Learning Based Motion Planning For Mobile Robot Navigation Using Motion Primitives

Presented By
Ainil Norazman and Ali Ekin Gurgen

Submitted To
Department of Mechanical and Aerospace Engineering
Princeton University

Undergraduate Junior Independent Work Final Report

May 15, 2020

Anirudha Majumdar
Michael Littman
MAE 340D
22 Pages

Abstract

Vision-based motion planning has been used in navigation for unmanned vehicles dominantly for ground and aerial robots. To improve planning performance in domains with restricted visibility and reduce training on actual hardware, this paper uses computer simulations and studies the fundamental question: “how do we learn a neural network based motion planner using data from an onboard camera to generate a sequence of motion primitives that avoid collision?”. In this paper, the training approach uses a policy based on PAC-Bayes framework [1] on a mobile robot navigation simulation to generate a rich dataset of depth map images. The training procedure consists of two main sections: (1) training a neural network pipeline that predicts the depth map image at the next time step when it is given the current image and the applied motion primitive, (2) training a neural network that predicts the safety label (probability of collision) for the next time step when it is given the current depth map image. Finally, a neural network based planning algorithm that can choose the motion primitive that results in the lowest probability of collision in longer time horizons is proposed. This paper discusses the successes and failures of the training procedures as well as proposing alternative approaches that can improve performance in future work.

Acknowledgements

We would like to thank Professor Anirudha Majumdar for his support and mentorship throughout the project as he provided us with the necessary resources and background knowledge for conducting our research, data collection and training. We express our deepest appreciation towards Sushant Veer who continuously assisted us with writing code, debugging and provided lessons that further enriched our understanding on computer simulations and neural networks. Special thanks to Jo Ann Kropilak-Love and Professor Luigi Martinelli for attending to our questions and easing arrangements related to logistical issues. Despite having to change our research direction towards a simulation-based approach and embrace a new research timeline following circumstances given by the global pandemic, we are thankful for the continuous support and help by the department for making this project possible within the new 8-week timeframe.

This final paper represents our own work in accordance with University regulations.

Table of Contents

1. Introduction

Navigation of mobile robots through dense obstacle environments is one of the most fundamental problems in robotics. A classical approach to tackling the navigation problem involves solving three main subtasks: localization, mapping and motion planning. With increasing computational power, vision-based motion planning has shifted towards a deep-learning-based approach using neural networks to produce “end-to-end” solutions to the navigation problem. The goal of this paper is to leverage on the end-to-end learning based approach where we design and train an artificial neural network pipeline to predict a sequence of motion primitives that result in collision-free paths in the environment by using the given camera vision inputs.

For many years, classical learning approaches to navigation problems have followed the modularity of conducting localization, mapping and motion planning in sequence. Localization is the problem of estimating the robot’s pose relative to a given map of the environment. Probabilistic localization algorithms that are variants of the basic Bayes filter can be used to perform mobile robot localization [2]. Mapping, on the other hand, is the problem of constructing a representation of the surrounding environment relative to the position of the robot. With the assumption that robot’s pose is known, algorithms such as Occupancy Grid Mapping can be used to generate maps from sensor measurements [3]. When these two subtasks are performed concurrently, we arrive at the Simultaneous Localization and Mapping, commonly abbreviated as SLAM, which has become the standard solution to the mapping and localization problems in many robotic applications [4]. Motion planning is the task of determining a viable path from point A to point B in the configuration space. Choset et al. [5] defined configuration space as the space of all configurations that a robot can achieve. Obstacle avoidance is the core problem of motion planning and various discrete (e.g., Bellman-Ford, A*) and continuous (e.g., rapidly-exploring random trees, probabilistic roadmap method) path planning algorithms are commonly used to perform this task.

In the last decade, the use of computer vision for performing the navigation of mobile robots has become a very powerful sensor modality. Vision can extract information beyond geometric representations of the environment and does not rely on having to emit signals. It provides an energy-efficient and light-weight sensor solution for the mobile robot navigation problem. Classical approaches that incorporate computer vision into robot navigation involve using RGB-D images to perform state estimation and mapping, which are later used for classical motion planning algorithms. However, classical computer vision approaches to path planning rely on high degree of environmental information, which would not be efficient in novel environments or densely populated environments. The main disadvantage of such a method is its computational intensiveness and inability to tolerate uncertainties that result from dynamic environments for performance of tasks in longer time horizons.

An end-to-end learning based approach forgoes the modularity that is inherent in classical approaches and provides a long-horizon solution to learning a policy that directly achieves a desired goal state. In comparison to the classical approaches, a deep learning based end-to-end approach is able to provide minimality in the solution to the navigation problem and it is significantly less expensive once it has been trained. Most importantly, a deep learning based motion planner can use experiential knowledge to predict future outcomes and give better performance results in longer time horizons.

In our end-to-end learning approach, the neural network learns to map depth images to latent space representations and performs planning in the latent space [6]. In particular, a dataset consisting of a sequence of depth map images up to point of collision is drawn (independently and identically) from a distribution given by a policy based on the PAC-Bayes framework that uses a set of random weights. Given a rich dataset that is split into training and validation sets, we train a neural network to encode the collision-free depth map images to their latent space representations. These representations are then concatenated with the applied primitive motion actions and passed through another neural network that reconstructs the predicted depth map image after applying the motion primitive. Then, by training a separate neural network that can predict the probability of collision given a latent space representation, we aim to construct a neural network based motion planner that predicts a sequence of motion primitives that result in collision-free paths in long-term horizons.

2. Background

2.1 Motion Primitive Libraries

Motion primitive libraries consist of pre-computed primitive trajectories that can be sequentially composed to generate a rich class of motions [7]. The use of motion primitive libraries have been prevalent in robotics and digital animation to produce natural-looking motions. Each primitive represents a single step in the robot trajectory. In this paper, we use the *Record and Playback* strategy [8] where the set of primitives that are included in the library are restricted to only maneuvers that can guarantee smooth trajectories and collision-free. Additionally, motion primitives considered in this paper bring the robot back to its initial heading direction after the primitive application is finalized. During motion planning, the planner generates a sequence of motion primitives that result in an overall smooth trajectory that is also feasible for the robot's dynamics. The approach that is used in this paper to decide on the sequence of primitives to be picked from the library is determined heuristically by the learned neural network.

2.1.1 Motion Primitives of the Husky Robot

A husky robot is a car-like vehicle with non-holonomic constraints (kinematic constraints) which impose restrictions on the robot's achievable configuration. In this paper, the husky robot is simulated in a constrained space where certain configurations, such as one that involves a

backing up maneuver would be unachievable. Similarly, the speed of the robot is constant and cannot be controlled by the motion planner. Since we can only change the heading direction for the Husky robot, the motion primitives that characterize smooth trajectories that are collision-free take the form of forward line segments, circle arcs of fixed radius, or contact arcs.

2.2 Artificial Neural Networks

The use of artificial neural networks in motion planning, pattern recognition, and optimization problems has been prevalent due to its ability to provide optimal solutions. A neural network architecture consists of an input layer, hidden layer and output layer where there exists a connection across the different layers but not one within [9]. Regular neural networks receive an input and transform it through a series of hidden layers. A neural network based approach is categorized by the configuration of the hidden layers (i.e Single Layer, Multi-Layer, Competitive layer), the methodology used to determine neurons connection weights and its activation functions (i.e Sigmoid, ReLU, Maxout). In this paper, we focus on designing the architecture of Convolutional Neural Network (CNNs) to process depth map images.

2.2.1 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks take advantage of the fact that input is an image and preserve the spatial structure of images [10]. The layers of the CNNs have neurons arranged in 3 dimensions: width, height, depth. Each layer in the CNN transforms an input 3D volume into an output 3D volume. The main layer to the CNNs is called the *convolutional layer* where each neuron in the convolutional layer is connected only to a local region in the input volume spatially, but to the full depth represented by color channels or depth value [11]. The convolutional layer convolves filters with image by sliding over all spatial locations and computing dot products of the weights, which is then followed by activation functions. CNNs generally apply a sequence of these convolutional layers with activation functions that perform a certain fixed mathematical operation.

The three hyperparameters that control the size of the output volume in a convolutional layer are filter size (kernel), stride and padding. The configuration of a convolutional layer is defined as Conv2d[input, output, kernel, stride and padding]. The kernel corresponds to the size of the filter that is used for convolution. Stride specifies the size of stride for sliding the filters along the image. Zero-padding refers to padding around the border of the input volume with zeros. [12]

Ultimately, CNNs would provide us with less number of parameters for image processing. The detail of the neural network architecture will be explained in Section 3.0. The convolution process in CNNs allows us to exploit spatial invariances in images because visual features in one portion of an image is the same in another portion.

2.3 Loss Functions

In the deep learning framework, loss functions measure the difference between the output of the network and the correct output according to the training set [13]. In the context of optimization, we seek to minimize the losses computed from the difference between the output image of our neural network and the correct output image according to the training set containing RGB-D input in order to determine that the network is accomplishing the task that it is intended to do. Some predefined loss functions available for a neural network model are Sigmoid Cross Entropy, Gaussian Max Likelihood and Maximum Mean Discrepancy.

2.3.1 Binary Cross Entropy

We compute the Binary Cross Entropy loss to compare between the “true” distribution over the discrete classes p(t) and the model distribution over the predicted classes q(x). Then, we backpropagate by summing up the gradient of the losses. The formulation [14] of Binary Cross Entropy Loss can be expressed by:

$$\mathcal{L}(x, t) = -\sum_{k=1}^{K} p(t = k) \log q(x = k)$$

Equation 1: Computation of loss function using binary cross entropy (BCE) where K = 2

3. Design Process

The design section will define the configurations related to the data collection, training pipeline, neural network architectures, and structure of the neural network based motion planner. Alongside configuration details, the section provides the list of software libraries that allow us to run simulations, run datasets and perform neural network training. The entire code for this project used the Python language and its relevant libraries such as PyBullet (for simulations) and PyTorch (for NN training). All of the files used in data collection and training are available in the “husky-motion-prim-nav” repository under Princeton IRoM Lab’s GitHub page (https://github.com/irom-lab/husky-motion-prim-nav). The diagram below illustrates the essential components of the project and their communication.

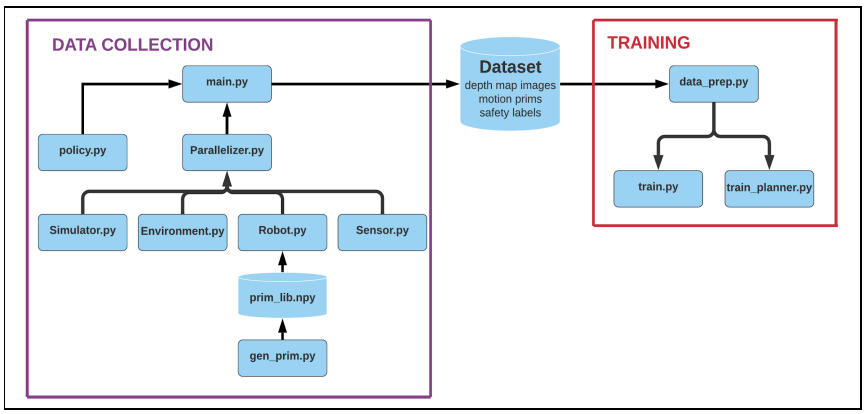


Figure 1: Illustration of the main components for Data Collection and Training procedures

3.1 Design of Simulation

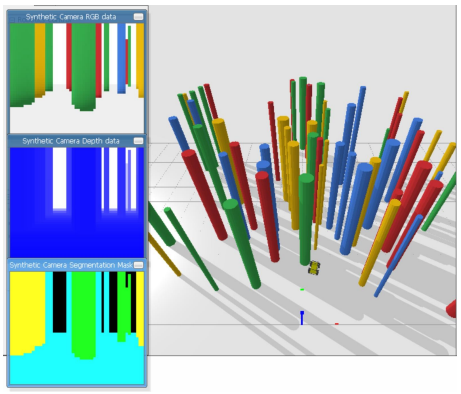


Figure 2: Visualization of the environment representing an obstacle course with a number of obstacles sampled by a random distribution where we train a husky robot to execute a trajectory which results in free collision.

3.1.1 Environment

The environment represents an obstacle course containing a number of obstacles that are sampled by a random distribution. For the obstacle course, we design an environment with area 10×10, and it is cluttered with 20 cylindrical obstacles. The height of the obstacle is defined as $100 \times$robot_height. The position (x,y) of each obstacle and the obstacle radius (r) are determined by random sampling using the Python function `random.random_sample()`. The positions and radii of the obstacles are within range $[(x_{min}, y_{min}), (x_{max}, y_{max})]$ and $[r_{min}, r_{max}]$.

For visualizing the output from the RGB-D camera mounted on the robot, we present the simulation of the environments in RGB map image and depth map image as displayed in Figure 2 above. The depth map images are stored as npy files for training purposes, but they are illustrated using a perceptually uniform sequential colormap for the purposes of our analyses.

3.1.2 Robot

We use the Husky URDF in Pybullet to simulate the dynamics of the robot with robot_height = 0.075. We train the husky robot to navigate through the obstacle course with smooth trajectories. The dynamics of the robot is governed by a library of motion primitives containing 25 primitives that can guarantee smooth trajectories that are also feasible for the Husky robot's dynamics. The execution of one motion primitive corresponds to one time step for the robot.

3.2 Design of Data Collection

The data collection procedure involves running the simulation described in Section 3.1 for a set number of videos, where each video consists of a maximum of 10 motion primitives. However, a

video can prematurely terminate due to the robot colliding with an obstacle. In fact, various data collection experiments have shown that when the simulation configurations explained in Section 3.1 along with a policy with random weights (used to randomly determine which primitive to apply at each step) are used, on average, 4.4 motion primitives per video are generated. Therefore for 10000 videos, we can get approximately 44000 time steps (a time step will refer to a motion primitive step for the entire report). For each motion primitive, we also save the associated depth map image (image taken right before we apply the primitive) as well as a safety label that indicates if the robot collides at the next time step when we apply it (0: no collision, 1: collision). Therefore, for 10000 videos, we get approximately 44000 distinct depth map images as well as the motion primitive associated with each image, and safety labels that indicate if a collision occurs after the motion primitives are applied to their associated images.

All of the depth map images are saved in a single dataset pool using a single-indexed convention and all of the primitive motions and safety labels are all stored in their respective numpy arrays. In other words, data from all 10000 videos are appended back to back to create a long sequence of depth map images. To resolve the conflict where the end of a video is followed by the beginning of another video, we use a convention of appending “-1” to the corresponding position in the array of motion primitives indicating collision.These locations where we have “-1” in the primitives array also corresponds to the locations of 1’s in the array of safety labels.

Preparing the collected data for training is one of the most important parts of the process. As it will be explained in the following section, the first part of our training procedure is to train an Encoder + Decoder pipeline that can predict the depth map image at the next time step when it is given the current depth map image and the applied motion primitive. For this training pipeline, it would not be reasonable to include the last image of each video since the reference image that would be used for calculating the loss does not exist due to the collision. Therefore, a special data loading script is used to filter out the depth map images where the safety label is 1 (indicating collision). Then, we randomly sample 80% of the left collision-free data to be used in the training set and use the remaining 20% in the validation set. The second part of the training procedure involves training a Reward Network that predicts the safety label (possibility of collision at the next time step) for a given depth map image. For this training, all of the unsafe (colliding) images were used while only 50% of the safe (not colliding) images were included. By doing so, we managed to get a 1.7:1 safe-to-unsafe ratio which is quite reasonable to consider our dataset as unbiased. Similarly to the first training, 80% of the dataset is used in training and 20% is used in validation.

3.3 Design of Training Pipeline

The two training pipelines proposed for conducting training on the dataset are illustrated below in Figure 3 and Figure 4. The first training pipeline seeks to predict a depth map image at time step (t+1) given an input depth image at time t. The second training pipeline seeks to predict a

safety label associated with the input image at time t, which would allow the network model to predict the probability of collision. The training for both pipelines are conducted separately and begins by the order of training encoder and decoder first, followed by training for safety label prediction using the updated weights from first training.

3.3.1 Training Pipeline for Prediction of Depth Map Image at Time (t+1)

The Encoder maps the input depth map image to a new latent space representation, which is then concatenated with the known applied motion primitive. The newly mapped latent space representation concatenated with the motion primitive is passed through the Decoder which is configured by a series of fully connected layers that perform linear transformations and outputs a predicted depth image of the next time step. We exploit the learning capability of the neural network by passing the losses between target image and predicted image computed using Binary Cross Entropy into a gradient-based optimizer, Adam optimizer, to update the recently learned weight distribution. In this approach, we learn the neural network with a learning rate of 1×10^{-5} .

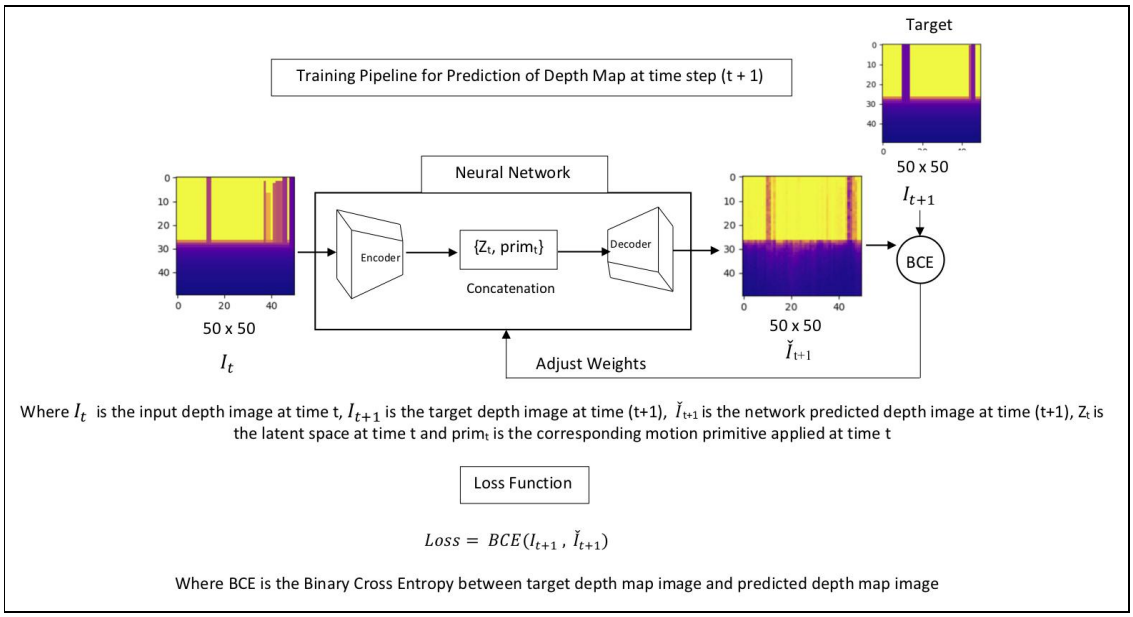


Figure 3: Illustration of training pipeline for prediction of depth map image at time (t+1). In this pipeline, we learn the Neural Network model by passing input of a depth map image at time t visualized as a 50x50 image to output a reconstructed image for depth map image at time (t+1)

3.3.2 Training Pipeline for Prediction of Safety Label for Image at Time (t)

The approach of this pipeline is interested in knowing the probability of collision at the next step given a depth map image at time t. The calculated probability of collision and probability of no collision are attached to the safety label associated with every depth map image. In this approach, the Reward Net learns to predict the correct safety label using updated weights from

the Encoder + Decoder training in the first training pipeline. The predicted safety label $\widehat{L}$ contains probability values $\widehat{p}$ and $(1-\widehat{p})$ that are between 0 to 1, where $\widehat{p}$ is the predicted probability of collision. The loss between the predicted safety label and the target label are calculated using Binary Cross Entropy. We do backpropagation to update the weights by passing the loss into a gradient-based optimizer, which would allow Reward Network to learn outputting predicted safety labels that are accurate with the target labels.

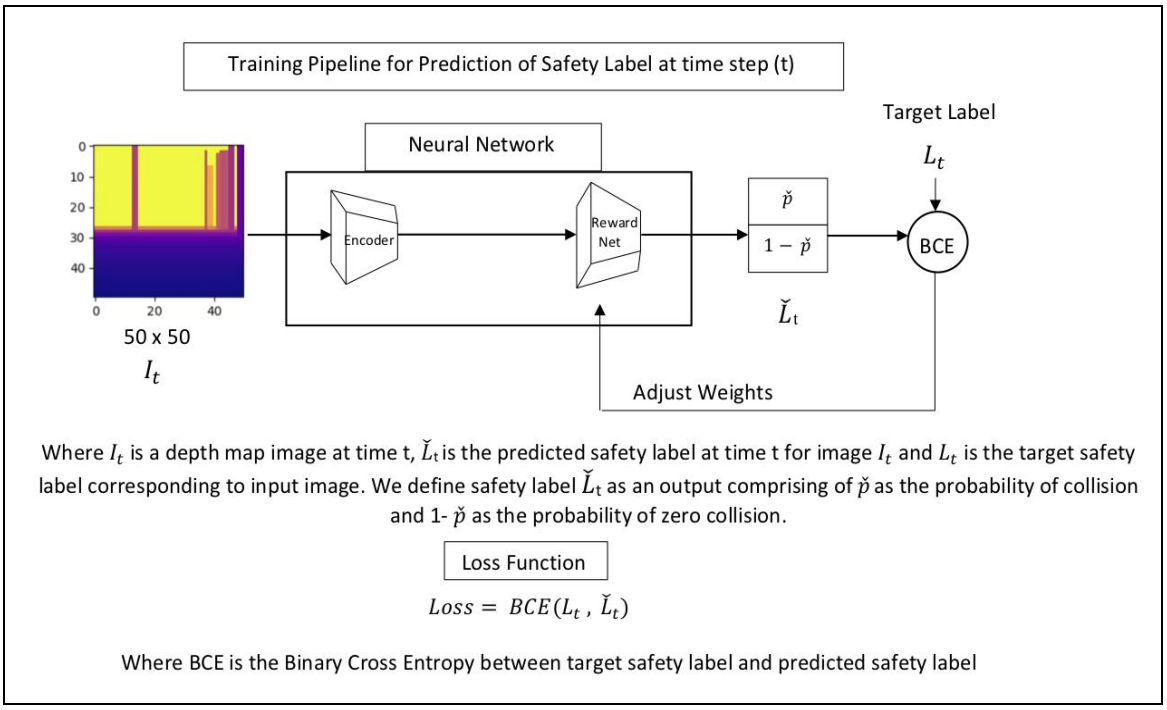


Figure 4: Illustration of training pipeline for prediction of safety label at time t that predicts probability of collision given a depth map image at time t as input

3.4 Design of Convolutional Neural Networks Architecture

The configuration of the convolutional neural network varies according to the complexity of the datasets. Configuring the size of each layer alters the capacity of the network. Generally, as we increase the size and number of layers in a Neural Network, the capacity of the network increases as the space representable functions grow and neurons can collaborate to express many different functions [15]. For the purposes of our neural network based motion planner, we model three different convolutional neural networks: Encoder, Decoder and Reward Network.

For specifying the network architecture, the software library Pytorch is used to define all the layers. The Pytorch package called Module allows us to call a number of different methods easily, where we first define the architecture in a class that inherits the properties from the base

class Module. The following classes are defined to perform data processing, training and validation:

I. Encoder

The Encoder class is designed to receive depth image data from the generated dataset. The encoder class outputs a single vector with dimension 15, representing the size of the latent space representation of the depth image input. The configuration of the neural network of the encoder is defined as:

- **Input** [1 x 50 x 50] will hold raw depth values of the image with a dimension of width 50, height 50 and depth values corresponding to grayscale values [0,1].
- **Normalization Layer 2d** will minimize internal covariate shift in data so the distribution of hidden units activation values remain the same during training [16].
- **Convolutional Layers 2d** [1, 32, 10, 2] and [32, 16, 8, 2] will process a tensor input with channel size = 1 by passing a 10 x 10 filter and a 8 x 8 filter over the image, performing dot products between filters to produce an output with channel size = 16.
- **Fully-Connected Layers** will perform linear transformation on the incoming input volume from the convolutional layer by flattening it into a single vector for input into the next layer.
- **Activation functions (Exponential Linear Unit)** will assign weights to every label with an offset bias. ELU is a good alternative to ReLU since it decreases the bias shift by pushing the mean activation towards zero [17].

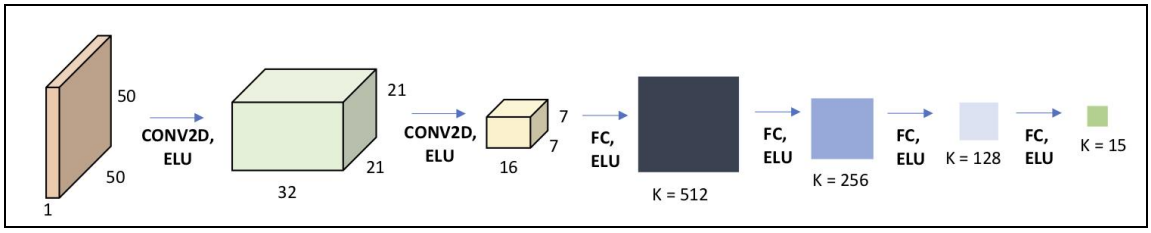


Figure 5: Illustration of CNN model for Encoder that receives the input image with dimension [1 x 50 x 50] and passes through a series of convolutional layers and fully connected layers where a 3D input image is flattened to a single vector followed by a bias offset.

II. Decoder

The Decoder class is designed to learn by receiving latent space representation input that has been concatenated with the motion primitive to reconstruct the depth map image at the next time step. The configuration of layers for the decoder is described as:

- **Input** [K = 16] represents the latent space representation of the input image that has been concatenated with the applied motion primitive at time t.
- **Fully Connected Layers** perform linear transformation on the latent space representation and increase capacity of layer after every transformation by increasing layers of network denoted by K in Figure 6 below. The last fully connected layer outputs

a reconstructed image with dimension 50 x 50, similar to the dimension of the input passing through the encoder.

- **Activation functions** (ELU and Sigmoid) together will shift the mean activation closer to zero and perform faster learning and convergence.

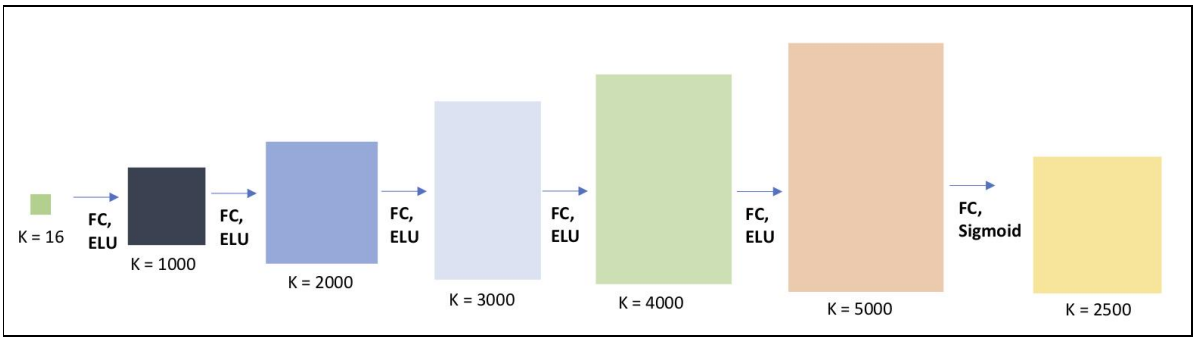
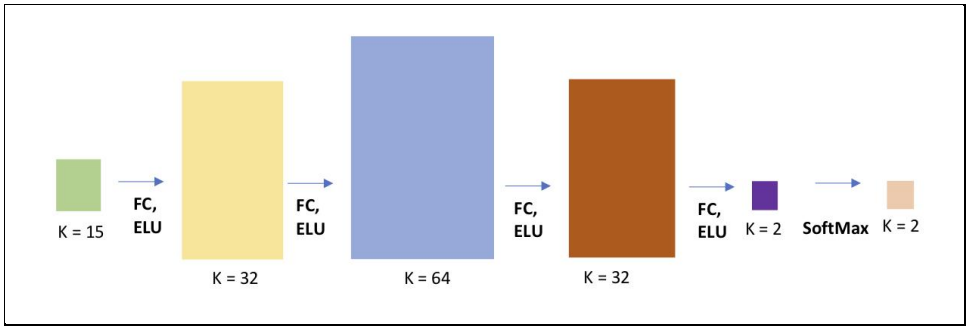


Figure 6: Illustration of CNN model for Decoder that receives a latent space representation with dimension 15 and passes through a series of convolutional layers and fully connected layers to reconstruct the depth map image at the next time step

III. Reward Network

The Reward Network class is designed to learn from a depth map image of the robot at time t with reduced dimensionality (latent space representation) and output a probability value that corresponds to probability of collision and probability of no collision. The input dimension of image after passing through the Encoder is equal to 15 and the output of Reward Network has dimension 2 which represents $\widehat{p}$ and $(1 - \widehat{p})$. The configuration for Reward Network is defined as:

- **Input** [K = 15] is the latent space representing parameters of the input image.
- **Fully Connected Layers** perform linear transformation on the latent space. Capacity of the network layer is expanded after every layer and further reduced down to dim = 2.
- **Activation functions** (ELU and Softmax) ELU will bring the activation mean closer to zero and speed up learning, while Softmax converts numeric output from FC layer to



probability values representing $\widehat{p}$and $(1 - \widehat{p})$.

Figure 7: Illustration of network model called Reward Net that outputs a safety label in the form [p, 1-p] where p is the probability of collision. This output is then compared with the target safety label, which is either [1, 0] (collision) or [0, 1] (no collision).

3.5
Design of Motion Planner

After successfully training the Encoder, Decoder and the Reward Network, we can construct a motion planner that, for each time step, chooses the motion primitive that corresponds to the lowest probability of collision in a long time horizon. The algorithm given below achieves this goal in a brute-force manner by applying each primitive motion in the library of primitives and assigning a score to them. Notice that at line 6, we have a for loop that indicates that in order to choose the motion primitive that will give the best result in a long time horizon, we are looking ahead for *look_ahead* number of time steps. The constant *look_ahead* would depend on how many time steps our neural network pipeline can make reasonably accurate predictions and it would usually be determined experimentally. Similarly, since our predictions would become less accurate as we look further ahead, at each time step in *look_ahead,* the associated weight gets halved (line 15).

Algorithm 1 Husky NN-based Motion Planner

```
for t = 0, …, T_Horizon do
    Get Image: I_t ← Simulation
    z = Encoder(I_t)
    Initialize: prim_scores ← ∅
    Initialize: weight = 1
    for i = 0, …, look_ahead do
        z_list ← ∅
        for prim in prims_lib do
            I_t+1 = Decoder(concat(z, prim))
            z_list[prim] = Encoder(I_t+1)
            prim_scores[prim] += weight * Reward_Net(z_t+1)
        end for
        prim_min ← index of min(prim_scores)
        z = z_list[prim_min]
        weight = weight / 2
    end for
    prim_best ← index of min(prim_scores)
    Apply prim_best → Simulation
    if collision then
        stop
    end if
end for
```

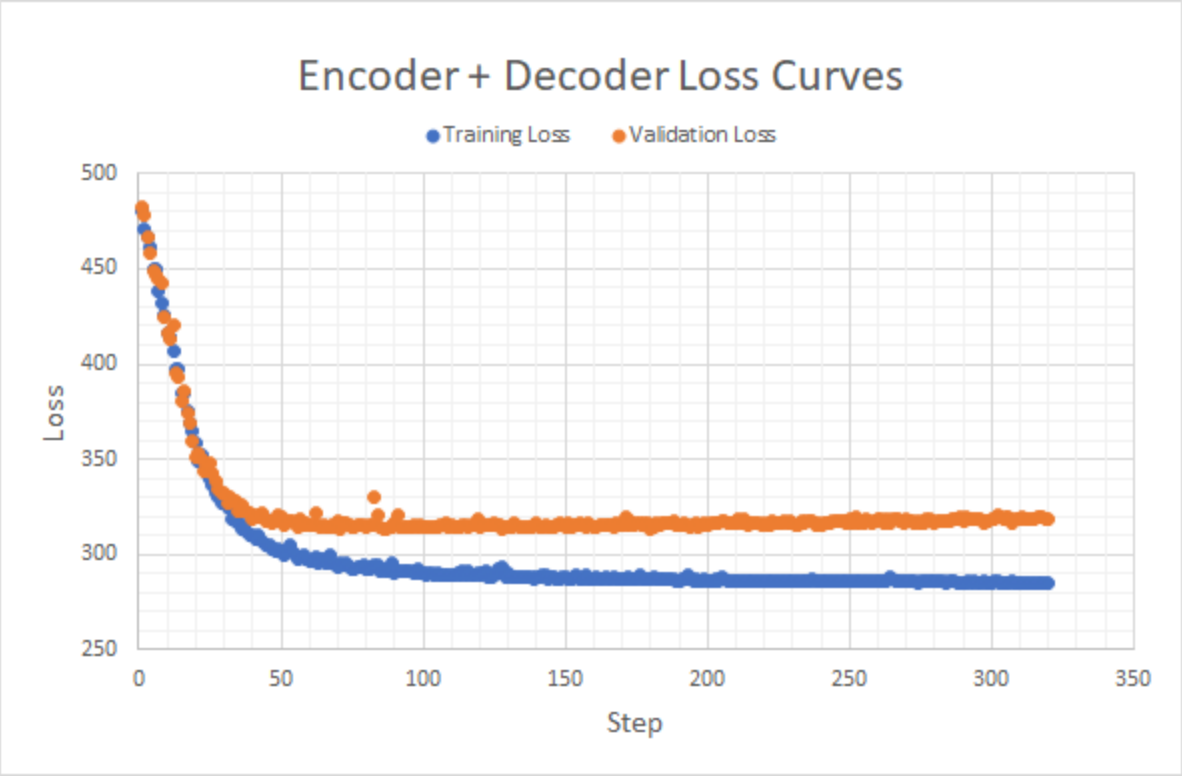
Finally, the performance of this motion planner will also be an additional validation method for the entire training process. Given that the random policy that was used during the data collection could avoid collision for 4.4 time steps on average, anything above this value would be a successful result.

4. Results and Discussion

This section will report the results of the training procedure and discuss core findings.

4.1 Encoder and Decoder Training Result

The first step of the training procedure was to train the Encoder + Decoder pipeline (see Figure 3). Using a batch size of 32 and learning rate 1e-5, we started training for 5000 epochs and stopped it when the loss curves seemed to be leveling off. The training continued for 3200 epochs, which took approximately 12 hours. The training was monitored using the Tensorboard visualization kit throughout the procedure and the resulting loss curves are shown in Graph 1 given below.



Graph 1: Encoder + Decoder Pipeline Training Loss and Validation Loss Curves

As it can be seen in Graph 1, both the training loss and the validation loss initially decrease at a reasonable rate and they converge to local minima (minimum points of the loss functions in the local region) as the training starts to level off. The reasonable steepness of the initial decrease indicates that our training did not suffer from overfitting or underfitting. Most importantly, the fact that the validation loss decreases along with the training loss validates the learning procedure of the neural networks. Given these observations, we inferred that the training of the Encoder + Decoder pipeline was valid and we proceeded with further visual tests where we used the trained networks on a completely new dataset.

A new dataset consisting of 20 images was generated using the same procedure explained in Section 3.1. The trained weights were loaded into the Encoder and Decoder networks, which were in evaluation mode (not updating weights), to test the accuracy of the predicted images by visually comparing them to the reference images. To make the comparison easier, we used a perceptually uniform sequential colormap to illustrate the depth map images. Figure 8, given below, shows 3 cases where we have an initial image at time t (left), the following image at time t+1 after applying a motion primitive (middle), and our trained neural network pipeline's prediction of the image at t+1 (right).

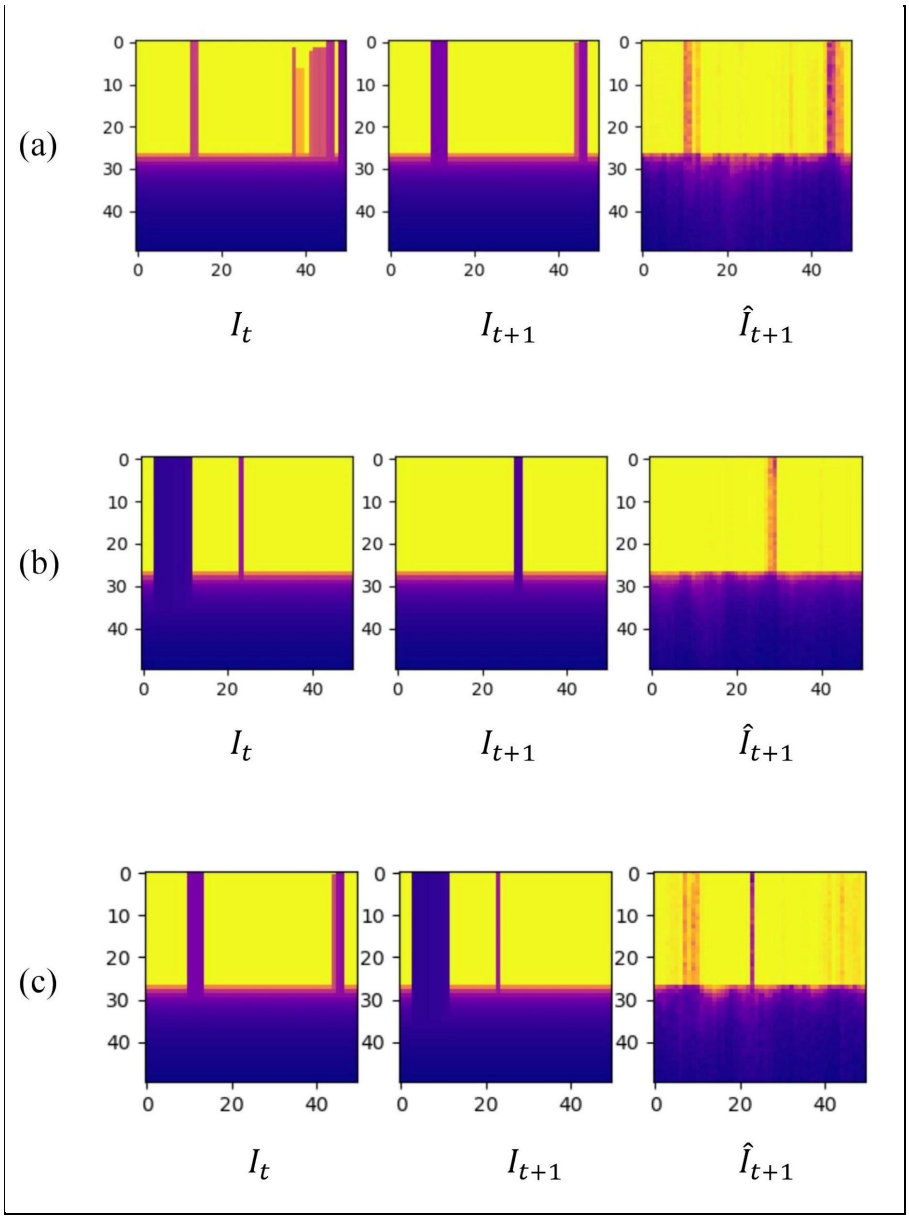


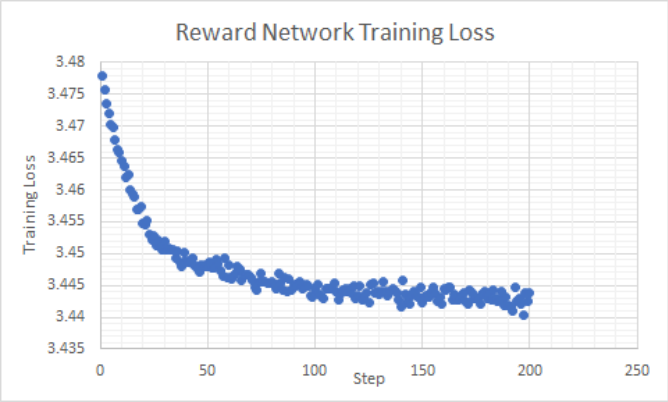
Figure 8: Further Visual Tests to Validate the Encoder + Decoder Training

The
Encoder + Decoder pipeline makes reasonably accurate predictions even for data that has not been used in the training. Figure 8, given above, shows how the predicted images can very accurately depict the locations of the obstacles in the x-y plane. However, as it can be told by the colors of the obstacles (which indicate the depth value in the z direction), our neural network pipeline was not so successful at predicting how far away the obstacles are. For example, in Figure 8(b), the obstacle has a blue color, which, according to the colormap we used, indicates that it is very close to the robot. The predicted image, on the other hand, has an orange obstacle that indicates that the obstacle is further away. Overall, with these observations, we

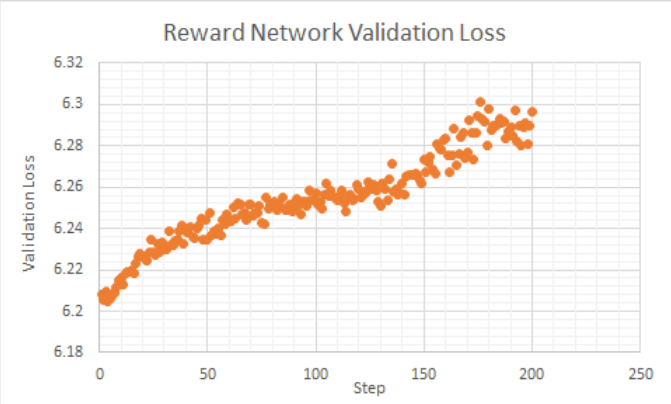
inferred that the Encoder + Decoder pipeline was accurate enough and we moved on to the second step of the training procedure.

4.2 Reward Network Training Result

The second step of the training procedure was to train the Reward Network (see Figure 4) that predicts the safety label (possibility of collision) given a latent space representation of a depth map image. For producing the latent space representations of the depth images, the trained Encoder from Section 4.1 was used in evaluation mode. Using a batch size of 16 and learning rate 1e-3, we started training for 5000 epochs. The training ran for 2000 epochs, which took approximately 6 hours. Once again, the training was monitored using Tensorboard throughout the procedure and the resulting loss curves are shown in Graph 2 and 3 given below.



Graph 2: Reward Network Training Loss Curve



Graph 3: Reward Network Validation Loss Curve

As Graph 2 demonstrates, even though the training loss curve has a reasonable shape, it only decreases from around 3.48 to around 3.44 in 2000 epochs. Similarly, the validation loss slowly increases during the entire training procedure. With these observations, we can clearly see that the Reward Network is not able to learn how to predict the safety labels. Some simple modifications such as changing the learning rate or modifying the size of the neural network's layers did not help with improving the training performance. Therefore, we came to the conclusion that there was an inherent mistake in our training method, Unfortunately, due to the limited time for the project, we were not able to adapt to a new training approach, but we will now discuss what might have gone wrong with our current method and how we can possibly improve it in the future.

As discussed in Section 4.1, our Encoder + Decoder pipeline was not quite successful at predicting how far away the obstacles are. Since the training of the Reward Network relies on the assumption that the Encoder can successfully output a latent space representation that contains all the relevant information about the input image, a failure in the Encoder would

significantly affect the performance of the Reward Network. Based on these observations, we are predicting that the trained Encoder's inability to successfully "encode" the depth values of the obstacles in the latent representation resulted in a failure for the Reward Network's training. In other words, the two-step training method we executed was not a good approach since the Encoder did not have a knowledge of our intention to predict safety labels, and therefore the latent space representations might not include enough relevant information for the Reward Network training. In order to overcome this issue, we could train a single end-to-end neural network pipeline that includes the Reward Network as well as the Encoder and the Decoder. Figure 9 given below illustrates this training approach. By training all three neural networks together and backpropagating on a total loss that includes the image reconstruction loss as well as the safety label loss, we are hoping to get better training results. That being said, the following section further explains alternative training methods and discusses how they can be used to give improved performance in future work.

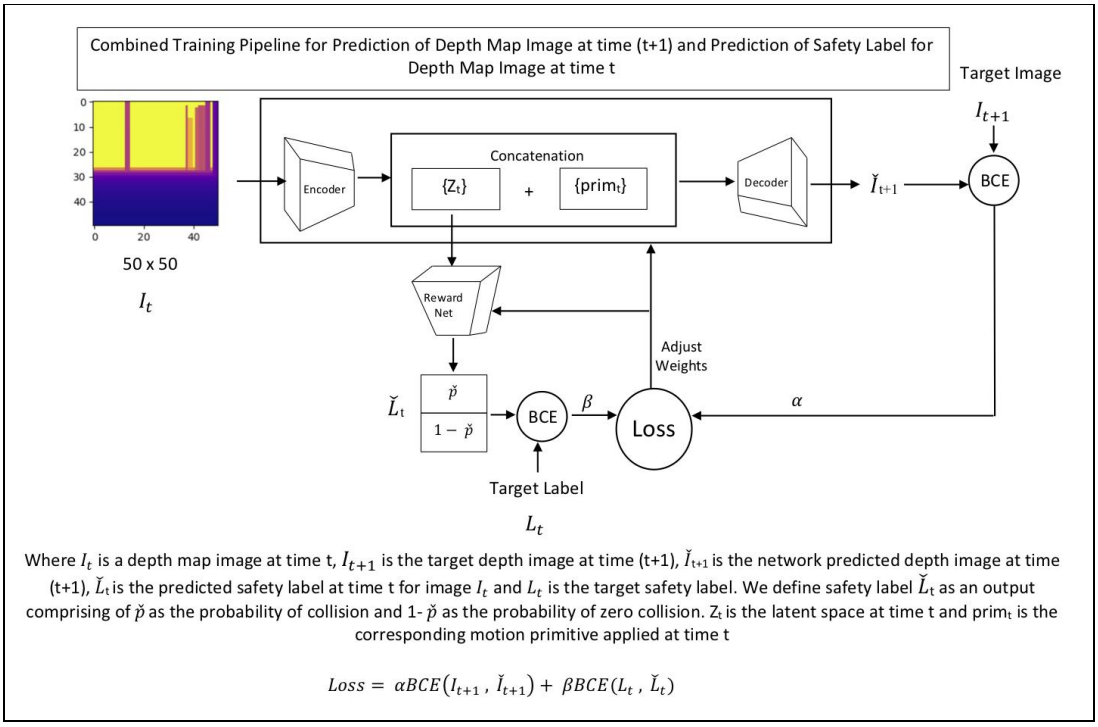


Figure 9: Illustration of training pipeline for the combined approach where prediction of the images and the prediction of safety labels happen in the same training

5. Future Work

One of the most important goals of implementing a neural network based end-to-end approach was to design a motion planner that can "look ahead" of the current time step to make decisions that can minimize the possibility of collision in a longer time horizon. In Section 3.4, we discussed how the motion planner can "look ahead" for a set number of time steps that depends on the neural network pipeline's ability to make reasonably accurate predictions for the future steps. Our current training approach is designed to learn how to reconstruct the depth map

image only at the next time step, while a better but more complex approach would involve reconstructing depth map images for future time steps as well. Diagram given below describes such a neural network pipeline design. A batch of data for this neural network pipeline would include all of the depth map images and corresponding motion primitives for each single video up until the point of collision. Note the use of an intermediate Recurrent Neural Network (RNN) in the training pipeline. RNNs are a class of neural networks that allow previous outputs to be used as inputs while having hidden layers, and therefore, they work better for cases where we have a series of sequential data [18].

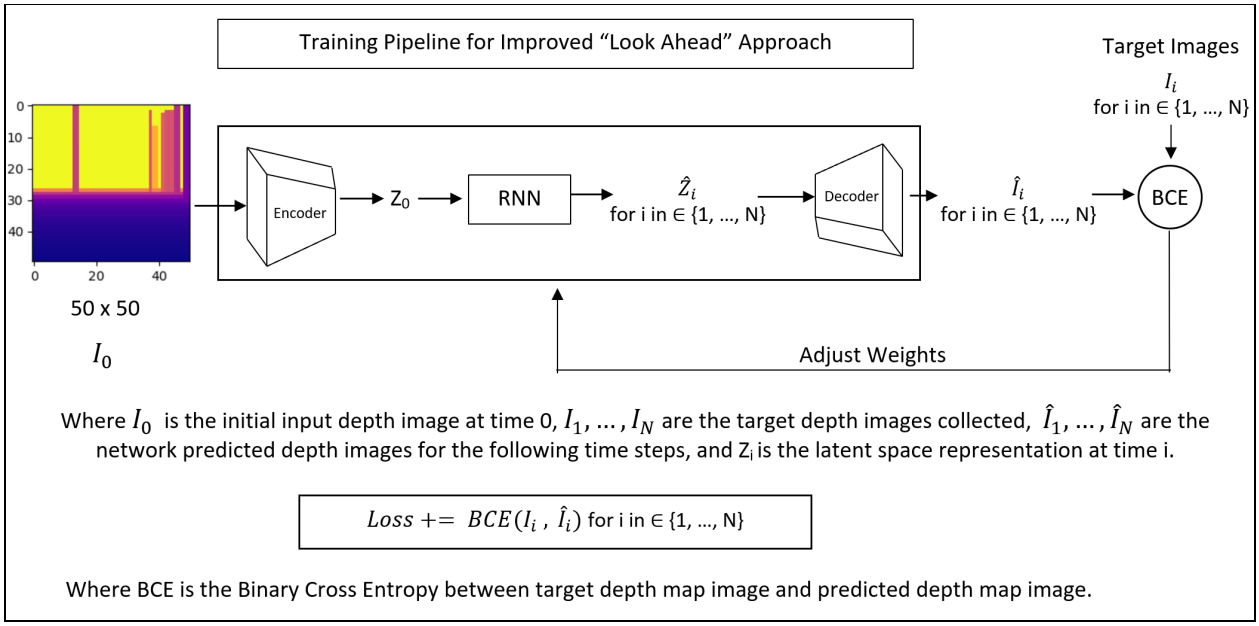


Figure 10: Illustration of the improved "look ahead" training approach

6. Conclusion

The aim of this project was to provide an end-to-end solution to the mobile robot navigation problem in obstacle dense environments by constructing a neural network based motion planner that can choose sequences of motion primitives to avoid collision in long time horizons. The first step of our learning approach was to train a neural network pipeline that can take an input depth map image as well as the applied motion primitive and construct the image at the next time step. As the network's performance on the validation data and further visual inspection tests proved in Section 4.1, we were able to achieve this goal to a reasonable degree of accuracy.

The second step of the procedure was to train a neural network that takes a latent space representation of the depth image and outputs a safety label that indicates the probability of collision in the next time step. As the results in Section 4.2 showed, we were not able to achieve this goal and several alternative training methods have been discussed in Section 4.2 and Section 5.0. Given a longer project timeframe, the proposed alternative approaches can be implemented to improve the results of training. Following a successful validation of the training results, one can construct the motion planner discussed in Section 3.5 and run it on the same computer simulation used in data collection. Essentially, the performance of the motion planner in this simulation would be the most important criterion used for determining the effectiveness of the learning based end-to-end solutions to the navigation problem. In further studies, a similar approach can be used to solve the navigation problem in different environment simulations as well as real world applications. One particularly interesting research topic is the application of such a learning based motion planner in environments where obstacles are dynamically changing.

References

[1] A.Majumdar and S.Veer, " Probably Approximately Correct Vision-Based Planning using Motion Primitives," arXiv preprint ArXiv:2002.12852, 2020

[2] S.Thrun, D.Fox and W. Burgard, "Mobile Robot Localization," *Probabilistic Robotics pp.157-158 1999-2000.*

[3] S.Thrun, D.Fox and W. Burgard, "Occupancy Grid Mapping," *Probabilistic Robotics pp. 221-230 1999-2000.*

[4] S.Thrun, D.Fox and W. Burgard, “Simultaneous Localization and Mapping,” *Probabilistic Robotics pp. 245-256 1999-2000.*
[5] H. Choset, K. M. Lynch, S. Hutchinson, G. Kantor, W. Burgard, L. E. Kavraki, S. Thrun, “Principles of robot motion: theory, algorithms, and implementations”. In Boston: MIT Press, 2005.
[6] A.Majumdar and S.Veer, “ Probably Approximately Correct Vision-Based Planning using Motion Primitives,” arXiv preprint ArXiv:2002.12852, 2020
[7] A.Majumdar and S.Veer, “ Probably Approximately Correct Vision-Based Planning using Motion Primitives,” arXiv preprint ArXiv:2002.12852, 2020
[8] K.Hauser, T.Bretl, K.Harada, J. Latombe, “Using motion primitives in probabilistic sample-based planning for humanoid robots,”
[9] F.Li, “Module 1: Neural Networks Part 1: Setting up the Architecture,” *CS231n Convolutional Neural Networks for Visual Recognition [Course Notes],* Stanford University, Spring 2020,
[10] F.Li, “Convolutional Neural Networks: Architectures, Convolution/Pooling Layers, Module 2: Convolutional Neural Networks, *CS231n Convolutional Neural Networks for Visual Recognition [Course Notes],* Stanford University, Spring 2020
[11] F.Li, “Convolutional Neural Networks: Architectures, Convolution/Pooling Layers, Module 2: Convolutional Neural Networks, *CS231n Convolutional Neural Networks for Visual Recognition [Course Notes],* Stanford University, Spring 2020,
[12] J.Leskovec, A.Rajaraman and J.D Ullman, “Chapter 13 Neural Nets and Deep Learning,” *Mining of Massive Data Sets* 3rd ed. pp 509-550, Cambridge University Press, December 2014
[13] J.Leskovec, A.Rajaraman and J.D Ullman, “13.2.7 Loss Function”, Chapter 13 Neural Nets and Deep Learning, *Mining of Massive Data Sets Third Edition* pp 522, Cambridge University Press, December 2014
[14] L.Johnson, “Loss Functions” *Theanets 0.7.3 Documentation* 2015
[15] F.Li, “Module 1: Neural Networks Part 1: Setting up the Architecture,” *CS231n Convolutional Neural Networks for Visual Recognition [Course Notes],* Stanford University, Spring 2020
[16] J. Bjorck, C. Gomes, B. Selman, K. Q. Weinberger, “Understanding Batch Normalization” arXiv preprint arXiv:1806.02375 Cornell University, 2018
[17] D. Pedamonti, “Comparison of non-linear activation functions for deep neural networks on MNIST classification task” arXiv preprint arXiv:1804.02763 University of Edinburgh 2018
[18] A. Ng and K. Katanforoosh, “Module C5M1: Recurrent Neural Networks”, *CS230 Deep Learning [Course Notes]*, Stanford University, Spring 2020