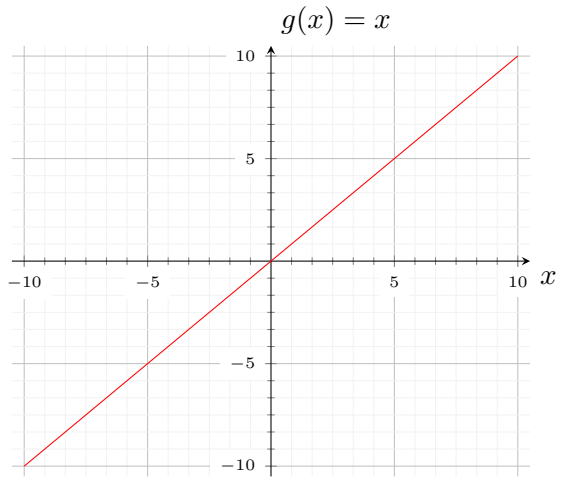
Pooling

The pooling layer serves to progressively reduce the spatial size of the representation, to reduce the number of parameters and amount of computation in the network, and hence to also control overfitting. The intuition is that the exact location of a feature is less important than its rough location relative to other features.

Different type of pooling mechanisms such as max pooling, average pooling are available in 1D, 2D and 3D forms.

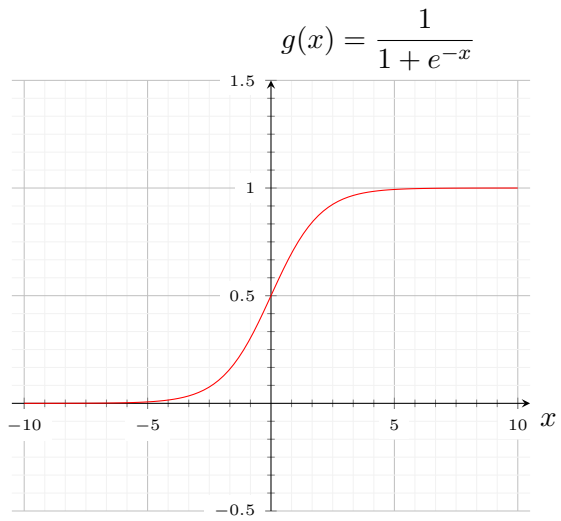
3.3 Activation Functions

Linear



Output of the linear activation function is same as an input. Hence, essentially it is an identity function. It is generally used in output layer of the regression problem.

Sigmoid



Adjoining figure shows sigmoid non-linearity. It takes a real-valued number and *squashes* it into range between 0 and 1. In particular, large negative numbers become 0 and large positive numbers become 1.
There are two issues with sigmoid neurons, i) *Sigmoids saturate and kill gradients*—a very undesirable property of the sigmoid neuron is that when the neuron's activation saturates at either tail of 0 or 1, the gradient at these regions is almost zero, and ii) *Sigmoid outputs are not zero-centred*—this is undesirable since neurons in later layers of processing in a Neural Network (more on this soon) would be receiving data that is not zero-centred [10].

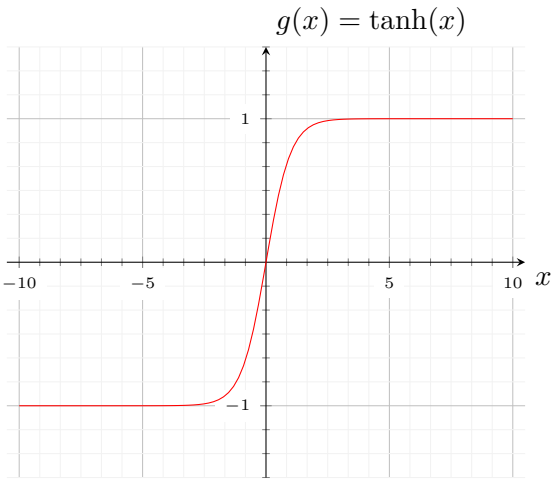
Softmax

It is used to compute probability distribution from a vector of real numbers. The Softmax function produces an output which is a range of values between 0 and 1, with the sum of the probabilities been equal to 1. Mathematically,

$$g(\boldsymbol{x})_i = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

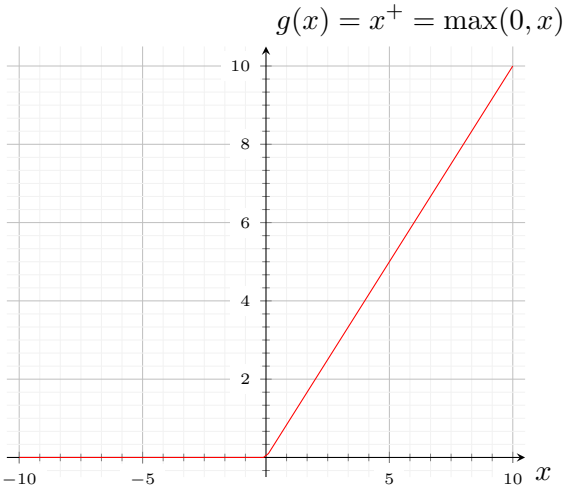
The Softmax function is used in multi-class models where it returns probabilities of each class, with the target class having the highest probability. The Softmax function mostly appears in almost all the output layers of the classification problems. The main difference between the Sigmoid and Softmax is that the Sigmoid is used in binary classification while the Softmax is used for multivariate classification tasks.

tanh



This activation function squashes a real-valued number to the range $[-1, 1]$. Like the sigmoid neuron, its activations saturate, but unlike the sigmoid neuron its output is zero-centred. Therefore, in practice the tanh non-linearity is always preferred to the sigmoid non-linearity [10].

ReLU

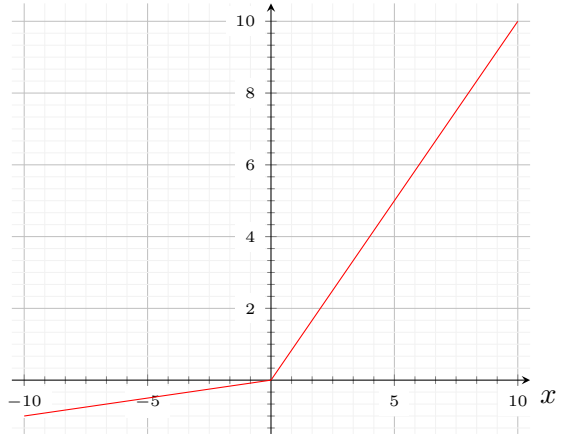


Recently the Rectified Linear Unit (ReLU) has become very popular in the recent years. The activation is simply thresholded at zero. It has several pros and cons [10]:

- \+ greatly accelerates the convergence of SGD as compared to the sigmoid/tanh functions. It is argued that this is due to its linear, non-saturating form.
- \+ compared to tanh/sigmoid neurons that involve expensive operations (exponentials, etc.), the ReLU can be implemented by simply thresholding a matrix of activations at zero.
- – unfortunately, ReLU units can be fragile during training and can *die*.

PReLU

$$g(x) = \begin{cases} x & \text{if } x > 0, \\ ax & \text{otherwise.} \end{cases}$$



Parametric ReLUs are one attempt to fix the *dying ReLU* problem. Instead of the function being zero when $x < 0$, a PReLU will instead have a small negative slope a which is a trainable parameter for each neuron. Further, this parameter can also be shared by neurons along the axes.

3.4 Optimizers

The loss function lets us quantify the quality of any particular set of weights $\boldsymbol{w}$. The goal of optimization is to find $\boldsymbol{w}$ that minimizes the loss function. Basic optimizers such as Batch Gradient Descent, Stochastic Gradient Descent and more advanced optimizers which consider parameters such as momentum. RMSProp (Root Mean Square Propagation, considers momentum), Adagrad (Adaptive Gradient Descent, a combination of RMSProp and SGD) and Adam are available.

3.5 Regularization

Regularization is used in machine learning models to cope with the problem of overfitting. The idea behind the regularization is to penalize and make some parameters small to make hypothesis simpler and less prone to overfitting.

L1 and L2 Regularization

In L1 norm we shrink the parameters to zero. When input features have weights closer to zero that leads to sparse L1 norm. In Sparse solution majority of the input features have zero weights and very few features have non zero weights. L1 regularization does feature selection. It does this by assigning insignificant input features with zero weight and useful features with a non zero weight.

Mathematically,

$$\mathcal{L} = L(\boldsymbol{x}, g(f(\boldsymbol{x}))) + \lambda \sum_i |\boldsymbol{w}_i|$$

L2 regularization forces the weights to be small but does not make them zero and does non

sparse solution. L2 is not robust to outliers as square terms blows up the error differences of the outliers and the regularization term tries to fix it by penalizing the weights

Mathematically,

$$\mathcal{L} = L(\boldsymbol{x}, g(f(\boldsymbol{x}))) + \lambda \sum_i \boldsymbol{w}_i^2$$

Early Stopping

When training a large network, there is be a point during training when the model stops generalizing and starts learning the statistical noise in the training dataset. This overfitting over the training dataset results in an increase in generalization error, making the model less useful at making predictions on new data. At this point, validation error starts increasing instead of decreasing. If training is stopped at such point, overfitting can be avoided. This technique is called early stopping.

Droupout

The idea behind dropout is to randomly drop units (logits) with their connection from the network during training phase. Dropping the units prevents too much co-adapting. Research by Srivastava et al. has shown that dropout significantly reduces overfitting and gives major improvements over other regularization methods.

3.6 Autoencoders

Autoencoders may be thought of as being a special case of feed-forward networks which are trained to attempt copy its input as an output. Autoencoder generally consists of two components an encoder and a decoder. It has a hidden layer $\boldsymbol{Z} = f(\boldsymbol{x})$, which describes the code used to represent the input. The representation of the input in this layer $\boldsymbol{Z}$ is called *latent space encoding*. Further, decoder function decodes the coded $\boldsymbol{Z}$, to reconstruct the input $\boldsymbol{x}$ such that, $\hat{\boldsymbol{x}} = g(\boldsymbol{Z})$.

It is important to note that $\hat{\boldsymbol{x}} \approx \boldsymbol{x}$, as autoencoders are designed in such a way that they do not exactly copy the inputs, but they copy the only approximately by prioritizing which aspects of the input should be copied. Hence, it often learns useful properties of the data. Fig. 4.1 shows a generic diagram of the autoencoder.

Autoencodes are widely used for dimensionality reduction or feature learning. Now recently, they are also being used as forefront of Generative Adversarial Networks (GAN). These type of neural networks belong to class of *unsupervised learning*.

Contractive Autoencoders

Both undercomplete and overcomplete autoencoders do not learn the most salient features of the input distribution if they are given too much freedom. This freedom can be regulated using different means of regularizers for e.g., sparse encoders, denoising and penalizing.

Contractive autoencoders are regularized by means of penalty $\boldsymbol{\Omega}$ such that, loss function becomes

$$\mathcal{L} = L(\boldsymbol{x}, g(f(\boldsymbol{x}))) + \boldsymbol{\Omega}(\boldsymbol{Z}, \boldsymbol{x})$$

where,

$$\boldsymbol{\Omega}(\boldsymbol{Z}, \boldsymbol{x}) = \lambda \sum_i \|\boldsymbol{\nabla}_{\boldsymbol{x}} Z_i\|^2$$

This forces the model to learn a function that does not change much when $\boldsymbol{x}$ changes slightly. Because this penalty is applied only at training examples, it forces the autoencoder to learn features that capture information about the training distribution.

Rifai et al. define this penalty term in [12] as,

$$\boldsymbol{\Omega}(\boldsymbol{Z}) = \lambda \left\| \frac{\partial f(\boldsymbol{x})}{\partial \boldsymbol{x}} \right\|_F^2$$

The penalty term is the Frobenius norm of the jacobian matrix, which is the sum squared over all elements inside the matrix.

3.7 Episodic Memory

In most of the supervised learning methods examples are drawn from the *training dataset* $D_{tr} = \{(x_i, y_i)\}_{i=1}^n$, where (x_i, y_i) is a single *training example* which is composed of *feature vector* $x_i \in \boldsymbol{X}$ and target vector $y_i \in \boldsymbol{Y}$. Each training example is an identically and independently (iid) distributed sample.

The aim of supervised learning is to construct a model $f : \boldsymbol{X} \to \boldsymbol{Y}$. To accomplish this principle of Empirical Risk Minimization (ERM) is employed and f is found by minimizing the loss l over $f(x)$ and y.

ERM is a major simplification from what human learning. Learning humans observe data as an ordered sequence and rarely observe same example twice and can only memorize few pieces of data, and sequence concerning different tasks. Hence, iid assumption, along with ERM falls apart. Therefore, straightforward applications of ERM lead to *catastrophic forgetting*. That is, learner forgets how to solve past tasks after it is exposed to new tasks [9].

Hence, *continual learning* can be used to lessen this side-effect. Continual learning is the ability of a model to learn continually from a stream of data, building on what was learnt previously, hence exhibiting positive transfer, as well as being able to remember previously seen tasks.

In following paragraphs briefly discuss about continual learning based on Gradient Episodic Memory (GEM) as per Lopez-Paz and Ranzato.

GEM is a model that learns over the continuum of data which mitigates the problem of *catastrophic forgetting* while transferring the beneficial knowledge to the past. One of the main features of the GEM is an *episodic memory* $\mathcal{M}_t$, which stores a subset of the observed examples from task t.

While learning memory is populated with recent m examples from each task t_i using suitable memory update strategies, hence, loss l becomes the function of $f(x)$, y and episodic memory $\mathcal{M}$ for that task.

The performance of the learner can be measured on the following metrics:

Backward Transfer it is the influence that learning a task t has on the performance of the task k, such that, $k \prec t$. It is interesting to note that large negative backward transfer is also known as catastrophic forgetting.

Forward Transfer it is the influence that learning a task t has on the performance of the task k, such that, $k \succ t$. *zero-shot learning* can be an example of positive forward transfer.

Hence, the goal of the GEM is to minimize negative backward transfer and/or increasing positive forward transfer.

Chapter 4

Problem Analysis

Optimal, efficient and intuitive robotic programming is still a challenge in robotic manufacturing and one of the main reasons why robots are not widely implemented in small and medium-sized enterprises (SME). In order to effectively and efficiently respond to the current product variability requirements, SMEs require easy and optimal programmable robotic manufacturing systems in order to achieve profitable and rapid changeover [13].

The current implementation of optimal motion planner at Fraunhofer IPA is based on the Product, Process and Resource (PPR) model.

The PPR approach proposes interpretation of the manufacturing process by semantically describing and mathematically modelling *the product*, *the process*, *the resource* (a robot) and a manufacturing motion cost, which determines the relation between robot motions and process quality. The interpretation determines the automatic configuration of an Rapidly exploring Random Trees, a sample-based algorithm used for the computation of optimal motions when uncertainties are present. Further product, process and resource mathematical models could be integrated for computing optimal motions for other manufacturing processes. For instance, stiffness model of industrial robots could be embedded as motion costs for optimizing motions for machining in joint space allowing easy reconfiguration of the model.

4.1 Problem Statement

As *deep learning* is the new skill in town, aim of the internship to have a proof of concept implementation of the deep learning algorithm(s) to *find an optimal collision-free path considering kinematics and industrial process constraints*. For this purpose, MPNet–a motion planning network developed by Qureshi et al. in [3], [1] and [2] was chosen.

4.2 Development Plan

While implementing following iterations are considered.

Iteration 1 Simple-2D environment with point robot without continual learning

- This will help to set up the environment and understand underlying components of the algorithm as well as getting acquainted with TensorFlow

Iteration 2 3D environment with a rigid robot without continual learning

- extension of above use case and base for next step
- only consider end effector and collisions due to inverse kinematics shall be ignored

Iteration 3 3D environment with a rigid robot without continual learning and robot kinematics

- 3D environment and interfacing with robot SDK to take inverse kinematics in account without continual learning
- Use-case 2 shall be extended with some major modifications to adapt to robot SDK and planner

Lastly, whole environment is to be adapted for continual learning.

4.3 Analysis

Qureshi et al. in their work outline the importance of cross-fertilization of machine-learning algorithms with sample-based motion planning (SMP) techniques to solve planning problems. The core idea behind the approach is, to learn how to plan a path from data samples provided as a training set. This training set contains environment/obstacle point clouds, paths, goal and target configurations. Hence, time required to plan the collision-free, optimal path in the sense of same optimality criterion (as that on training examples) will be considerably small.

Point clouds can be stored in Point Cloud Data (PCD) format.[1] A path is represented as set of join-configurations whose dimensionality depends upon degree of freedom of the robot and length of the path. In case of point robot case we these paths can be current position of the robot. As all the points present in point clouds are considered as obstacles and empty space between the points is considered as free space, it is important that resolution of the PCD should be sufficiently larger compared to size of the end-effector or gripper.

It is also important to note that most machine- or deep- learning algorithms require a fixed input size. Point clouds normally have variable sizes making them not machine-learning friendly. However sampling or simplification algorithms can be used to standardize point clouds for these tasks. Here, standardizing point clouds mean making sure each point cloud has same number of points, hence same number of inputs to network. Hence, we require to define number of point in point clouds, which directly affects the number of inputs for MPNet.

While analysing the solution few following advantages and disadvantages over traditional methods were found.

4.3.1 Advantages

1. Significant reduction in time required to plan the path once network is trained.
2. If network fails to plan the path, the network can be augmented by classical/SMP-based techniques to plan the path for that particular area. This innovative knowledge of the new plan can be used again to train the network.

[1] description of file format encoding is available at http://pointclouds.org/documentation/tutorials/pcd_file_format.php

4.3.2 Disadvantages

1. Planning is not done directly considering robot kinematics but they are inherent because paths present in training dataset are found by respecting these constraints.
2. Highly specialized to single robot geometry, type and configuration and can not be directly used to new type of robot unless retrained using dataset specifically designed for the same.

4.4 The MPNet Model

MPNet is a deep neural-neural network based iterative, recursive and bidirectional planning algorithm which comprises of two separate neural networks viz., *ENet*, the encoder network and *PNet*, the planning network. That is, MPNet uses neural models to plan forward path, from start to goal, as well as to plan backward, from goal to start, until both paths meet each other. For connecting forward and backward paths some greedy heuristics such as RRT-Connect can be used.

Further it is Recursive Divide-and-Conquer Planning as it begins with global planning which results in critical states that are vital to generating feasible trajectory. If any of the consecutive critical nodes in the global path are not connectable, MPNet takes them as a new start and goal and recursively plans a motion between them.

Being a neural network based approach, it can suffer from *catastrophic forgetting* when data is given in streams or in sequence [14]. Hence continual learning approach based on episodic memory as discussed in section 3.7 can be employed. In subsequent subsections, we briefly discuss the uses of ENet and PNet with machine learning pipeline.

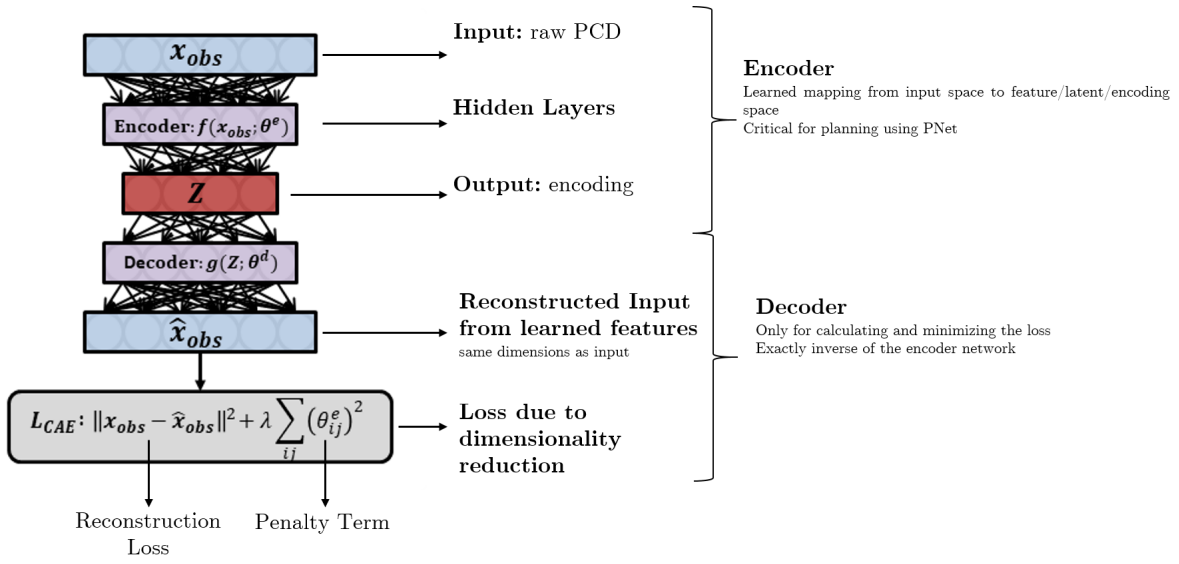


Figure 4.1: ENet, annotated on image from [3]

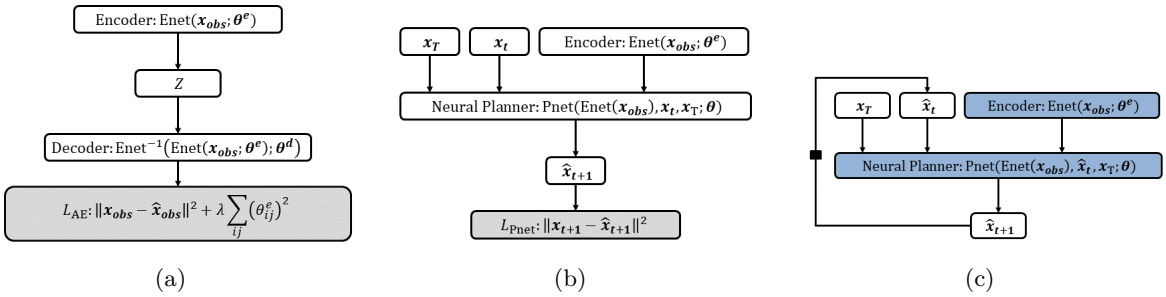


Figure 4.2: Block diagrams of: (a) ENet; (b) PNet; and (c) MPNet—online, adapted from [1]

4.4.1 ENet: The Encoder Network

ENet is a Contractive Auto-encoder (refer section 3.6) network which can be trained in two ways i) independently or ii) end-to-end with PNet. ENet takes flattened version of raw PCD as an input and outputs a latent space encoding **Z** corresponding to the input. Figure 4.1 gives the intuitive idea of the working and fig. 4.2(a) shows the block digram of the network.

If Enet is trained independently it can be regarded as supervised learning as we can compare the reconstructed environment with the original environment. But, in the case of end-to-end training can be regarded as unsupervised training due to the fact that loss from PNet is backpropagated to the ENet. Here, we consider the case of independent training. Further, it is also important to note that, while implementing continual leaning we need to train ENet and PNet in an end-to-end fashion.

4.4.2 PNet: The Planning Network

PNet is a feed-forward neural network which takes latent space encoding **Z** with current state x_t and goal state x_T as an input and predicts next robot state $\hat{x}_{t+1}$. Furthermore, architecture of PNet also contains dropout to add the stochasticity. Figure 4.2(b) shows block diagram of the PNet.

Figure 4.2(c) shows complete block diagram of the *neural planner* heuristic algorithm which uses PNet to find the path between current start and goal states.

4.4.3 Neural Planner

Neural planner is an incremental bidirectional path generation heuristic which takes latent space representation **Z** of obstacles along with start and goal state as an input and outputs a path connecting the given states. See Alg. 4.1

Here, τ is the *coarse path*, composed of sequence of configurations. It generates two paths one from start τ^a and one from goal τ^b incrementally marching towards each other. After each step planner attempts to connect both paths using some heuristic (straight line in case of 2D world), if possible, using function `steerTo`. in case both paths τ^a and τ^b are connectable, algorithm returns a concatenated path $\tau = \tau^a \cup \tau^b$.

Following subsections describes different components of MPNet algorithm.

Global Planning

Global planning is done by iteratively calling neural planning algorithm. Path planned in this step is referred as *coarse path*.

Replanning

Replanning algorithm can be used in two modes *i) Neural Planning* and *ii) Hybrid Planning*. As mentioned earlier NeuralPlanner outputs the coarse path, if all consecutive nodes in τ are connectable, i.e., the trajectories connecting them (consecutive configurations) are in obstacle-free space then there is no need of replanning. If there are any beacon nodes in τ then there is a need to connect them. Beacon states is referred to states that are not directly connectable as the connecting trajectory passes through the obstacles (yellow dots in fig. 4.3(a)). To replan, the beacon nodes are presented to the replanner as start and goal pair together with the obstacle information i.e., encoded obstacle space $\mathbf{Z}$. Oracle Planner is any other path planning algorithm, for e.g., RRT*. In this implementation RRT* was preferred as it was used while generation the training set data. Refer alg. 4.2. Parameter p in the algorithm is to select the planning mode.

1. *Neural Replanning:* It takes beacon states and makes a limited number of attempts to find a path solution between them using `NeuralPlanner`. In case of neural planning p is zero.
2. *Hybrid Replanning* It combines neural planning and classical/oracle planning techniques. Further it tries to find a path using neural planner until specific number of attempts N_r, unless found one. Else uses oracle planner as defined previously.

Steering and IsFeasible

Check feasibility of connecting two consecutive states and checks whether they lie entirely in obstacle-free space.

Lazy States Contraction

LSC is a smoothing technique. Hence, it tries to make path more smooth and with few joint movements. It is *not an essential* component of the proposed method. Its inclusion helps to generate *near-optimal* paths. This method improves the computational efficiency as the algorithm will have to process fewer nodes during planning. This is run after every planning step.

Fig. 4.3 shows all the components in an example sweep of the MPNet algorithm (see alg. 4.3).

Algorithm 4.1: Algorithm for Neural Planner [1]

Input: $x_{\mathbf{start}}$, $x_{\mathbf{goal}}$, $\mathbf{Z}$
Output: τ

$\tau^a \leftarrow x_{\text{start}}$;
$\tau^b \leftarrow x_{\text{goal}}$;
$\tau \leftarrow \varnothing$;
Reached $\leftarrow$ False;
for $i \leftarrow 0$ **to** N **do**
  $x_{\text{start}} \leftarrow \text{PNet}(\mathbf{Z}, \tau^a.\text{end}, \tau^b.\text{end})$;
  $\tau^a \leftarrow \tau^a \cup \{x_{\mathbf{start}}\}$;
  Connect $\leftarrow$`steerTo`$(\tau^a.\text{end}, \tau^b.\text{end})$;
  if *Connect* **then**
    $\tau \leftarrow \text{concatenate}(\tau^a, \tau^b)$;
    return τ
  end
  swap(τ^a, τ^b);
end
return $\varnothing$;

Algorithm 4.2: Algorithm for Replanning [1]

Input: τ, $\mathbf{Z}$
Output: τ_{new}

$\tau_{new} \leftarrow \varnothing$;
for $i \leftarrow 0$ **to** $size(\tau) - 1$ **do**
  if *steerTo*(τ_i, τ_{i+1}) **then**
    $\tau_{new} \leftarrow \tau_{new} \cup \{\tau_i, \tau_{i+1}\}$;
  else
    if $p == 0$ **then**
      $\tau' \leftarrow$`NeuralPlanner`$(\tau_i, \tau_{i+1}, \mathbf{Z})$
    else
      $\tau' \leftarrow \text{OraclePlanner}(\tau_i, \tau_{i+1}, x_{obs})$
    end
    if τ' **then**
      $\tau_{new} \leftarrow \tau_{new} \cup \tau'$;
    else
      return $\varnothing$;
    end
  end
end
return τ_{new};

Algorithm 4.3: Algorithm for MPNet [1]

Input: $x_{\mathbf{start}}$, $x_{\mathbf{goal}}$, $x_{\mathbf{obs}}$
Output: τ

$\mathbf{Z} \leftarrow$ `ENet`$(x_{\mathbf{obs}})$;
$\tau \leftarrow$`NeuralPlanner`$(x_{\mathbf{start}}, x_{\mathbf{goal}}, \mathbf{Z})$;
$\tau \leftarrow$`LazyStatesContraction`(τ);
if *IsFeasible*(τ) **then**
  return τ;
else
  p = 0 ; `// set planner to use NeuralPlanner`
  for $i \leftarrow 0$ **to** N_r **do**
    $\tau \leftarrow$`Replan`$(\tau, \mathbf{Z}, p)$;
    $\tau \leftarrow$`LazyStatesContraction`(τ);
    if *IsFeasible*(τ) **then**
      return τ;
    end
  end
  p = 1 ; `// set replanner to use oracle planner`
  $\tau \leftarrow$`Replan`$(\tau, \mathbf{Z}, p)$;
  $\tau \leftarrow$`LazyStatesContraction`(τ);
  if *IsFeasible*(τ) **then**
    return τ;
  end
  return $\varnothing$;
end

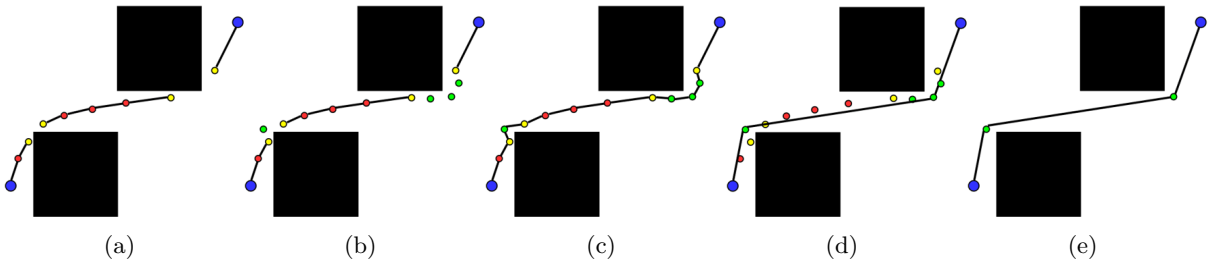


Figure 4.3: Images depicting example sweep of the MPNet: (a) Global planning; (b) Neural-Replanning; (c) Steering; (d) Lazy States Contraction; and (e) Optimized Path, image adapted from [1]

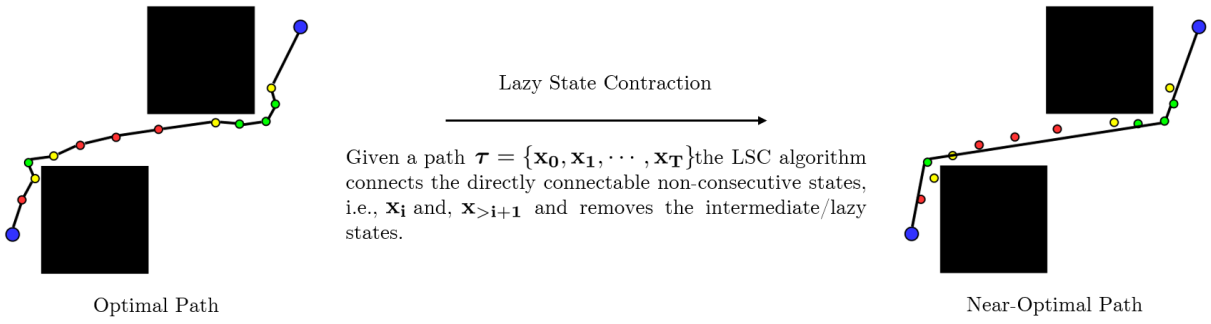


Figure 4.4: Lazy States Contraction explained, annotated on image adapted from [1]

Chapter 5

Approach and Methods

This chapter briefly describes model architecture of the MPNet, its implementation and training setup for deep learning task. The case of Iteration 1 is discussed below.

All the models were programmed using *functional* API of the Keras with TensorFlow as a backend. This makes models callable just like layers. This also gives an ability of weight sharing, shared layers or to have multiple inputs or outputs. Furthermore, it is also easily possible to connect one layer to any other layer.

5.1 ENet Model Architecture

ENet is an neural network which uses auto-encoder network to learn the encoding function ENet$(\cdot)$. The encoding function ENet(x_{obs}) three linear layers each followed by Parametric Rectified Linear Unit (PReLU) layer, input and output layer (named as `encoded` in the model summary). It is important to note that `shared_axes` parameter of PReLU layer was set to 1, which means that all learnable parameters will be shared along that axis. Hence, in this case learnable parameters reduce to 1.

The input of the encoder is the flattened raw point cloud obtained from depth camera or LIDAR. Its dimensionality depends upon number of points in point cloud and dimensionality of workspace. In case of Iteration 1, a 2D case, point cloud is subsampled to 1400 points hence, size of the input becomes 2800. In this case encoder function encodes obstacle space in 28 dimensional latent space.

Trainable parameters for `dense_1` can be calculated as $2800 \times 512 + 1 \times 2800$ i.e., $1,434,112$.

The decoder function which is inverse of ENet can be denoted as ENet$^{-1}(x_{obs})$. It has completely opposite structure of encoding part of the network.

Following is the Keras style model summary of auto-encoder network used to train ENet$(\cdot)$ function.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 2800)              0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               1434112
_________________________________________________________________
p_re_lu_1 (PReLU)            (None, 512)               1
_________________________________________________________________
dense_2 (Dense)              (None, 256)               131328
_________________________________________________________________
p_re_lu_2 (PReLU)            (None, 256)               1
_________________________________________________________________
dense_3 (Dense)              (None, 128)               32896
_________________________________________________________________
p_re_lu_3 (PReLU)            (None, 128)               1
_________________________________________________________________
encoded (Dense)              (None, 28)                612
_________________________________________________________________
dense_4 (Dense)              (None, 128)               3712
_________________________________________________________________
p_re_lu_4 (PReLU)            (None, 128)               1
_________________________________________________________________
dense_5 (Dense)              (None, 256)               33024
_________________________________________________________________
p_re_lu_5 (PReLU)            (None, 256)               1
_________________________________________________________________
dense_6 (Dense)              (None, 512)               131584
_________________________________________________________________
p_re_lu_6 (PReLU)            (None, 512)               1
_________________________________________________________________
decoded (Dense)              (None, 2800)              1436400
=================================================================
Total params: 3,206,674
Trainable params: 3,206,674
Non-trainable params: 0
_________________________________________________________________
```

5.1.1 Loss Function

As already discuss the contractive type of auto-encoder used for learning the ENet function as according to Qureshi et al. CAE learns robust (towards small changes around the training examples) and invariant feature space required for planning and generalization to unseen workspaces.

Hence, the custom loss function or reconstruction loss is defined as:

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

5.1.2 Hyper-parameters

- Number of epochs: 400
- Batch size: 1024
- Optimizer: Adam

5.2 PNet Model Architecture

PNet is the planning network which takes encoded obstacle space $\boldsymbol{Z}$, start and goal states as an input. As $\boldsymbol{Z} \in \mathbb{R}^{28}$, start and goal states for 2D mobile robot case are x and y co-ordinate i.e., x_{start} and $x_{goal} \in \mathbb{R}^2$ each. Hence input to the PNet belongs to $\mathbb{R}^{32}$ in this case.

Following is the Keras style model summary of auto-encoder network used to train planning network.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 32)                0
_________________________________________________________________
dense (Dense)                (None, 1280)              42240
_________________________________________________________________
p_re_lu (PReLU)              (None, 1280)              1
_________________________________________________________________
dropout (Dropout)            (None, 1280)              0
_________________________________________________________________
dense_1 (Dense)              (None, 1024)              1311744
_________________________________________________________________
p_re_lu_1 (PReLU)            (None, 1024)              1
_________________________________________________________________
dropout_1 (Dropout)          (None, 1024)              0
_________________________________________________________________
dense_2 (Dense)              (None, 896)               918400
_________________________________________________________________
p_re_lu_2 (PReLU)            (None, 896)               1
_________________________________________________________________
dropout_2 (Dropout)          (None, 896)               0
_________________________________________________________________
dense_3 (Dense)              (None, 768)               688896
_________________________________________________________________
p_re_lu_3 (PReLU)            (None, 768)               1
_________________________________________________________________
dropout_3 (Dropout)          (None, 768)               0
_________________________________________________________________
dense_4 (Dense)              (None, 512)               393728
_________________________________________________________________
p_re_lu_4 (PReLU)            (None, 512)               1
_________________________________________________________________
```

```
dropout_4 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_5 (Dense)              (None, 384)               196992
_________________________________________________________________
p_re_lu_5 (PReLU)            (None, 384)               1
_________________________________________________________________
dropout_5 (Dropout)          (None, 384)               0
_________________________________________________________________
dense_6 (Dense)              (None, 256)               98560
_________________________________________________________________
p_re_lu_6 (PReLU)            (None, 256)               1
_________________________________________________________________
dropout_6 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_7 (Dense)              (None, 256)               65792
_________________________________________________________________
p_re_lu_7 (PReLU)            (None, 256)               1
_________________________________________________________________
dropout_7 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_8 (Dense)              (None, 128)               32896
_________________________________________________________________
p_re_lu_8 (PReLU)            (None, 128)               1
_________________________________________________________________
dropout_8 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_9 (Dense)              (None, 64)                8256
_________________________________________________________________
p_re_lu_9 (PReLU)            (None, 64)                1
_________________________________________________________________
dropout_9 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_10 (Dense)             (None, 32)                2080
_________________________________________________________________
p_re_lu_10 (PReLU)           (None, 32)                1
_________________________________________________________________
dropout_10 (Dropout)         (None, 32)                0
_________________________________________________________________
dense_11 (Dense)             (None, 2)                 66
=================================================================
Total params: 3,759,661
Trainable params: 3,759,661
Non-trainable params: 0
_________________________________________________________________
```

5.2.1 Hyper-parameters

- Number of Epochs: 500
- Batch Size: 1024
- Dropout Rate: 0.5

- Optimizer: Adagrad
- Loss Function: Means squared Error (MSE)

5.3 Training and Testing

5.3.1 ENet

For training the encoder network 27000 (in total) samples, out of which 25000 were used as training set and 2000 were used as validation set. Complete dataset was shuffled before using it as an input. Each point cloud had 2800 data points.

For testing remaining 3000 obstacles were used which were completely unseen. Accuracy and contractive loss were used as metric and plotted using TensorBoard.

5.3.2 PNet

To train PNet 4000 paths from 100 environments each with their latent space encoding were used. All paths are of name length hence, smaller paths were zero-padded to make size of the array consistent.

For testing, remaining 10 unseen and few start and end points from seen environments were used to check if the feasible path is found or not. Both obstacles actual path and path planned by MPNet are plotted using `matplotlib` and are reported in Chapter 6.

As already mentioned, both networks were trained independently, hence training of both networks can be considered as supervised training.

5.4 Technologies Used

5.5 Training Setup

The system used for training and testing has following configuration:

- System Memory: 512 GB
- GPU Memory: 256 GB in total; 32 GB per GPU
- CPU: Intel® Xeon® CPU E5-2698 v4 @ 2.20GHz; 20-Dual Core
- Storage: 4×1.92 TB SSD

- 8×Tesla V100 GPU

Further GPUs are divided on quota basis and each department gets 2 GPUs. Which means only one GPU with 32 GB memory was used to train the model. RAM is always shared between all the active users. Training was not performed on the independent computer, but a shared cluster, hence performance results may vary.

Docker container `nvcr.io/nvidia/tensorflow:19.05-py3`, built by NVIDIA, was used for all the purposes. Two ports were exposed from the host, one for connecting the Python kernel and other for the TensorBoard.

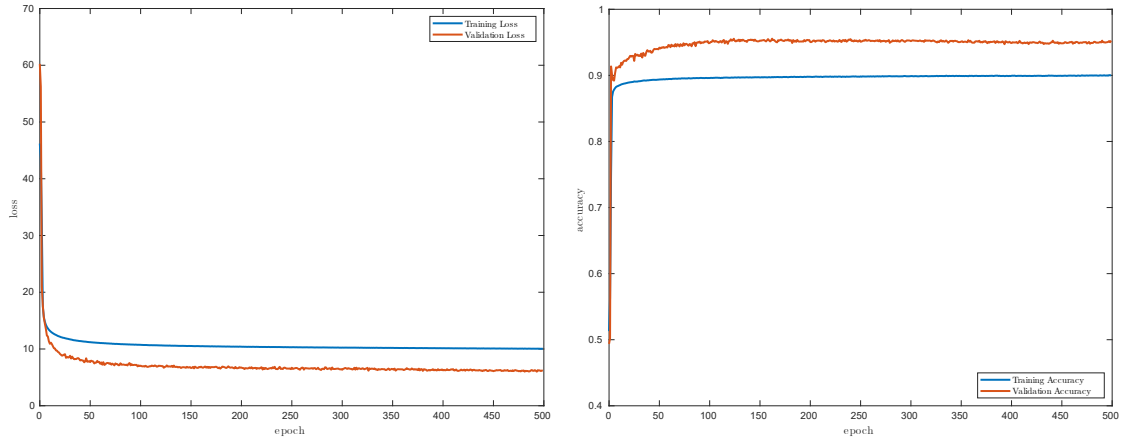
Chapter 6

Results

This sections briefly provides the results of the above exercise. It is important to note that, as the implementation was proof of concept various values of hyperparameters are not tried to find a good values hyperparameters.

Figure 6.1 shows loss and accuracy of PNet. Accuracy and loss of PNet is inherently dependent upon performance of the ENet. In fact, accuracy of ENet and PNet do not have much significance in this case as purpose of the network is not to imitate or plan same paths but to plan innovative and feasible paths by augmenting neural capabilities with an oracle planner. Hence, of network is able to plan the different but feasible path, it should be considered as a success.

But, from the accuracy plot we can get a general idea, if the network can behave as expected or not.

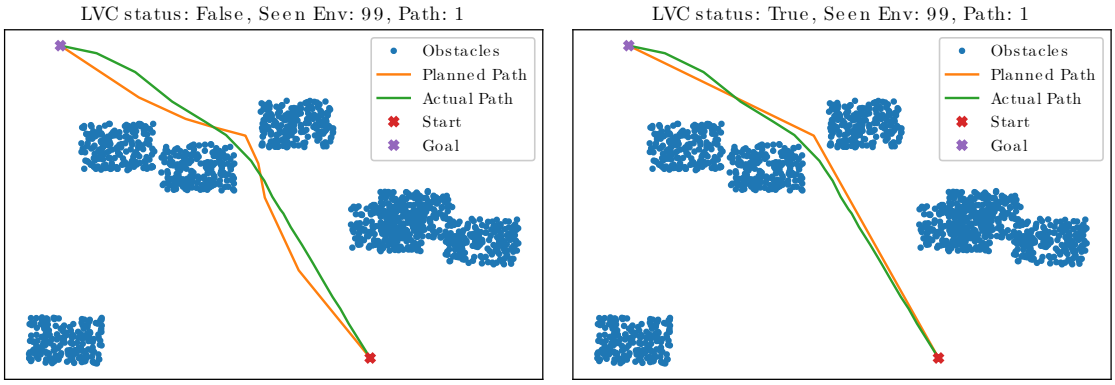


(a) (b)

Figure 6.1: Training and Validation Metrics for PNet: (a) MSE loss (b) accuracy

Figure 6.2 show results on seen test set with and without lazy state contraction or lazy vertex contraction. It is clear from figures 6.2(a) and 6.2(b) that path planned with LVC is smoother which results in lower configuration changes which directly affects the reaching time. Furthermore, we can also see the planned path is not exactly same as actual path, which may give early indications of network is not overfitted on seen environments.

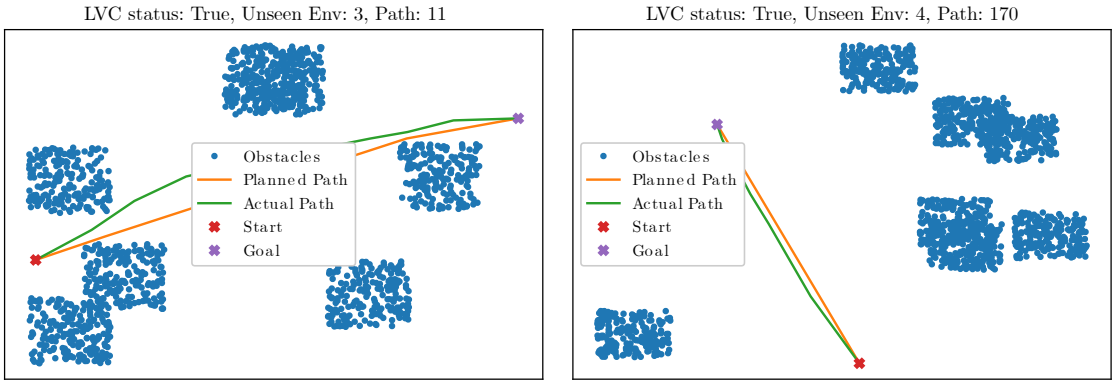
Figure 6.3 shows the similar results on previously unseen environments by both ENet and PNet.



(a) (b)

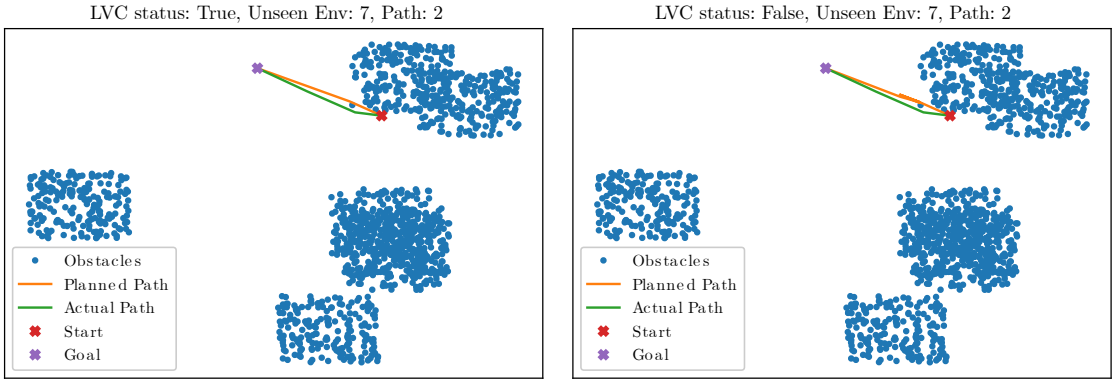
Figure 6.2: Actual and planned paths for environment 99 with unseen start and goal positions: (a) with lazy vertex contraction (b) without lazy vertex contraction

It is also interesting to see that MPNet with LVC tries to plan more smoother paths than actual paths respecting robot kinematic model. It is also interesting to note the usage of replanning stages in fig. 6.3(a) which allows it to plan relatively complex paths.



(a) (b)

Figure 6.3: Actual and planned paths for unseen environments with LVC: (a) environment 3 (b) environment 4



(a) (b)

Figure 6.4: Failed example of path planning: (a) with lazy vertex contraction (b) without lazy vertex contraction; (Oracle Planner is not employed)

Despite of replanning stage, MPNet fails to plan paths in certain environments. Some times turning LVC off helps to plan paths successfully this is due to turning LVC off allows to plan more cornered paths. Fig. 6.4 shows one such example. In fig. 6.4(b) when LVC was off, results are quite different from fig. 6.4(a) where LVC was turned off. In fig. 6.4(b) where LVC was off, path length is 2117 (due to replanning stages with lot of relaxation in absence of LVC) where as in case of fig. 6.4(a) path length is much small. This failure is attributed to absence of oracle planner.

Chapter 7

Discussions

This chapter presents a discussion on different aspects of MPNet and also compares results of the experiment with literature.

Results presented in Chapter 6 are in accordance with the literature reviewed. Although thorough statistical tests were not done due other interests of company at that time.

If MPNet fails to find end-to-end path replanning stage is enacted which iteratively plans path segments using neural planning and/or underlying oracle planner. As in this case MPNet incorporates RRT* as an oracle planner which is a probabilistically complete algorithm. Hence, we can guarantee the probabilistic completeness of the algorithm.

Training vs. Validation Accuracy

It is also interesting to note that in fig. 6.1 validation accuracy is higher than that of the training accuracy. This happens when Dropout is used, since the behaviour when training and testing are different. While training, a percentage of the features or neurons are set to zero (50% in this case since `Dropout(0.5)`). While testing, all features are used (and are scaled appropriately). So the model at test time is more robust and can lead to higher testing accuracies.

Lazy State Contraction

Lazy State Contraction or Lazy Vertex Contraction is already discussed in section 4.4 and it is introduced as a smoothing technique. Further, one can provide certain relaxations and regulations on various configuration parameters to regulate or to provide planning process required flexibility.

Using as DeepSampler

MPNet can also be used to generate the samples from workspace using machine learning approach. As mentioned PNet model has Dropout at almost every layer and this stochasticity can be exploited to generate multiple informed samples. Further, these samples can be used for any sampling based planner which can drastically reduce the planning time for classical SMP.

Chapter 8

Conclusion and Recommendations

During the course of internship a deep learning based motion planner called MPNet based on work of Qureshi and Yip which is able to plan collision-free near-optimal paths rapidly and efficiently is presented. This method is also able to generate samples for sampling-based motion planners in a subspace of a given configuration space that most likely contains solutions including the optimal path.

Further, strategies such as continual learning and active continual are to be implemented. Implementing these strategies will make sure robot learns continually i.e., innovative paths planned by oracle planner component of the MPNet can be utilized for training the network again.

Another task is to implement fundamental modules such as collision checker, cost function to optimize for planning the requisite motion. Further, reinforcement learning can be used to learn such a cost function with different constraints. Another concern to be addressed is planning in dynamically moving environments.

Chapter 9

Side Tasks

9.1 Virtual Measurement

The aim of the virtual measurement is to find the unknown transform ${}^{W}\mathbf{T}_{S}$ using known transforms. Where, ${}^{W}\mathbf{T}_{R}$,${}^{W}\mathbf{T}_{R}$,${}^{R}\mathbf{T}_{F}$ and ${}^{F}\mathbf{T}_{S}$ are the known transforms defined as,

${}^{W}\mathbf{T}_{S}$: Homogeneous transformation matrix from *camera* coordinate system S to *workpiece* coordinate system W

${}^{W}\mathbf{T}_{R}$: Homogeneous transformation matrix from *robot base* coordinate system R to *workpiece coordinate* system W

${}^{R}\mathbf{T}_{F}$: Homogeneous transformation matrix from *flange* coordinate system F to *robot base* coordinate system R

${}^{F}\mathbf{T}_{S}$: Homogeneous transformation matrix from *camera* coordinate system S to *flange* coordinate system F

Transform ${}^{R}\mathbf{T}_{F}$ is found by forward robot kinematics. This is easily available from robot controller. Transform ${}^{W}\mathbf{T}_{R}$ and ${}^{F}\mathbf{T}_{S}$ are estimated using *hand-eye calibration*. Knowing these transforms, finding unknown transform ${}^{W}\mathbf{T}_{S}$ is pretty trivial task. Fig. 9.1 depicts the scenario diagrammatically.

Considering right-handed coordinate system[1]

$$ {}^{W}\mathbf{T}_{S} = {}^{W}\mathbf{T}_{R}^{R}\mathbf{T}_{S} $$

but

$$ {}^{R}\mathbf{T}_{S} = {}^{R}\mathbf{T}_{F}^{F}\mathbf{T}_{S} $$

hence

$$ {}^{W}\mathbf{T}_{S} = {}^{W}\mathbf{T}_{R}^{R}\mathbf{T}_{F}^{F}\mathbf{T}_{S} $$

[1] although motion planning framework at Fraunhofer IPA uses left-handed coordinate system, all mathematical calculations are shown in right-handed coordinate system for intuitiveness

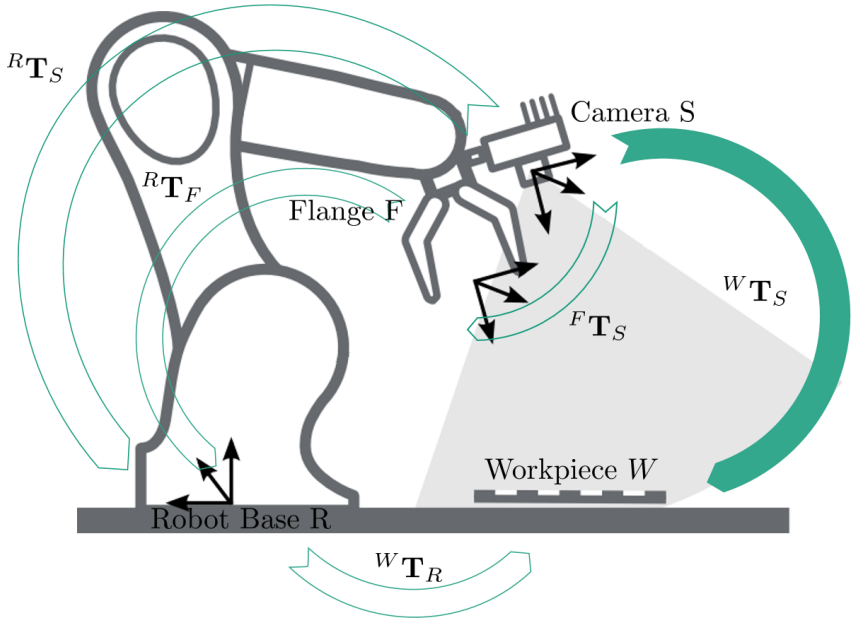


Figure 9.1: Virtual Measurement Setup

Hand-Eye Calibration

The hand-eye calibration problem first appeared and got its name from the robotics community, where a camera, *the eye* was mounted on the gripper, *the hand* of a robot. The cameras was calibrated using a calibration pattern. Then the unknown transformation from the robot coordinate system to the calibration pattern coordinate system as well as the transformation from the camera to the hand coordinate system need to be estimated simultaneously [15].

Transforms using hand-eye calibrations were already available using previous implementations at IPA. Implementation of virtual measurements was done in C++ with Point Cloud Library (PCL[2]) to augment the current functionality of the program.

Further, same was implemented in Python using Open3D[3] library. This implementation is meant to be used by consuming the web-services. This is described in depth in section 9.2.

Although, both PCL and Open3D have Python bindings, Open3D was selected for following reasons:

- PCL is now largely dormant, whereas, Open3D is actively updating and is backed by labs at Intel.
- Less number of carefully chosen dependencies [16], hence can be set of on various platforms easily.
- According to Zhou et al., the implementations using Open3D Python interface are approximately 5 times shorter than implementations based on PCL. This cleans a lot of code boilerplate.
- Open3D also provides direct memory access to various point cloud data fields via numpy arrays making is quick and easy to extend it for machine learning tasks.

[2] http://pointclouds.org/
[3] http://www.open3d.org/

- According to Zhou et al., Open3D implementations are generally faster than their counterparts in other 3D processing libraries.

Figures 9.2(a)–9.2(d) show sequence of operations on the workpiece to simulate point-cloud using sensor according to its field of view.

9.2 Web-services for simulation environment

Web Service are the *avatar*[4] that assist in going beyond boundaries and using the internet as a collaborative medium. They are essentially an instance of Service Oriented Architecture (SOA). Further using web services as a platform for SOA can address certain challenges in enterprise computing such as availability, heterogeneity, time to market & cost of development, operation and maintenance in a better manner.

Keeping this in mind it was planned to convert current application integration like spaghetti to distributed layers of application like lasagna. For this two technologies are widely used: Simple Object Access Protocol (SOAP) and Representational state transfer (REST).

SOAP vs. REST

SOAP	REST
Language, platform, and transport independent; standard implementation using HTTP and SMTP	REST requires use of HTTP
Works well in distributed enterprise environments, complex systems and tight integration	REST assumes direct point-to-point communication
Standardized XML-schema conforming message format	XML or JSON can be used for messages
Built-in error handling	No built-in error handling
Larger message formats due to XML	Smaller message formats
Larger learning curve	Smaller learning curve

Further, detailed comparison is available on https://www.soapui.org/learn/api/soap-vs-rest-api.html.

Keeping above differences in mind, specially, learning curve, integration and interoperability with multiple legacy systems RESTful web services philosophy was chosen.

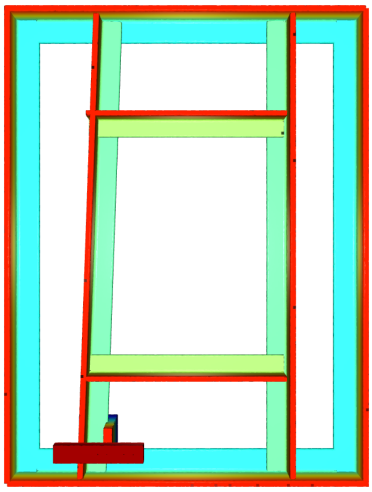
For implementation Flask micro-framework with Flask RESTPlus plugin was used. The reason behind RestPlus was easy marshalling of Python objects to JSON and OpenAPI compliance. Following is one of the screenshots (fig. 9.3) of system develop to consume these services. The results obtained using consuming the services are exactly same as previous section.

Proof of concept webpage (fig. 9.4) is developed with MySQL connectivity to so that exhaustive library of CAD models with different sensors, robots can be generated. Such library can be used for applications such as machine learning or other internal applications. Further, this concept can also be extended to provide service from cloud to existing customers.

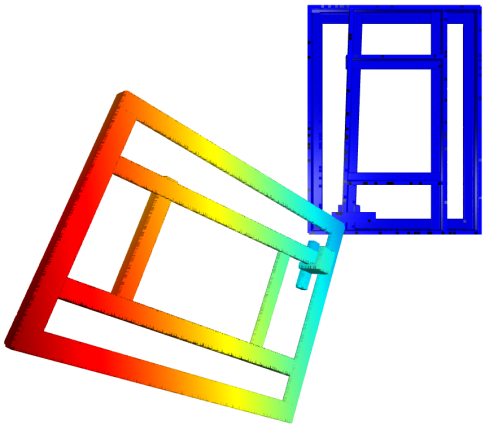
9.3 Data collection plug-in for Laser Tracker

Aim of this task is to create a plug-in for Leica Absolute Tracker AT960 to save the reading of the laser tracker for machine learning purposes. The requirement of plug-in is specifically that,

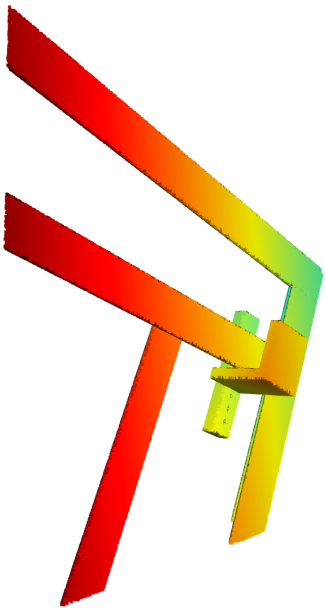
[4] incarnation, /ǽvə,tɑɽ/



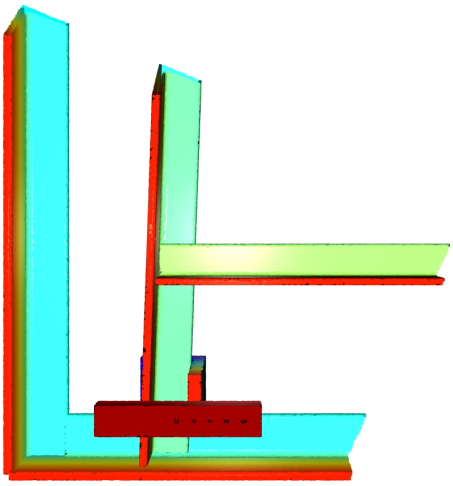
(a)



(b)



(c)



(d)

Figure 9.2: Images of point clouds depicting: (a) original workpiece (obtained from CAD model); (b) virtually transformed workpiece using ${}^{W}\mathbf{T}_{S}$; (c) cropped workpiece as per sensor view-field; and (d) transformed back into workpiece coordinate frame W using ${}^{S}\mathbf{T}_{W}$

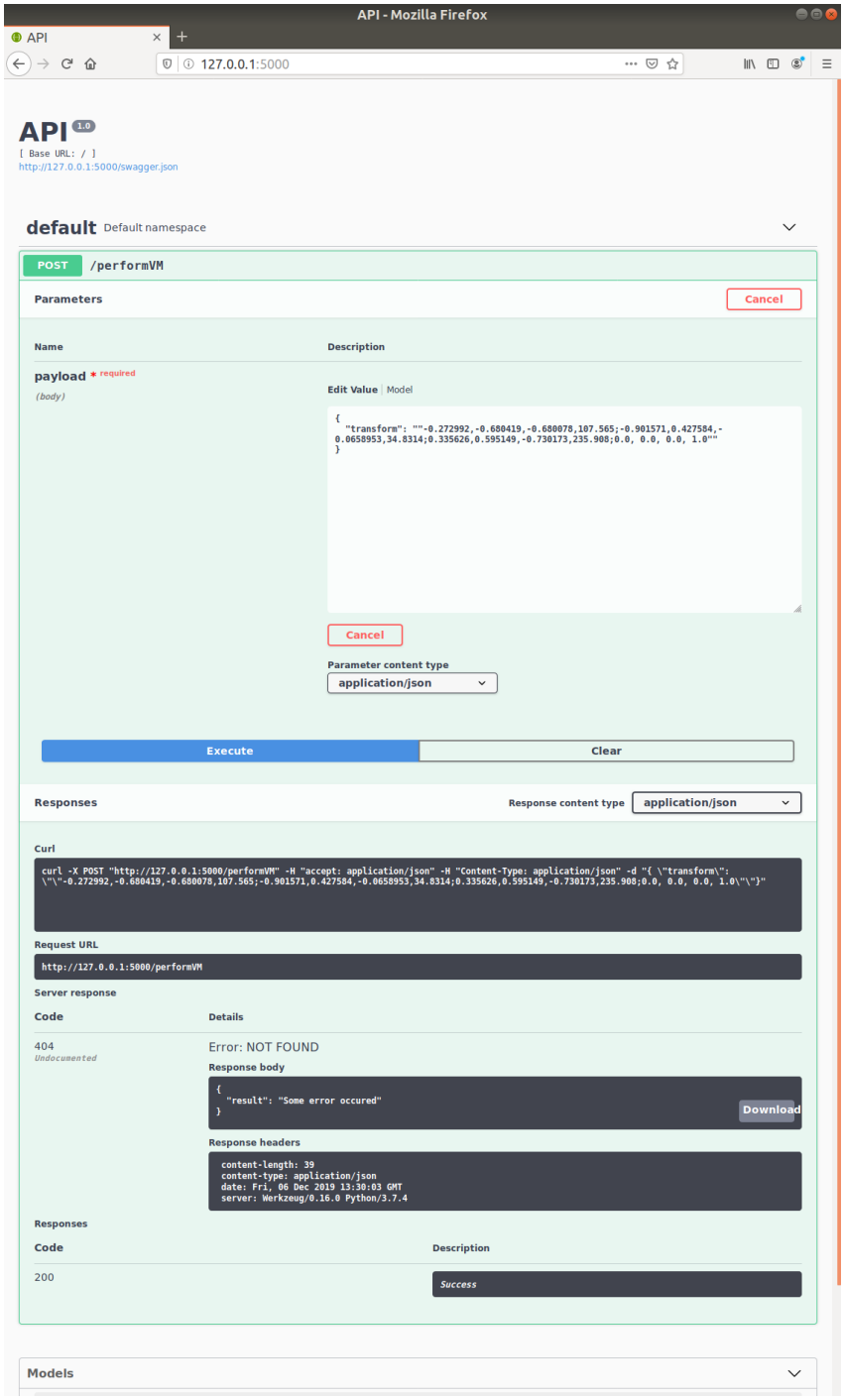


Figure 9.3: Swagger Documentation and Swagger UI for API consumption

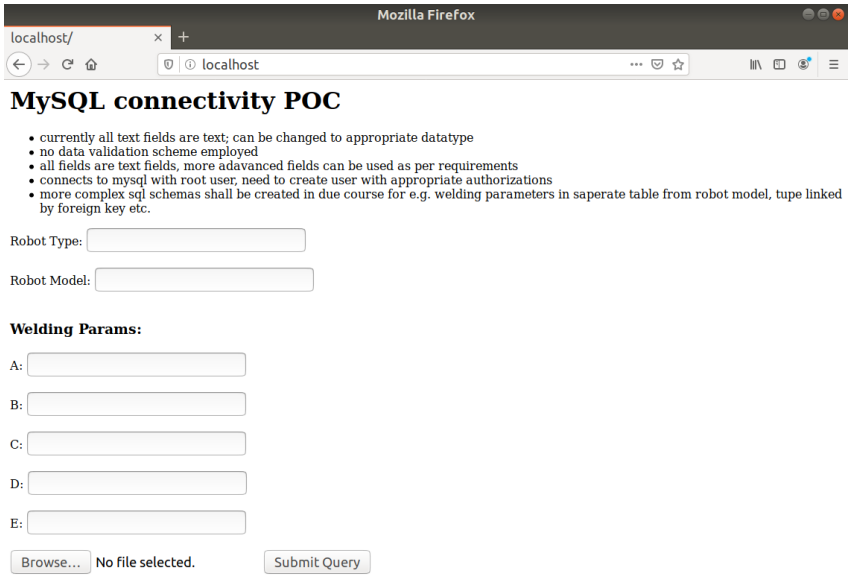


Figure 9.4: Proof og concept webpage

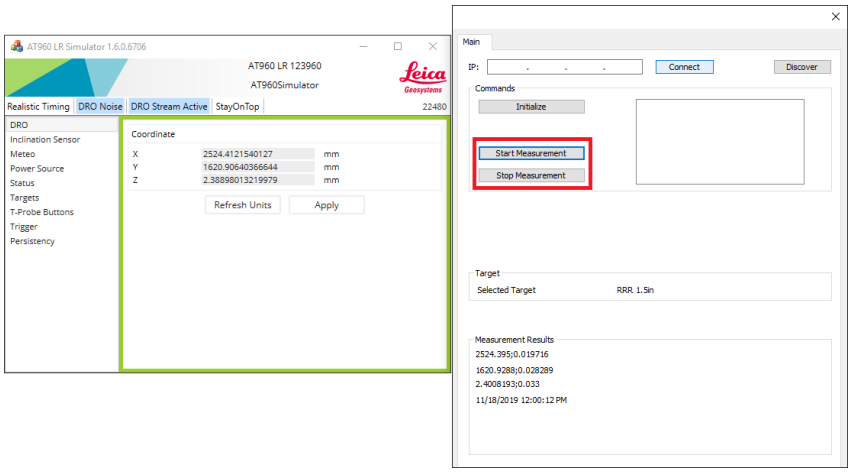


Figure 9.5: GUI of laser tracker plug-in built on the top of Leica SDK

it should be able to save 6D measurements when robot sends an signal indicating it has reached a desired position.

The plug-in was developed in C++ with MFC using SDK provided by the manufacturer. Using this SDK it was possible to save the reading in a comma separated values format (CSV).

While implementing this, new worker thread was created alongside GUI thread which continuously listens to communication channel between robot and the software. The simulator AT960LR is provided by the manufacturer.

Acknowledgements

I would like to extend my sincere gratitude to Mr. Christian Landgraf, Mr. Akhil Sadanandan Anand and Dr. Gauthier Hentz for giving me a chance to do my internship at such a renowned institution. Furthermore, I would also like to thank Mr. Christian Landgraf explicitly for supervising my internship and providing valuable feedback from time-to-time. I express my deepest thanks to dr.ir. Ferdi van der Heijden at the University of Twente for taking part in supervising my internship at the university level. I would also like to thank Mrs. Sanne S. Gritter-Spuls for her help before and during mobility.

I am also thankful to Fraunhofer-Institute für Produktionstechnik und Automatisierung for trusting in me and providing me with this wonderful opportunity. Since all work and no play make Jack a dull boy, I would also like to thank my friends who helped me settle down and ease into the workplace to the best of their abilities. I would also like to mention Aditi Ailavajhala who was there for me and kept me sane during my time in Stuttgart.

Last but not the least, I would like to thank my family back in India, for providing me with constant support; morally, emotionally, financially and being there for me always.

Bibliography

[1] A. H. Qureshi, A. Simeonov, M. J. Bency, and M. C. Yip, "Motion planning networks," *arXiv preprint arXiv:1806.05767*, 2018.

[2] A. H. Qureshi, Y. Miao, A. Simeonov, and M. C. Yip, "Motion planning networks: Bridging the gap between learning-based and classical motion planners," *arXiv preprint arXiv:1907.06013*, 2019.

[3] A. H. Qureshi and M. C. Yip, "Deeply informed neural sampling for robot motion planning," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2018, pp. 6582–6588.

[4] S. M. LaValle, *Planning Algorithms*. Cambridge University Press, 2006. [Online]. Available: http://planning.cs.uiuc.edu/

[5] R. U. Kavraki Lab, "Open motion planning library: A primer." [Online]. Available: ompl.kavrakilab.org/OMPL_Primer.pdf

[6] J. J. Kuffner and S. M. LaValle, "Rrt-connect: An efficient approach to single-query path planning," in *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, vol. 2, April 2000, pp. 995–1001 vol.2.

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

[8] S. M. LaValle, "Rapidly-exploring random trees: A new tool for path planning," Iowa State University, Ames, Iowa, USA, Tech. Rep., 1998.

[9] D. Lopez-Paz and M. A. Ranzato, "Gradient episodic memory for continual learning," in *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds. Curran Associates, Inc., 2017, pp. 6467–6476. [Online]. Available: http://papers.nips.cc/paper/7225-gradient-episodic-memory-for-continual-learning.pdf

[10] A. Karpathy. Cs231n: Convolutional neural networks for visual recognition. [Last Accessed: December 04, 2019]. [Online]. Available: http://cs231n.github.io/neural-networks-1/

[11] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014. [Online]. Available: http://jmlr.org/papers/v15/srivastava14a.html

[12] S. Rifai, P. Vincent, X. Muller, X. Glorot, and Y. Bengio, "Contractive auto-encoders: Explicit invariance during feature extraction," in *Proceedings of the 28th International Conference on International Conference on Machine Learning*. Omnipress, 2011, pp. 833–840.

[13] J. R. Diaz P., T. Dietz, P. Ockert, A. Kuss, M. Hägele, and A. Verl, "Automatic optimal motion generation for robotic manufacturing processes: Optimal collision avoidance in robotic welding," in *2016 IEEE International Conference on Automation Science and Engineering (CASE)*, Aug 2016, pp. 154–161.

[14] G. I. Parisi, R. Kemker, J. L. Part, C. Kanan, and S. Wermter, "Continual lifelong learning with neural networks: A review," *Neural Networks*, vol. 113, pp. 54 – 71, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0893608019300231

[15] Hand-eye calibration. Lehrstuhl für Informatikanwendungen in der Medizin & Augmented Reality, Technische Universität München. [Last Accessed: August 15, 2019]. [Online]. Available: http://campar.in.tum.de/Chair/HandEyeCalibration

[16] Q.-Y. Zhou, J. Park, and V. Koltun, "Open3D: A modern library for 3D data processing," *arXiv:1801.09847*, 2018.