Transfer Learning for Mapless Quadrotor Navigation Using Recurrent Neural Network

Liyuan Hsu

Master Thesis
October 2018

Supervisors:
Stefan Stevšić
Prof. Dr. Otmar Hilliges
Prof. Dr. Moritz Diehl

Abstract

We propose two deep recurrent neural network architectures (reinforcement learning and supervised learning) to solve quadrotor obstacle avoidance and navigation problems. First, training these neural networks only in simulation environment, they are able to directly transfer into real world without any fine-tuning. Both models achieve navigation tasks with success rate over 90%. Second, we show the generalization ability of these models. Training on few simple environments and transferring directly into unseen complex environments, both models perform navigation success rate up to 90%.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Introduction

In recent years, more and more applications of micro aerial vehicles (MAVs) rely on real-time collision avoidance technique or require safe navigation and trajectory planning abilities. Traditional navigation methods [3] use Simultaneous Localization And Mapping (SLAM) [4] to acquire environment map by processing the sensor measurements in robot itself. Or use model-based methods such as Model Predictive Control (MPC) and motion primitive library search method [5]. These methods need a specific dynamics model and require real-time estimation of obstacle positions from sensor data. Although they perform more aggressive maneuvers and behave more reliable on disparate environment conditions, time-consuming and convoluted computational demands are two major concerns for these approach.

Deep reinforcement learning (RL) based navigation has made significant progress in recent years [6] [7]. A learned controller yields control policy directly from environment sensor input without estimating obstacles and constructing the map. With model-based RL [8] [9], quadrotor perform dangerous maneuvers to learn model parameters, which causes safety issues and damage the equipment. Here we use model-free RL which generate the discrete motion actions directly from range finder inputs without insight of model dynamics for quadrotor and environment. Also, model-free RL doesn't require continuously estimating obstacle position and generating navigation map, which largely reduces computation requirement. Another alternative to avoid tedious efforts of estimating obstacle position and realizing model dynamics is imitation learning, which learns desired control actions from an oracle [10].

To tackle mapless end-to-end navigation problems, we first train a easy 1D forward navigation model with fully connected RL algorithm. Using 180° sparse front facing laser range finder as environment demonstration, this model generates the discrete actions to guide the quadrotor at each time step. It can easily performs 1D forward navigation with obstacle avoidance tasks. And learned knowledge can be transferred from virtual to real world without fine-tuning.

However, fully connected RL model fails on 2D complex navigation tasks with randomized start and goal positions. We further implemented a reinforcement learning model with recurrent neural network (A3C_LSTM) and a imitation learning model (SL_LSTM) which mimic the moving action generated by a path planner supervisor. To compare the reinforcement learning and supervised learning navigation models, We test both models on simple navigation maps with same shape of obstacles and large open space.

With success rate over 90% in both models, we move further to build three complex navigation environments with diverse obstacles and small open space. Although navigation success rate also up to 90% in both models, they behave distinctively in failure cases.

To further improve the performance of navigation models, we evaluate model generalization ability for unseen environment. Domain randomization is shown success for training a generalized model [11]. We use same idea to train the navigation models in different maps, and test it on unseen maps. The learned knowledge can be transferred from one map to another, and model can also combine the navigation knowledge from more than one maps.

2

Related Work

2.1 Related Work

2.1.1 General navigation problems

General robotics navigation use Simultaneous Localization And Mapping (SLAM) [4] [3] which needs to build real-time environment maps by obstacle position estimation. This method is high computational demanding with complex control architecture. To avoid extra map estimation works and reduce computation time, we use end-to-end training algorithm to generate a control policy directly from environment demonstration.

2.1.2 Reinforcement learning for navigation

In recent years, Reinforcement learning (RL) has shown great achievements either in aerial vehicles control[12] [9], playing video games [13] or control robot arms[8]. For benefits of low computation power and simpler model architecture, we use RL algorithm with fully connected neural networks to train an end-to-end mapless navigation model with sparse range findings as input. It is also possible to transfer the learned model from virtual to real world without any fine-tuning [6]. However, robot might get stuck in more complicated and compact environments, because the range finder only provides distance information which might be similar in different locations. This problem is solved by adding Recurrent neural network (RNN) to help navigation model learn the temporal dependencies of environment [14][15].

2.1.3 Supervised learning for navigation

Instead of training a navigation model from scratch by RL, an alternative is to use SLAM [16], or learning path following and obstacle avoidance behavior from imitation learning [17]. To avoid extra path generating works, it is possible to train a navigation controller which directly calculates the available actions at each environment state. [18] proposed an end-to-end motion planning for autonomous ground robots. They first generate numbers of navigation trajectories from motion planner, and use these trajectories as training data to teach a policy controller which maps environment states into control actions. Yet in our project, we move further to improve environment generalization ability on top of supervised learning navigation model.

2.1.4 Transfer learning

Transfer learning is a method to train the learning model in one domain but carry out tasks in another domain. For example, transferring from simulation to real world, or transferring navigation behavior from one map to another. For quadrotor collision avoidance, transfer learning is able to train a deep reinforcement learning control policy for indoor flight entirely in a simulated environment, and transfer the navigation knowledge to fly in real world with unseen RGB images as input [19] [20]. We use same technique to transfer the navigation control policy directly from simulation to real world, but replace the RGB images by laser range finder which needs less data pre-processing efforts while training the model. Besides, domain randomization is an idea to increase data diversity in training domain, which performs a better generalization ability in test domain. Take robot grasping for example, train a model in simulation with randomized objects achieves over 90% success rate in real world with unseen objects[11] [11]. Likewise, we use domain randomization idea to train a navigation model in diverse simple maps and transfer the knowledge to unseen and more complex environments.

3

Preliminaries

We provide basic knowledge about Reinforcement learning and Supervised learning, which are the fundamental to build different quadrotor navigation models in next chapter.

3.1 Reinforcement Learning

Reinforcement learning, different from supervised learning, is a learning method without any correct labels provided by a supervisor. An agent directly interact with the environment and learn from the environment. Take a baby for example, he always try to touch and interact with any objects surrounding him, and get some feedback from these interactions. In RL problems we use the following terms:

- Agent: The learner or decision maker
- Environment: The thing agent interact with, or every thing outside the agent
- State (S_t): The current situation information provide by environment
- Action $(A_t$: How an agent interact with the environment
- Reward (R_t): Feedback from environment after an action being done

First, we define an Agent, a Environment and a sequence of discrete time step, $t = 0, 1, 2, 3, ...$, in which the Agent will interact with the environment. Second, in each time step, agent perform an action to environment, and get the reward and new state from environment.

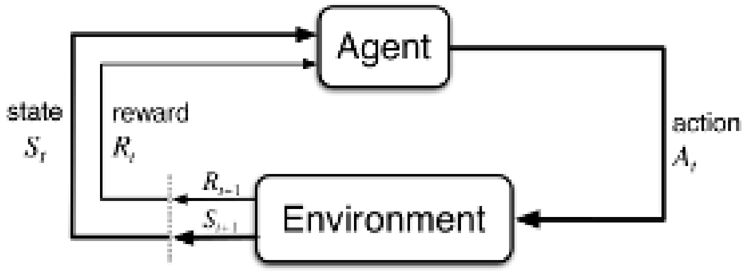

Figure 3.1: *Agent-environment relation [1]*

3.2 Asynchronous Advantage Actor-Critic Algorithm (A3C)

A3C, Asynchronous Advantage Actor-Critic [21], is a kind of reinforcement learning algorithm released by Google's DeepMind, which works in continuous and discrete action space. There are three key features in A3C algorithm listed below:

3.2.1 Asynchronous

In order to learn more efficiently, A3C algorithm creates more than one agent to interact with their own independent environment. Each independent agent has their own local network with the same network parameters which are copied from the global network in the beginning. The benefit of asynchronous method is that the experience learned by each agent is independent from other agents, which allows the training process more diverse and efficient.

3.2.2 Advantage

Value function is used to estimate how good is an agent in a given state. It is defined in terms of future rewards that can be gained by an agent, or in other words, the expected return for an agent in a given state. For a given policy π, we can define the state-value function $v_\pi(s)$ as

$$v_\pi(s) \equiv \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[\sum_{k=0}^{\infty}(r^k R_{t+k+1})|S_t = s] \tag{3.1}$$

where s is the current state, t is any time step, $\mathbb{E}_\pi[\cdot]$ is the expectation of a random variable given that it follows the policy π, G_t is return (cumulative discounted reward) following time t, S_t is the state at time t, r is the discount-rate parameter, R_t is the reward at time t.

Similarly, we can define an action-value function $q_\pi(s, a)$, as the expected return starting from certain state s, taking the action a, and following the policy π:

$$q_\pi(s,a) \equiv \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi[\sum_{k=0}^{\infty}(r^k R_{t+k+1})|S_t = s, A_t = a] \tag{3.2}$$

where a is the action taken along with state s and A_t is the action at time t. The action-value function $q_\pi(s,a)$ is also known as Q function.

After knowing the Value function and Q function, we can further define the Advantage function A to determine how much better an action turned out to be than expected at certain state s. This allows reinforcement learning model to focus on where it is lacking and improving the network parameters accordingly. This Advantage function A is simply define as the difference of Value function and Q function at certain state s:

$$A_\pi(s,a) = v_\pi(s) - q_\pi(s,a) \tag{3.3}$$

3.2.3 Actor-Critic

A reinforcement learning model can be trained on value-iteration methods or policy-iteration method. In A3C algorithm, the model combines the benefits in both approaches and estimate the Value function $V(s)$ (Critic) and Policy function $\pi(s)$ (Actor) at each time step. In this project, we use same network structure for both Value and Policy function except the last output layers are different. In each training step, the model use value estimation to update the policy function, which is more efficient that pure value iteration or policy gradient methods.

3.2.4 Training steps

In this project, the reinforcement learning model builds one global network and eight independent workers(local network) to interact with the environment and update global network parameters. The A3C algorithm has following five steps to train the model:

- 1. Worker reset to global network
- 2. Worker interacts with environment
- 3. Worker calculates value and policy loss
- 4. Worker gets gradients from losses
- 5. Worker updates global network with gradients

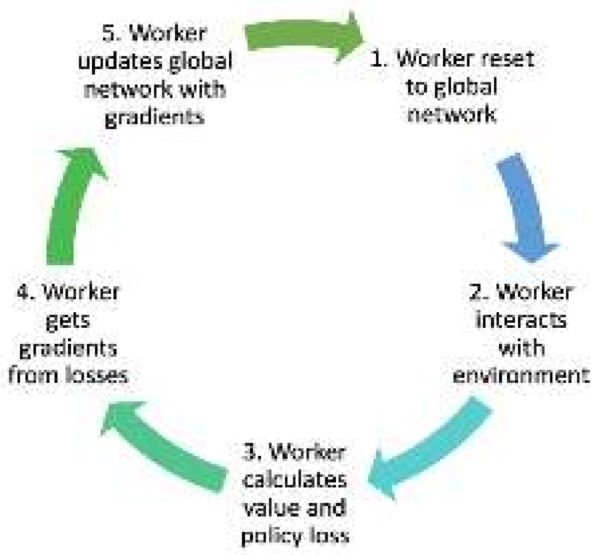
5. Worker updates global network with gradients
1. Worker reset to global network
2. Worker interacts with environment
3. Worker calculates value and policy loss
4. Worker gets gradients from losses

Figure 3.2: *A3C training steps [2]*

First, we create one global network and n local networks (workers) and reset all network parameters to the global one. Second, each local worker interact with their independent environment to gain experiences. Third, each worker calculates value and policy loss after episode finished. Fourth, each worker calculates the gradients from losses and finally, updates the global network parameters separately with those gradients. The value loss L_v and policy loss L_π are defined as:

$$L_v = \sum (R - V(s))^2 \tag{3.4}$$

$$L_\pi = -\log(\pi(s))A(s) \tag{3.5}$$

where R is the discounted return. To encourage model to explore environment more, we calculate an entropy of policy $H(\pi)$, which means the spread of action probability. If the probability of actions are relatively similar, the entropy will be high, but if there is only one action with very high probability, then the entropy will be low. The entropy of the policy $H(\pi)$ defines as:

$$H(\pi) = -\sum (\pi(s) \log(\pi(s))) \tag{3.6}$$

From the combination of Value loss L_v, Policy loss L_π and Entropy of policy $H(\pi)$, we can further calculate the total loss L as:

$$L = \alpha L_v + L_\pi - \beta H(\pi) \tag{3.7}$$

where α and β are loss parameters that can be tuned by users. After each worker calculate their independent loss by interacting with the environment, they obtain the gradient from loss. And finally each worker use gradient to update the global network parameters. After local network back propagate the gradient and update the global network parameters, a complete training step in a episode is finished.

3.3 Supervised Learning

In supervised learning, there is a supervisor providing the correct labels for each input data. And there is a model to produce the predicted results. In learning process, the model parameters are continuously adjusted to reduce the difference between "correct labels" and "predicted results", and increase the accuracy of the model. The supervised learning can be further divided into Regression and Classification problems, which means the output values are numerical and categorical, respectively.

In this project, we use softmax function and cross-entropy loss function to do the multi-class classification. The input data of distance readings and destination measurement will be classified into 8 discrete actions in each time steps. The combination of these actions will form a collision-free trajectory which lead the quadrotor from start position to goal position.

3.4 Transfer Learning

In typical machine learning topics, we assume that training and test process happened in the same domain. For example , in supervised learning, training set and test set are from the same distribution. Or in reinforcement learning, training data and test date are gathering from the same environment setting. However, this is rarely the real case. Most of time, it is difficult or time-consuming to perfectly match the training and test. Moreover, there are some other benefits for transfer learning as below:

- Time saving, training in simulation and test in real world.
- Risk reducing, training in simulation and test in real robot.
- Sharing learned knowledge

3.5 Dijkstra Path Planner

Dijkstra's path planning algorithm [22] finds the shortest path between two points. It is built and calculated under a graph structure, which contains a set of nodes connected a numbers of edges. The algorithm starts from the goal position and radiates outward to visit each available edges connecting with goal node. It keep updating the path length while exploring the graph and searching for new nodes until reach the start node. Finally, it finds a path with shortest edge length connecting from goal node to start node. A known graph structure or a map with environmental information is required for using this algorithm.

4

Method

4.1 Simulation

4.1.1 Training environment for reinforcement learning

In this project, we use Google TensorFlow[1] to build reinforcement learning network and trainer, and train the model in OpenAI[2] Gym environment, which is a toolkit for developing and comparing reinforcement learning algorithms. It supports the user to create virtual environments with different agents and obstacles. Agent interacts with environment while training the RL model. As mentioned in Section 3, training of a reinforcement learning model is done in an episodic way. In environment's step function, user has to define following 4 return values in OpenAI Gym environment. Observation is used to generate the action from policy function and reward will be cumulated to update the value function.

- observation (object): Observation of the environment from agent's view.
- reward (float): Reward achieved by the previous action.
- done (boolean): Whether the episode has terminated. (True if agent hit the obstacles or reach the goal)
- info (dict): Information used for debugging.

Below is a sample OpenAI Gym environment which will be used in following sections. For each training step in an episode, agent (quadrotor) starts from acquiring observation (laser dis-

[1]`https://www.tensorflow.org/`
[2]`https://openai.com/`

tance readings) from environment, and feed the observation into control policy to generate an appropriate action, and then enact this action and translate into next step. This process continues until episode finish or reach maximum episode length.

- Black square: quadrotor
- Blue dots: laser range finder
- Red cylinders: obstacles
- Green cylinder: goal position

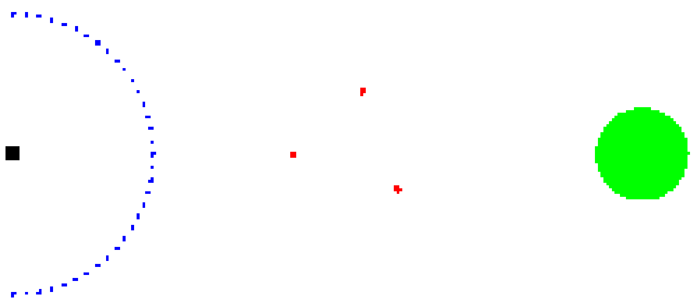
Figure 4.1: *OpenAI Gym environment*

4.1.2 Training environment for Supervised learning

To build a navigation model with supervised learning, there are following two steps. First, generate numbers of state-action pairs from motion planner oracle. Second, using data generated from first step as the correct labels to train a navigation model by supervised learning.

For generating state-action pairs, we build a 2D robot simulation environment [3] in ROS. And use ROS Navigation Stack [4] with Dijkstra path planning algorithm to generate 4000 navigation trajectories with random start and goal positions. In figure below:

- Blue circle: quadrotor at start position
- Red arrow: goal position
- Green line: shortest path from start to goal (navigation trajectory)
- Red dots: laser ranger finer

[3]`https://github.com/avidbots/flatland`
[4]`http://wiki.ros.org/navigation`

- Purple cylinder: obstacles
- Green and blue rings enclose obstacle: cost-map generated by motion planner showing high collision risk areas.
- White grid: environment scale with size $1m$

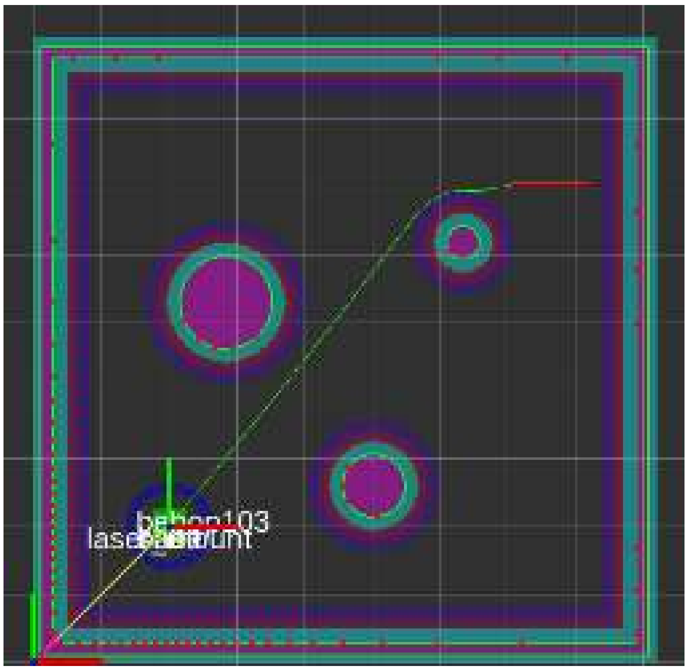

Figure 4.2: *Planner environment*

With generated navigation trajectory, we place quadrotor on start position and move it step by step (with action step size: $0.2m$) forward along trajectory until reach goal. And then collect environment state (laser distance reading, relative distance and angle of goal) and action at each step. Action is generated through directly mapping navigation trajectory to the closest discrete action as figure below. (Green path: navigation trajectory, Red arrow: discrete actions generated from navigation trajectory)

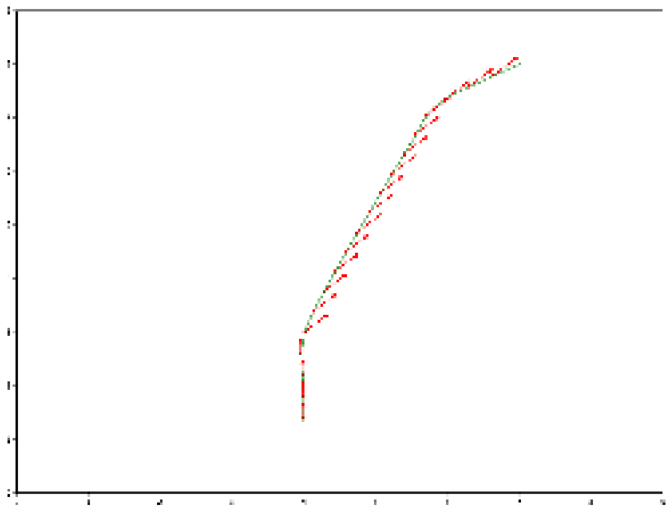

Figure 4.3: *Action mapping*

After collecting the state-action pairs from motion planner, navigation model is trained by supervised learning algorithm with these data as the correct labels.

4.1.3 Navigation Models

In this project, we test and compare 3 different end-to-end navigation models. First, A3C [21] deep reinforcement model with fully-connected layer. Second, we add one LSTM layer into first model and generate another model. Third, we use supervised learning to learn the correct actions from motion planner.

- A3C with fully-connected layer (A3C_FC)
- A3C with LSTM layer (A3C_LSTM)
- Supervised learning with LSTM layer (SL_LSTM)

Reinforcement learning model

We use reinforcement learning algorithm to build a simple model with only fully-connected layers (A3C_FC), and then we add one LSTM layer to this model to build second RL-based model (A3C_LSTM). These two models have 6 and 7 layers, respectively.

- Input layer
- LIDAR Fully-connected layer
- 1st Fully-connected layer
- (LSTM layer)

- 2nd Fully-connected layer
- Policy output layer
- Value output layer

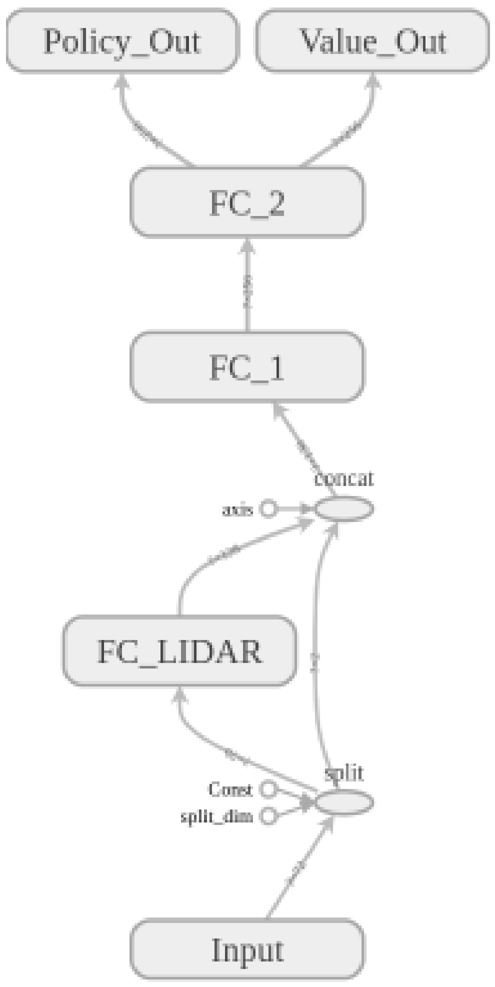

Figure 4.4: *A3C_FC*

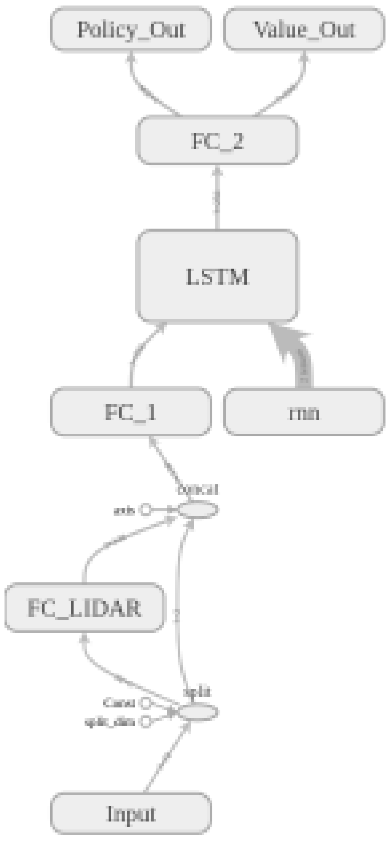

Figure 4.5: *A3C_LSTM*

Supervised learning model

SL_LSTM is exactly same as A3C_LSTM except the missing value output layer.

- Input layer
- LIDAR Fully-connected layer
- 1st Fully-connected layer
- LSTM layer

- 2nd Fully-connected layer
- Policy output layer

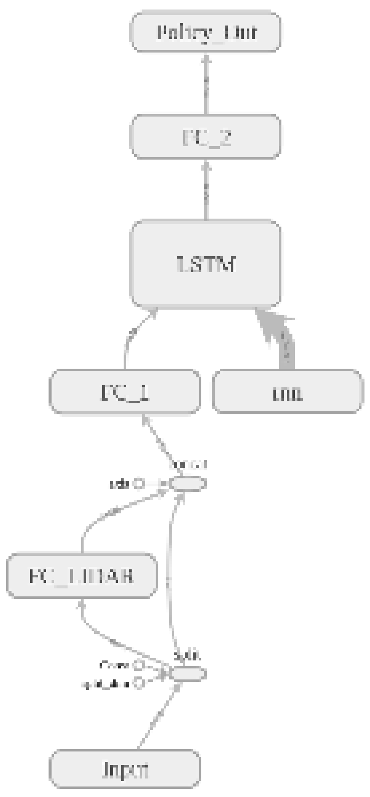

Figure 4.6: *SL_LSTM*

Learning temporal dependencies

Navigation is actually a temporal dependent problem, which means the action taken at last time step has influence on the action at current step. Therefore, we implemented A3C_LSTM and SL_LSTM models to tackle this problem. In following sections, we focus on comparing these two models.

4.2 Real World

4.2.1 Hardware

There are three parts of hardware used in this project. A PARROT BEBOP 2 quadrotor, a LIDAR range finder to detect the environment and a ODROID micro-controller to connect the LIDAR and PC.

PARROT BEBOP 2

We use PARROT BEBOP 2 quadrotor as the flying platform to test end-to-end navigation model in the real world. The quadrotor is shown in following figure and there are some features. [5]

- Connectivity: Wi-Fi 802.11a/b/g/n/ac
- Signal range: 300 m
- Max horizontal speed: 16 m/s
- Max upward speed: 6 m/s

Figure 4.7: *PARROT BEBOP 2*[6]

LIDAR

We mounted a Scanse LIDAR range finder on top of quadrotor to detect the environment information. The horizontal field of view is 360 degrees with 70 distance inputs per revolution. Following are some features for it. [7]

- Weights: 120 g
- Range: 40 m
- Resolution: 1 cm
- Update Rate: Up to 1075 Hz
- Horizontal Field of View 360 degrees

[5]https://www.parrot.com/
[6]https://www.parrot.com/
[7]http://scanse.io/

Figure 4.8: *Scanse LIDAR*[8]

ODROID

Because there isn't Wifi module on LIDAR ranger finder, an ODROID micro-controller is used to wirelessly transfer the distance data from LIDAR to PC.

Figure 4.9: *ODROID micro-controller*[9]

Quadrotor unit

As figure below, we use 3D printer to print a holding structure which directly mounted LIDAR ranger finder and ODROID micro-controller on to the top of quadrotor. The whole setup weights around 750 g. and it is able to fly around 10 minutes with a fully charged battery.

[8] `http;//scanse.io/`
[9] `https://www.hardkernel.com/`

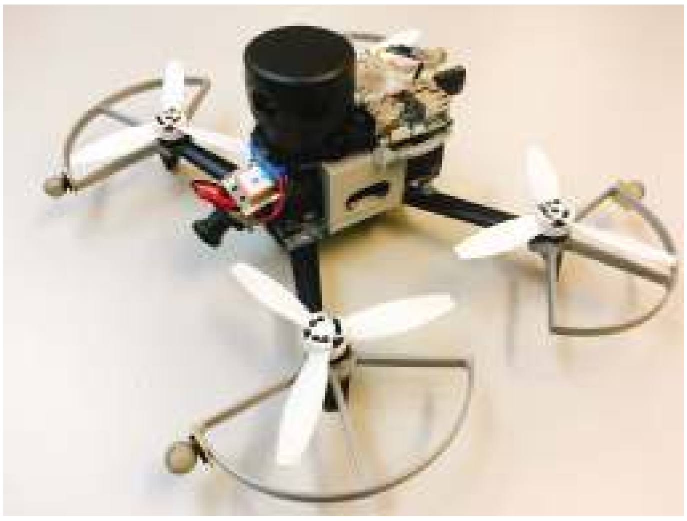

Figure 4.10: *Quadrotor Unit*

4.2.2 Environment Setup

Building in ROS [23] framework, the real world navigation model is conducted under vicon system to get the precise positions of quadrotor and obstacles. A PC is used to gather obstacle, quadrotor positions and distance readings and feed these date into navigation model to generate the action at each time step. From generated action and current quadrotor position, a new set position for quadrotor is calculated. Then, A simple PID position controller generate the yaw, row, pitch velocity commands according to the difference of set and current quadrotor positions. When quadrotor reach new set position, a time step is finished and the process starts from collecting distance readings again, until quadrotor reach the goal.

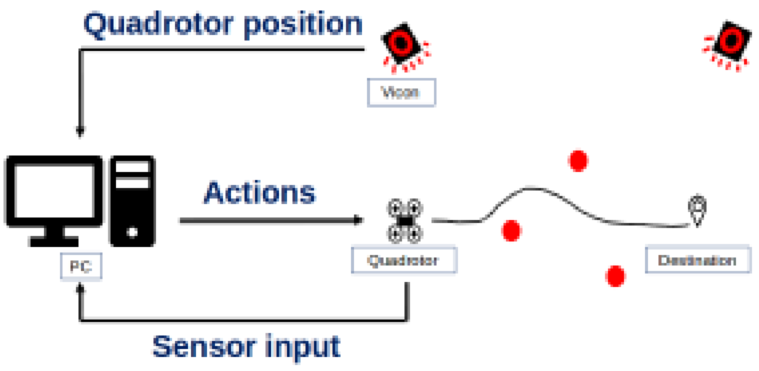

Figure 4.11: *Experiment Setup*

5

Experiments

5.1 1D easy forward navigation

We build a easy forward navigation setup to test the performance of A3C_FC model. The quadrotor always starts from same position, and move forward until cross the destination line. The obstacles in between have different shapes, sizes, numbers and positions. The navigation model maps environment demonstration into 5 discrete actions in each time step. These actions form a collision free trajectory guided quadrotor from start position to destination line.

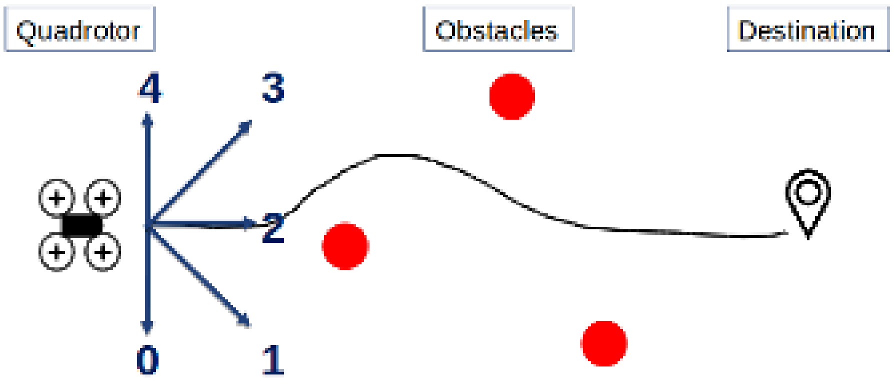

Figure 5.1: *Easy forward navigation*

5.1.1 Simulation environment setup

Under OpenAI Gym environment framework, First, we built a square object with size $0.2m \times 0.2m$ representing the quadrotor. Second, we put some obstacles between start and goal positions. Third, we created a range finder with 35 laser points with maximum range $2m$ mounted on the front of quadrotor to detect obstacles. Finally, we give quadrotor 5 discrete forward actions with step size $0.2m$. Below are detailed environment settings.

- Bounded box size: $13m \times 8m$
- Quadrotor size: $0.2m \times 0.2m$
- Number of actions: 5
- Step size of action: $0.2m$
- Number of laser readings: 35
- Field of view of laser readings: $180\deg$ (forward)
- Maximum laser range: $2m$
- Types of obstacles: Cylinder, Rectangle
- Reward (reach goal): 1.0
- Reward (collision): -1.0
- Reward (hit the bounded box): -1.0
- Reward (get closer to goal): 0.01

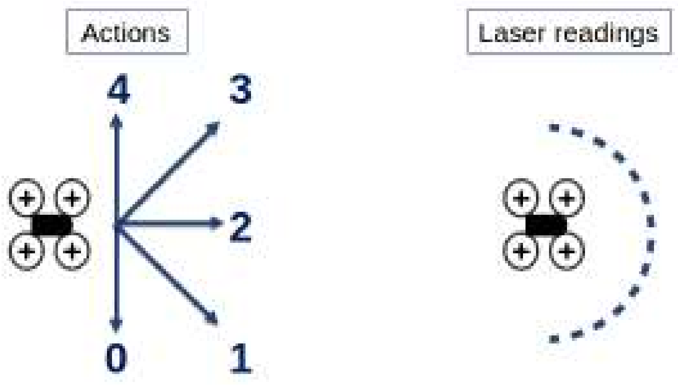

Figure 5.2: *Quadrotor actions and laser readings*

5.1.2 A3C model setup

As mentioned in section 4, the A3C_FC model has 35 laser reading inputs and 5 action outputs. In between are two fully-connected layers with 128 and 256 nodes.

We train the model from scratch with Adam [24] optimizer on one Intel Core i7-4790K CPU @ 4.00GHz with 8 parallel threads. To accelerate the training process, we bootstrap the model with current value estimation every 30 steps before the end of episode. Below are hyperparameters of this model:

- Bootstrap number: 30
- Maximum episode length: 300
- Reward discount rate: 0.99
- Learning rate: $1e-4$
- Input size: 35
- Hidden layer size: 128, 256
- Output size: 5

The training takes $12min$ with $1.2k$ steps for value function converging to near 1.0. The episode length is up to 32 steps at beginning, because model tries to explore the environment. After value function converges, episode length also converges to around 24 steps.

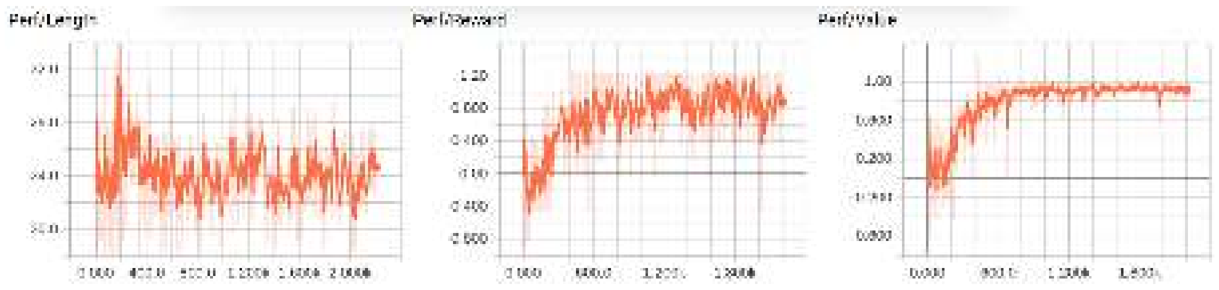

Figure 5.3: *Training result for easy forward navigation model*

5.1.3 Simulation result

A navigation process with 18 time steps is shown below. Green spot is goal, and quadrotor always starts from very left and moves rightward to reach the goal. In $t = 0$, the range finder doesn't detect anything so quadrotor chooses an rightward action. In $t = 6$, the range finder detects first obstacle and choose upward action to avoid it. After $t = 11$, there isn't any obstacle in front, so the quadrotor keep choosing rightward actions until reach the goal.

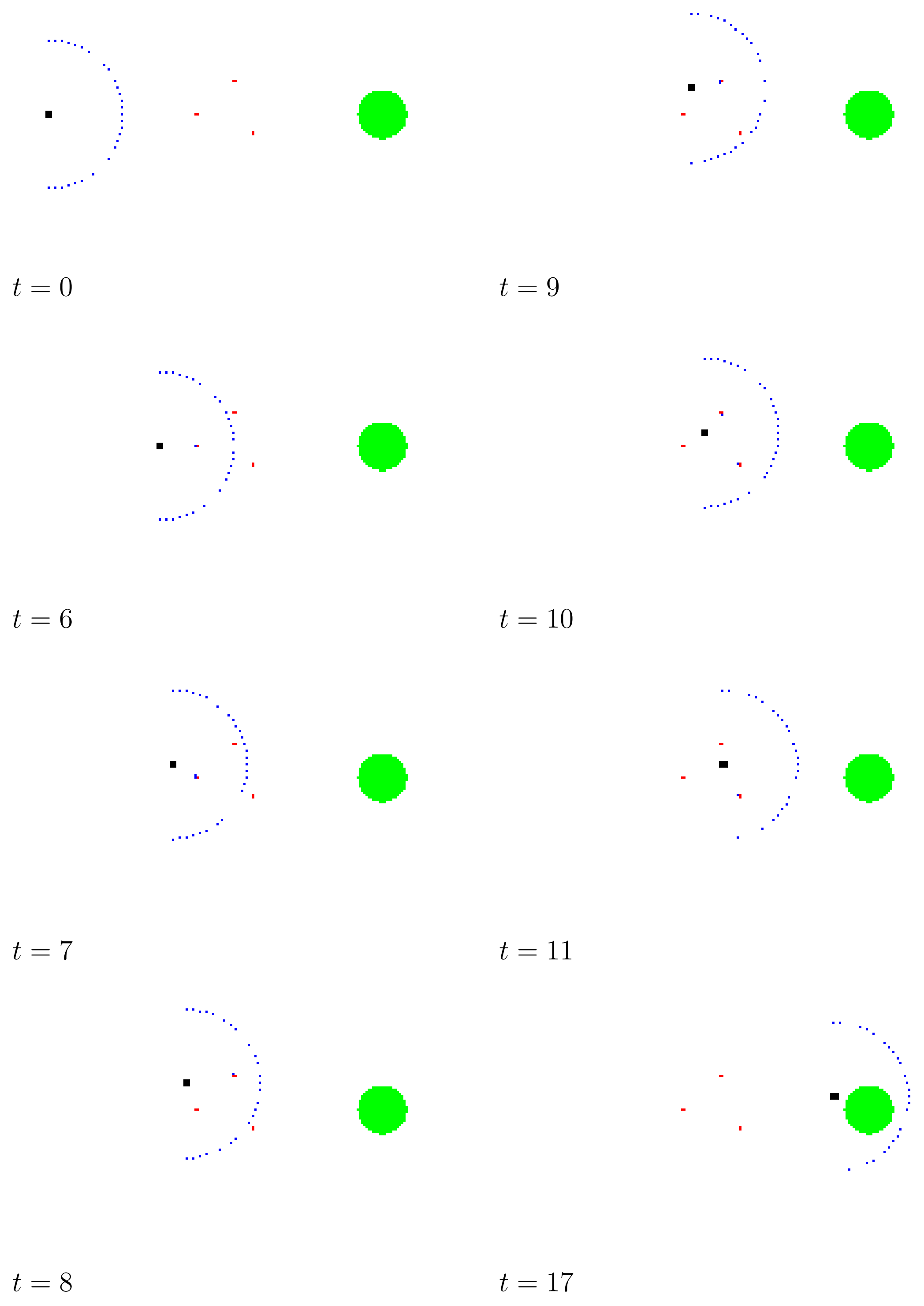

Figure 5.4: *Simulation result for easy forward navigation with 3 cylinder obstacles (up to down, left to right)*

5.1.4 Real world test

As mentioned in Section 4, we test the easy forward navigation with A3C_FC model in real world with 3 different cases. Although the distance readings from LIDAR sensor are not stable, the quadrotor is able to fly through the obstacles and reach the goal without any collision.

One obstacle

On down left shows quadrotor location (black square), obstacle location (red cylinder), action (black arrow), and laser distance readings (blue dots). With one cylinder obstacle, the quadrotor simply moved leftward to avoid the cylinder and moved forward to reached the goal.

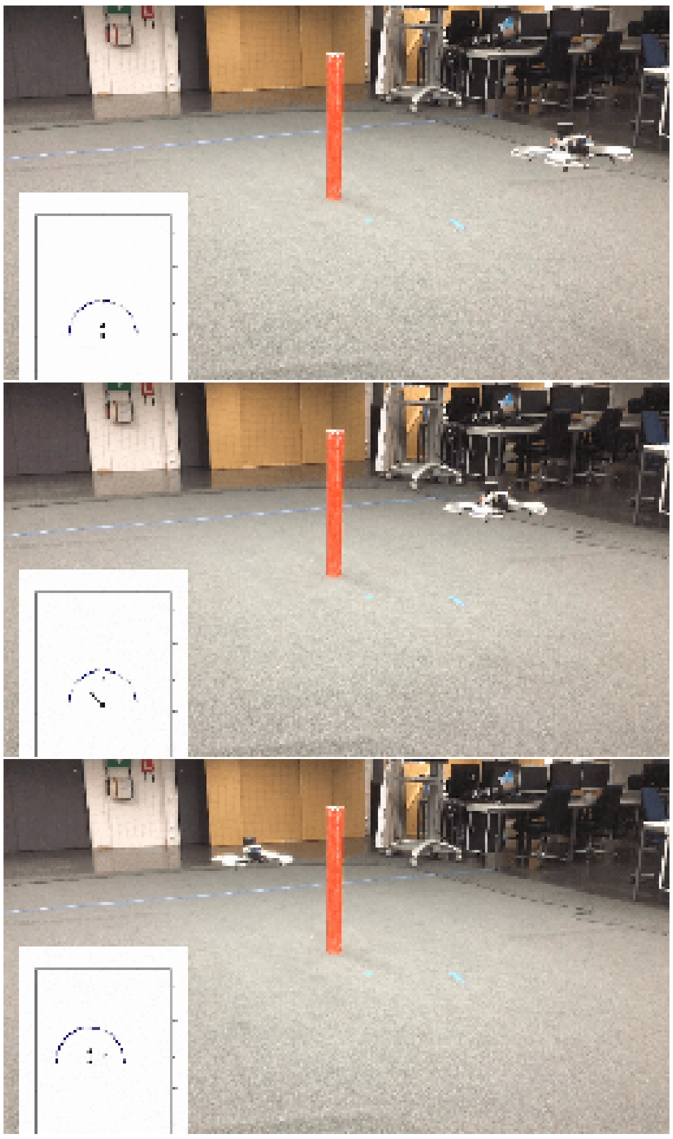

Figure 5.5: *Real world test on one cylinder obstacle*

Two obstacles

With two cylinder obstacles, quadrotor first moved leftward to avoid the first cylinder, and moved rightward to avoid second cylinder. Finally it kept moving forward to reach the goal.

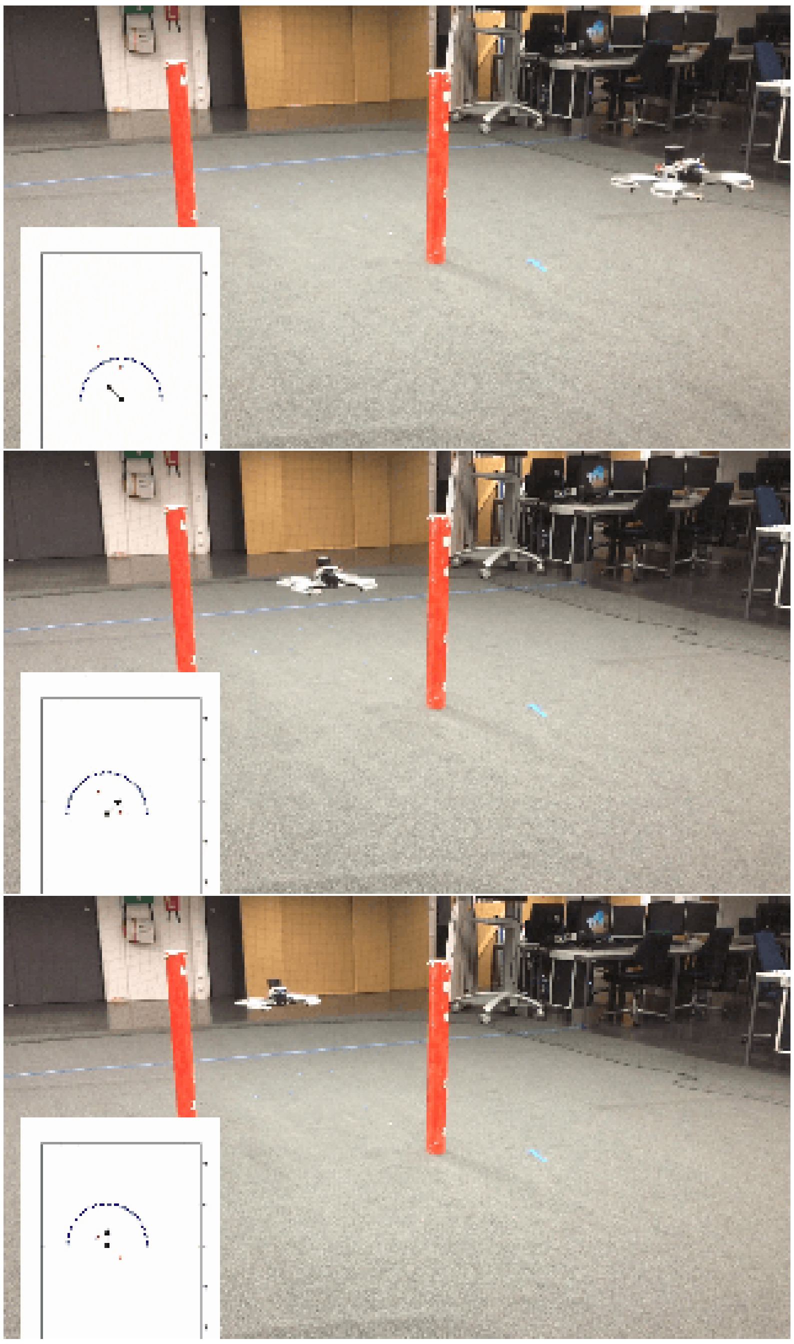

Figure 5.6: *Real world test on two cylinder obstacles*

Three obstacles

First, quadrotor chose rightward action because it detected two cylinders on the left side, and moved leftward to avoid third cylinder and then kept forward to reached the goal. From these three real world tests, it shows the ability of A3C_FC model to navigate in different obstacle arrangements. It also means navigation model can be transferred from virtual to real world without fine-tuning.

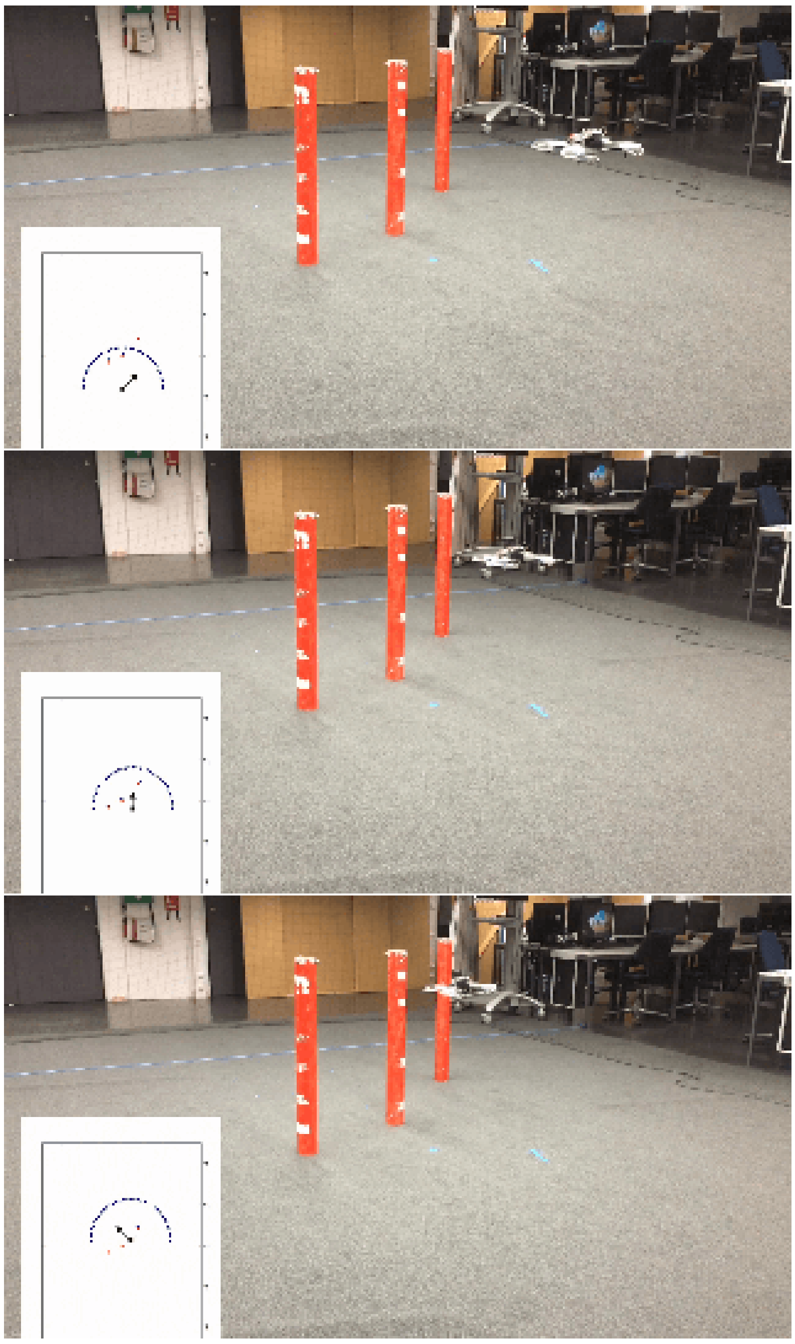

Figure 5.7: *Real world test on three cylinder obstacles*

5.1.5 Problems in real world test

Unstable distance readings and actions

When transferring from simulation to real world, there are two main problems on distance readings. Take the distance reading no.18 in figure below for example. First, distance readings have deviation ranging from $0.83m$ to $0.99m$ in this case. Second, same laser beam sometimes detects the obstacle, but sometimes not. Within 19 time steps for laser reading no.18, it successfully detect the obstacle 14 times. (In this case, the longest distance range is set to $2m$. If the reading is $2m$, it means laser doesn't detect the obstacle.)

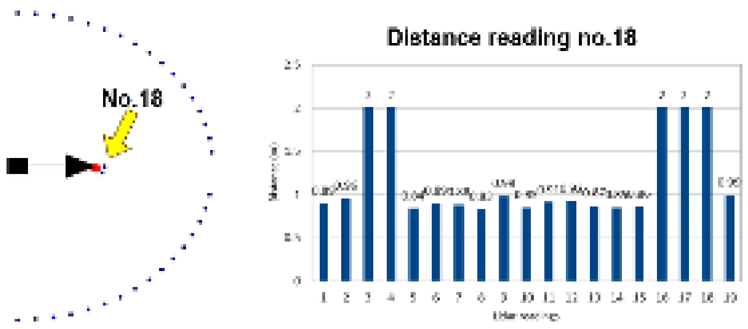

Figure 5.8: *Unstable distance readings*

Unstable distance readings leads to unstable action. As the figure below, in simulation, a model always choose the same action to go rightward when quadrotor and obstacle stay at same position. But in real word, the model might choose actions to go right-upward, go rightward or go right-downward when quadrotor stays at same position. (Small black square represents quadrotor, and black arrow represents direction of the chosen action.)

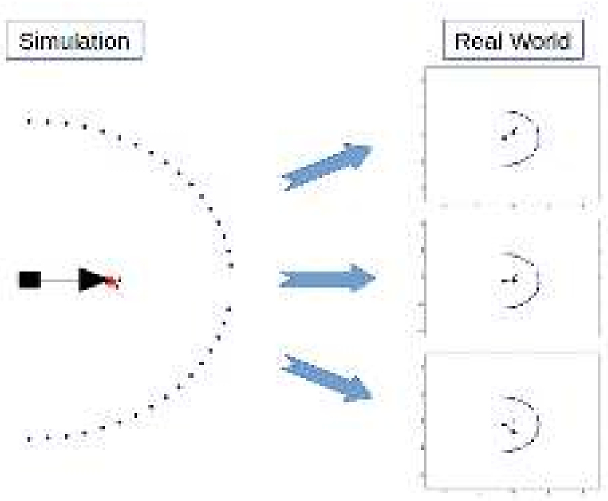

Figure 5.9: *Unstable actions in real world*

According to design and working principle of LIDAR sensor, there are two possible root causes for unstable laser readings.

Adaptive sampling rate

LIDAR sensor uses adaptive sampling methods, which means there is no fixed interval between distance readings, and the number of readings per revolution is different. In this experiment setup, LIDAR only guarantees to return roughly 70 distance readings per revolution, but not exactly 70. Therefore, if one reading is just on the edge of a obstacle, in some cases it can detect the obstacle, but not always.

Ghosting effect

When measuring the distance, LIDAR sensor first measures a large number of sub-samples. The final distance readings are the average of some amounts of sub-samples. In normal case, if all sub-samples are reflected from the same surface, final reading will be the distance of that surface. When ghosting effect happens, it means some sub-samples are in one surface, but some are in another surface. The final readings will be distance between these two surfaces, but actually there is no object at that distance.

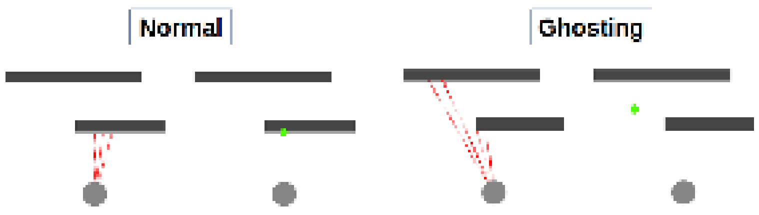

Figure 5.10: *Ghosting effect*[1]

Missing distance readings

Another even worse problem is that sometimes LIDAR sensor might display a bunch of non-sense distance values or return nothing as the figure below. This causes navigation model to choose a totally meaningless action which leads to collision. Fortunately, this problem only happened seldom, and quadrotor is able to get back to right track quickly when LIDAR returns the correct distance readings.

[1]`http://scanse.io/`

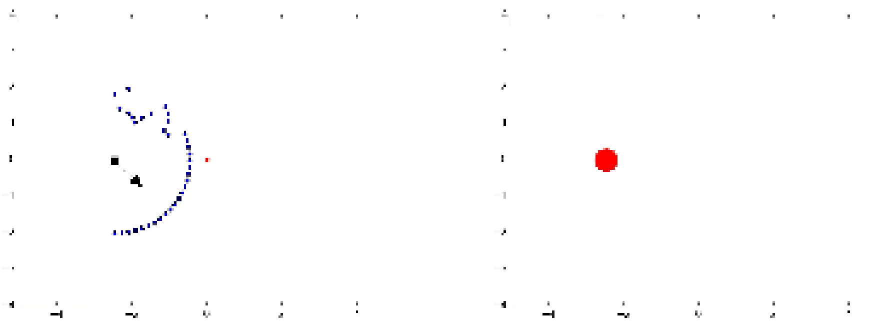

Figure 5.11: *Error and missing distance readings*

The possible root cause is mechanics design issue inside LIDAR. Because there isn't any protection for the turning part of LIDAR, the slip ring might have wear problem easily. When slip ring is mismatched or short-circuited, distance signal could not be transmitted normally, which leads to distance readings error.

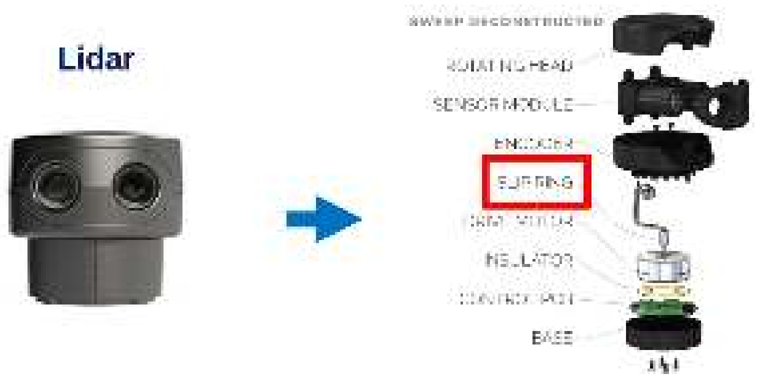

Figure 5.12: *LIDAR slip ring issue*[2]

Discussion

From real world experiment results, unstable distance readings didn't cause any collision and navigation model was still able to generate proper trajectories. Although the action in some time steps might be incorrect and guide quadrotor get closer to obstacles, the correct action in next time step is able to quickly bring quadrotor back to a appropriate trajectory and move away from obstacles. In case the action size is small enough ($0.2m$ in our model).

A straightforward solution is replacing the current LIDAR by more accurate one. The alternative solution on training model is to reduce the action step size to lower the collision risk for each incorrect action.

[2]`http://scanse.io/`

5.2 2D complex navigation with randomized start and goal

5.2.1 From 1D easy forward navigation to 2D complex navigation

After the success in easy forward navigation, we further tested the model in more complicated navigation condition: start from a random position and reach a random position. We made following three changes in training environment.

Change input and action size

To navigate from point A to point B in a 2D environment, the quadrotor has to move in 360° directions. Therefore, we changed the range finder into 360° field of view with 70 distance readings and 8 actions.

Change environment bounded box from rectangle to square

Switching from 1D to 2D navigation, We also changed the bounded box from rectangle to square because quadrotor has to move in both X and Y directions.

Add two extra inputs to obtain goal information

To acquire the information of goal location, we added the relative distance and angle between quadrotor and goal as two extra inputs.

Oscillation problem in complex navigation

When we used same A3C_FC model to navigate in randomized start and goal position, quadrotor always stuck between obstacles. Although it did not hit the obstacle, the quadrotor just oscillated back and forth till the end of episode. There are two reasons caused oscillation problem.

First, with 2m laser range, most of laser beams do not detect anything. In other words, most of input data are useless. Also, the distance readings might be similar when quadrotor stay in different locations. Therefore, short laser range confuse the model and can not provide enough environment information to it.

Second, using range finder in an open space, distance readings only provide the relative position between quadrotor and obstacles. In contrast, with an environment closed by physical walls and long range lasers, the distance readings provide relative positions between quadrotor, obstacles and walls. Actually, this also provides the absolute position of obstacles, because walls and obstacles are fixed in the map. The model has ability to learn the pattern of the environment, or in other words, learn the absolute position of obstacles. This make the model more robust to provide the correct actions in each state.

From above two reasons, we made following two changes, the detailed comparison and explanation is in Section 6.

- 1. Build the physical walls to close the environment
- 2. Increase laser range from 2m to 10m

From open to closed environment

With a physical bounded box surrounding the environment, laser ranger finder is able to detect the relative position of obstacles and walls, and learn the pattern of entire environment. This gives model more environmental knowledge to train a more reliable control policy.

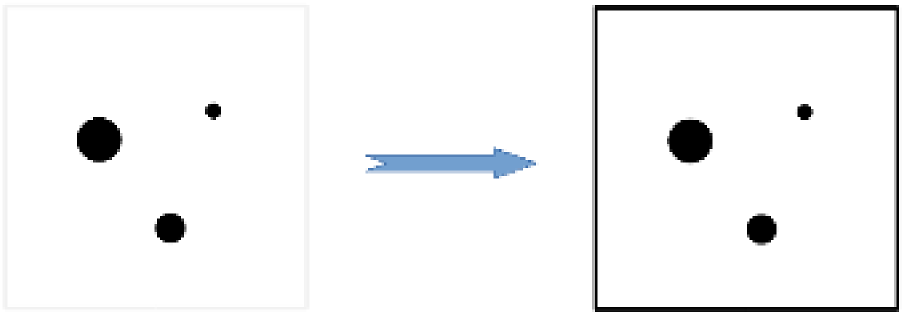

Figure 5.13: *From bounded space to open space*

The value function shows that the model can not learn anything in open environment. Without existing of the wall, the model only acquire the relative position of obstacle but can not learn the pattern of the map.

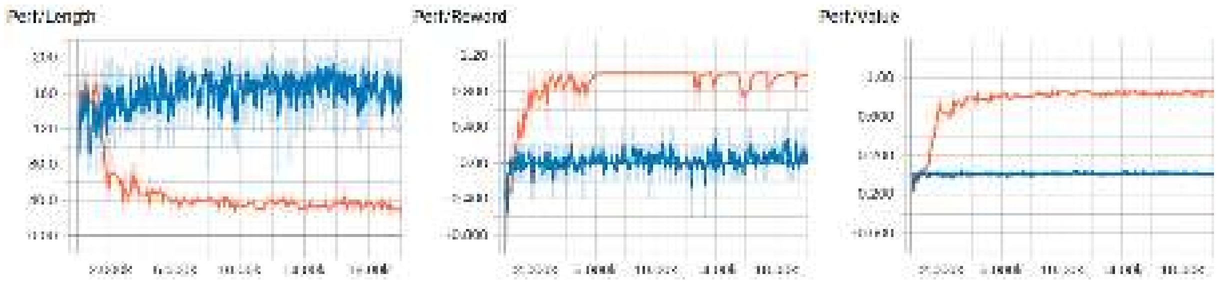

Figure 5.14: *Closed(orange) vs Open(blue) environment*

From short to long laser range

The benefit of longer laser range is same as the closed environment. When range finder detects more objects in the environment, it provides more knowledge to model which is favorable for training.

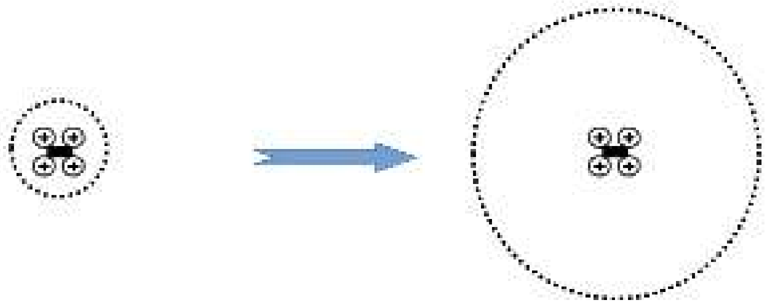
Figure 5.15: *From long laser range to short laser range*

Short range laser and open space are actually two similar situation for quadrotor. Most of the lasers detect nothing in the environment, which means most of the input data are useless. The whole map structure can not be learned in these two cases.

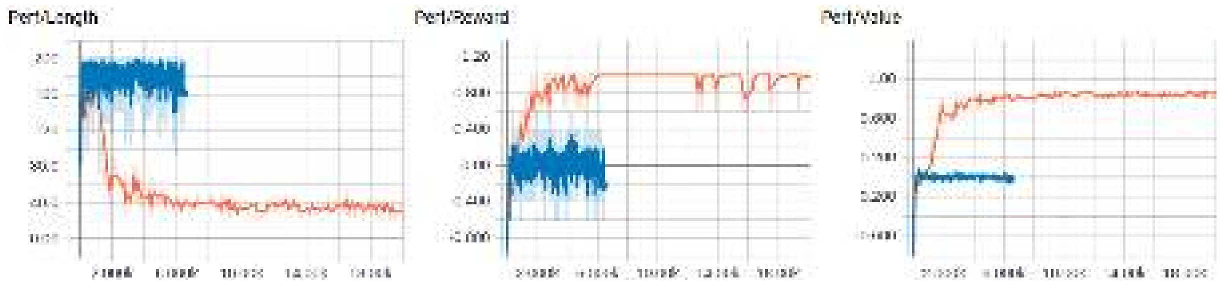

Figure 5.16: *Long(orange) vs Short(blue) Laser Range environment*

5.2.2 Simple Environment

After these changes on environment, we built three simple maps, which contains only one shape of obstacles with larger open space between obstacles and boundaries. The idea behind is that we can represent every environment as the combination of obstacles, and each complex obstacles can be seen as a combination of simple obstacles. Here we choose cylinder, rectangle, and wall to represent simple obstacles as the figure below.

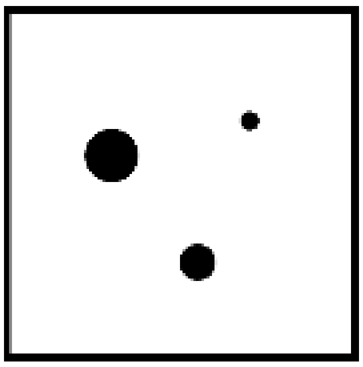
Figure 5.17: *Cylinder_map*

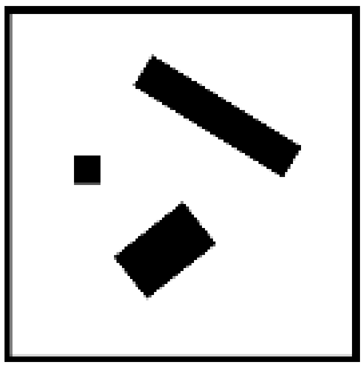
Figure 5.18: *Rectangle_map*

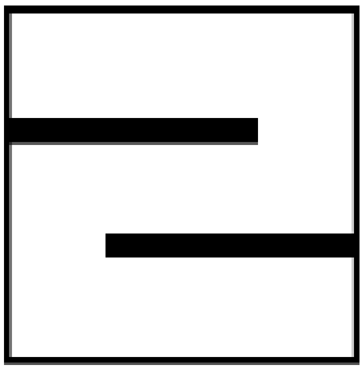
Figure 5.19: *Wall_map*

To test performance of these two navigation models, we randomized the start and goal positions at beginning of each episode and run it 200 times to record the navigation success rate. There are three outcomes for each navigation test: reach the goal, hit the obstacles or walls, oscillate without hitting anything till the maximum length of episode. To analyze the failure conditions, we calculated the ratio of collision cases in all failure cases as **C%** in the table. Comparing these two models, SL_LSTM model has worse performance with higher collision ratio. Comparing these three maps, Rectangle_map has lowest success rate in both models. Because Rectangle_map is the most compact one in three models with shortest gap between obstacles and walls.

Table 5.1: *Success rate in simple maps*

	Cylinder_map (C%)	Rectangle_map (C%)	Wall_map (C%)
A3C_LSTM	100% (-)	99% (0)	100% (-)
SL_LSTM	96% (100)	81% (87)	94% (56)

C%: Ratio of collision cases in all failure cases

5.2.3 Complex Environment

After testing on simple environment with only one shape of obstacles, we go further to test three complex environment: Office, Street and Forest. Which are maps combining different obstacles and smaller open space.

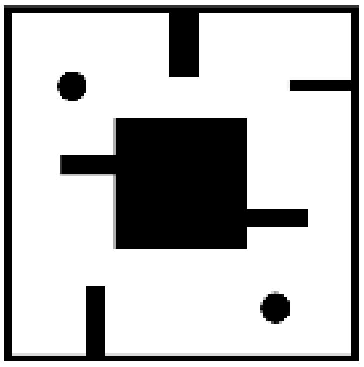
Figure 5.20: *Office_map*

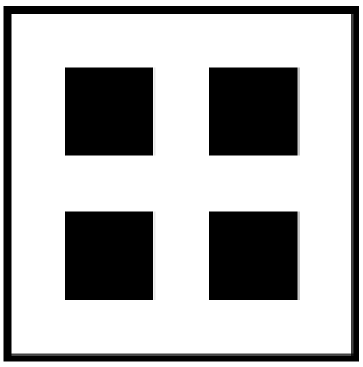
Figure 5.21: *Street_map*

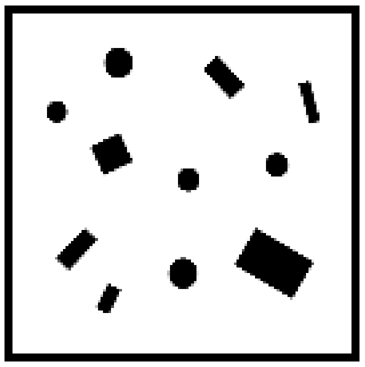
Figure 5.22: *Forest_map*

Results are similar with simple_maps. A3C_LSTM model performed better than SL_LSTM model with lower collision ratio. The Street_map has highest success rate in both models, because it is the least complicated map with only one type of obstacles and larger open space.

Table 5.2: *Success rate in complex maps*

	Office_map (C%)	Street_map (C%)	Forest_map (C%)
A3C_LSTM	98% (0)	100% (-)	99% (0)
SL_LSTM	95% (30)	100% (-)	87% (81)

C%: Ratio of collision cases in all failure cases

Using A3C_LSTM and SL_LSTM models, the navigation success rate are able to get over 90%. However, this happened only when training and navigating on the same map. Now we would like to see environment generalization ability for these two models.

5.3 Generalization ability of navigation model for different maps

The goal of this project is not training a navigation model only works in the same map, but we also want to see the generalization ability of navigation models. When using deep reinforcement learning model, it is shown that domain randomization [25] can successfully generalize the model to perform robot grasping task in unseen objects. We also applied the idea of domain randomization, but the difference is that we trained in a randomized simple environment, and transferred it to an unseen complex environment. Our idea is to transfer knowledge from simple tasks to complex tasks. For example, we know how to play basketball just by learning some simple knowledge, like running, dribbling, shooting and rules. We combine these knowledge in our mind then we know how to play basketball.

In this section, We would like to see how well the generalization ability are for A3C_LSTM and SL_LSTM models and how to improve it. We start from cross tests on simple and complex maps, and transfer from simple to complex maps. Finally we compared all transferring methods to find which has the best performance.

5.3.1 Simple maps cross test

First, we simply trained the model on one simple map and tested it on another simple map to see whether the learned navigation knowledge can directly transfer from one map to another.

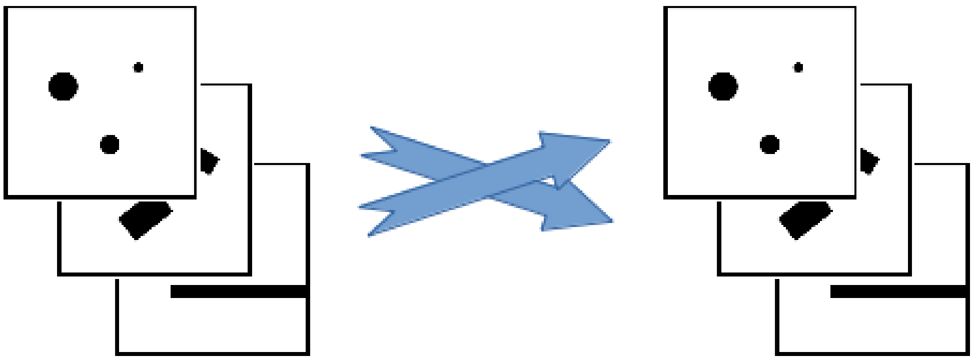

Figure 5.23: *Simple maps cross test*

For A3C_LSTM model, tssting on Cylinder_map got the highest success rate when transferring from other two maps, because it has the smallest obstacle size and larget open space.

Table 5.3: *Simple_map cross test (A3C_LSTM)*

	Cylinder_map (C%)	Rectangle_map (C%)	Wall_map (C%)
Train on Cylinder_map	–	65% (59)	61% (19)
Train on Rectangle_map	92% (41)	–	56% (10)
Train on Wall_map	89% (36)	49% (60)	–

C%: Ratio of collision cases in all failure cases

For SL_LSTM model, the success rate is similar to A3C_LSTM model but the collision ratio is higher in most of maps.

Table 5.4: *Simple_map cross test (SL_LSTM)*

	Cylinder_map (C%)	Rectangle_map (C%)	Wall_map (C%)
Train on Cylinder_map	–	51% (87)	45% (85)
Train on Rectangle_map	89% (57)	–	39% (54)
Train on Wall_map	86% (29)	53% (59)	–

C%: Ratio of collision cases in all failure cases

5.3.2 Complex maps cross test

Here we directly transferred the navigation knowledge from one complex map to another. A3C_LSTM model worked better when training on Street_map and Forest_map.

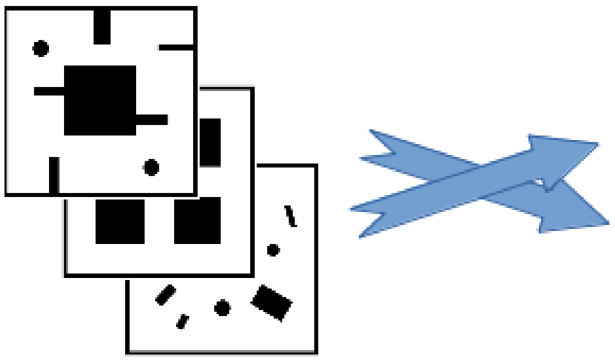

Figure 5.24: *Complex map cross test*

Table 5.5: *Complex_map cross test (A3C_LSTM)*

	Office_map (C%)	Street_map (C%)	Forest_map (C%)
Train on Office_map	–	65% (8)	73% (24)
Train on Street_map	75% (86)	–	82% (89)
Train on Forest_map	90% (0)	90% (15)	–

C%: Ratio of collision cases in all failure cases

Table 5.6: *Complex_map cross test (SL_LSTM)*

	Office_map (C%)	Street_map (C%)	Forest_map (C%)
Train on Office_map	–	87% (69)	82% (70)
Train on Street_map	64% (93)	–	71% (75)
Train on Forest_map	78% (70)	84% (94)	–

C%: Ratio of collision cases in all failure cases

5.3.3 From simple to complex

training on one simple map

First, we train the navigation model in only one simple map, and directly test the same model in three different complex maps as the figure below. All hyper-parameters are identical in these three trainings.

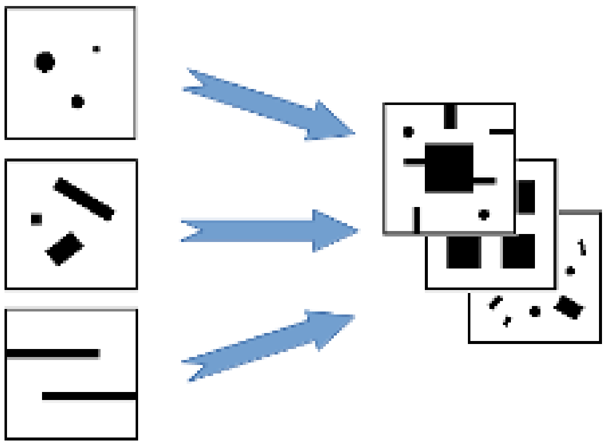

Figure 5.25: *Training on one simple map and testing on complex maps*

For A3C_LSTM model, training on Rectangle_map performed the best on transferring the navigation knowledge to complex maps.

Table 5.7: *Success rate for training on one simple map and testing on complex maps (A3C_LSTM)*

	Office_map (C%)	Street_map (C%)	Forest_map (C%)	Average
Cylinder_map	49% (0)	48% (15)	56% (19)	51%
Rectangle_map	**78%** (0)	**88%** (26)	**75%** (12)	**80%**
Wall_map	66% (0)	85% (10)	67% (10)	73%

C%: Ratio of collision cases in all failure cases

Table 5.8: *Success rate for training on one simple map and testing on complex maps (SL_LSTM)*

	Office_map (C%)	Street_map (C%)	Forest_map (C%)	Average
Cylinder_map	58% (67)	64% (100)	**84%** (94)	69%
Rectangle_map	63% (75)	60% (91)	73% (78)	65%
Wall_map	**81%** (56)	**77%** (62)	79% (67)	**79%**

C%: Ratio of collision cases in all failure cases

training on two simple maps

Here, both models performed better than training only on one simple map. Because in two simple maps, models learned different obstacles knowledge and more map pattern knowledge. It also showed that navigation models has the ability to gather the knowledge learned from different maps.

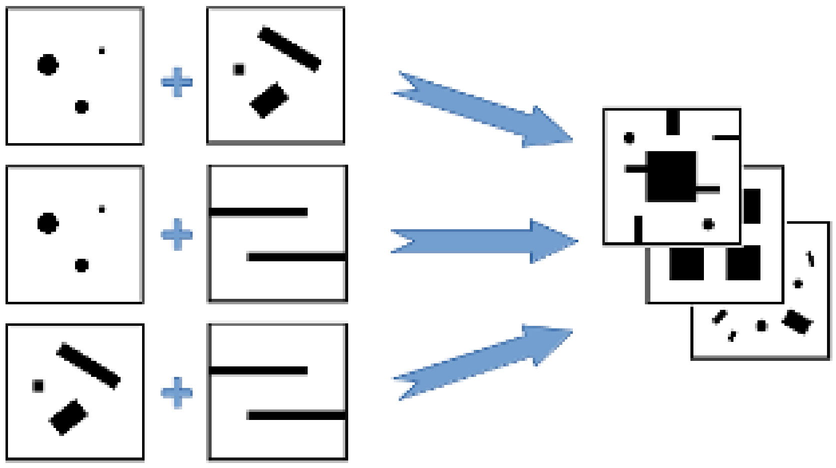

Figure 5.26: *Training on two simple maps and testing on complex maps*

Table 5.9: *Success rate for training on two simple maps and testing on complex maps (A3C_LSTM)*

	Office_map (C%)	Street_map (C%)	Forest_map (C%)	Average
Cylinder_map & Rectangle_map	**90%** (0)	85% (13)	**91%** (37)	**89%**
Cyliner_map & Wall_map	73% (0)	86% (4)	83% (3)	81%
Rectangle_map & Wall_map	76% (0)	**87%** (41)	80% (41)	81%

C%: Ratio of collision cases in all failure cases

Table 5.10: *Success rate for training on two simple maps and testing on complex maps (SL_LSTM)*

	Office_map (C%)	Street_map (C%)	Forest_map (C%)	Average
Cylinder_map & Rectangle_map	75% (76)	67% (95)	85% (93)	74%
Cyliner_map & Wall_map	69% (89)	**79%** (91)	86% (100)	73%
Rectangle_map & Wall_map	**80%** (92)	72% (88)	**86%** (86)	**75%**

C%: Ratio of collision cases in all failure cases

training on three simple maps

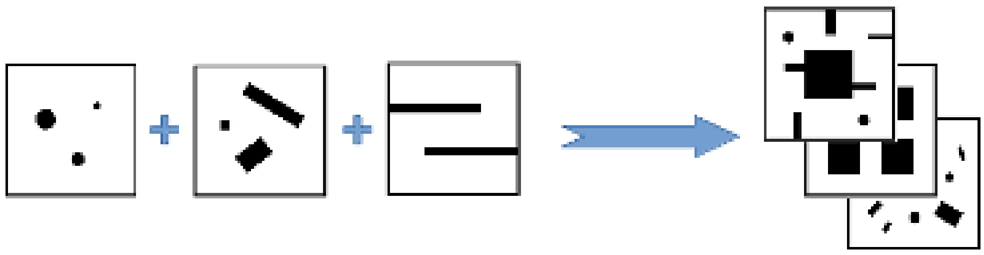

Figure 5.27: *Training on three simple maps and testing on complex maps*

Training on three maps, both models learned cylinder, rectangle and wall shape knowledge, and performed better than all previous training methods.

Table 5.11: *Success rate for training on three simple maps and testing on complex maps (A3C_LSTM)*

	Office_map (C%)	Street_map (C%)	Forest_map (C%)	Average
Combination of three simple maps	93% (6)	92% (13)	93% (87)	93%

C%: Ratio of collision cases in all failure cases

Table 5.12: *Success rate for training on three simple maps and testing on complex maps (SL_LSTM)*

	Office_map (C%)	Street_map (C%)	Forest_map (C%)	Average
Combination of three simple maps	87% (89)	86% (91)	89% (83)	87%

C%: Ratio of collision cases in all failure cases

5.3.4 Summary for all transferring methods

We got two important information from this summary. First, navigation knowledge can be transferred from one map to another, and training on more maps the model can learn more knowledge. Second, A3C_LSTM model performed better than SL_LSTM model is most training cases. In A3C_LSTM model, most of failure cases are collision, but it is opposite in SL_LSTM model.

Table 5.13: *Summary for all map transferring methods (A3C_LSTM)*

	Office_map (C%)	Street_map (C%)	Forest_map (C%)
One simple_map (best one)	78%(0)	88% (26)	75% (12)
One complex_map (best one)	90% (0)	90% (15)	82% (89)
Two simple_maps (best one)	90% (0)	87% (41)	91% (37)
Three simple_maps	**93%** (6)	**92%** (13)	**93%** (87)
Same map (Top line)	98% (0)	100% (-)	99% (0)

C%: Ratio of collision cases in all failure cases

Table 5.14: *Summary for all map transferring methods (SL_LSTM)*

	Office_map (C%)	Street_map (C%)	Forest_map (C%)
One simple_map (best one)	81% (56)	77% (62)	84% (94)
One complex_map (best one)	78% (70)	**87%** (69)	82% (70)
Two simple_maps (best one)	80% (92)	79% (91)	86% (100)
Three simple_maps	**87%** (89)	86% (91)	**89%** (83)
Same map (Top line)	95% (30)	100% (-)	87% (81)

C%: Ratio of collision cases in all failure cases

6

Comparison and Discussion

6.1 Simulation vs Real-World

In reinforcement learning, transferring the learned knowledge from simulation to real world could largely reduce the training time and cost, and it also reduces the safety risk while train a robot in real world. However, the model might perform different results even the simulation environment is very similar to real world. As mentioned in Section 5, unstable laser readings and actions are two main problems while transferring a navigation model from simulation to real world. The deviation of laser readings in real world depends on accuracy of LIDAR sensor, and how accurate we are able to estimate and control the quadrotor position. According to our experiment results, unstable distance readings didn't lead to any collision and quadrotor could always reach the destination. The reason is that even quadrotor get closer to the obstacle due to wrong action in a time step, it can still move away from obstacles by the correct action chosen in next time step. This happened only if the action step size is small enough ($0.2m$ in our experiments). Therefore, It it safe to say the navigation model can be successfully transferred from simulation to real world without any fine tuning. The main problems are in hardware and quadrotor control accuracy, but not in navigation model itself.

6.2 Reinforcement Learning vs Supervised Learning

There are two steps for training a supervised learning navigation model. First, generate numbers of navigation trajectories from Dijkstra path planner and move along these trajectories step by step to acquire correct actions at different states (laser distance readings, relative angle and distance to the goal). Second, use these data to train a navigation model which maps environment state to motion action. In this setting, navigation error might come from data generating part or from navigation model itself. However, training reinforcement learning only has one step, which is feeding environment states to train the model and get the desired motion actions. In this case navigation error only comes from model itself. We made an comparison to evaluate how these differences influence the navigation results (success rate and collision ratio) and training time.

6.2.1 Success rate

The figures below clearly shows three points. First, top red lines (the success rate of training and testing on same map) have highest success rate in all cases and we assume it is the highest possible success rate. Second, square scatter lines (A3C_LSTM model) are above triangle scatter lines (SL_LSTM model) in most cases, which means A3C_LSTM behaves better. Third, training on three simple maps outperforms training on one complex maps in all tests.

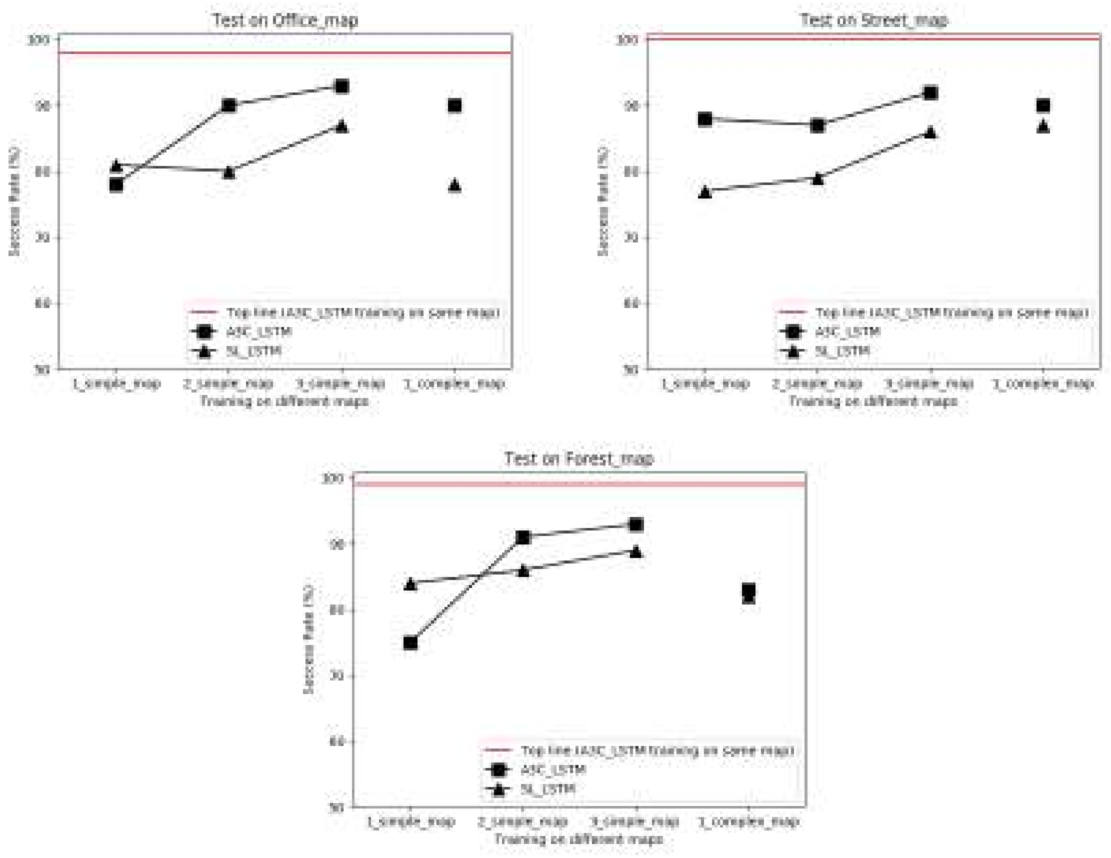

Figure 6.1: *Success rate comparison for A3C_LSTM and SL_LSTM models testd on three complex maps*

For navigation success rate, A3C_LSTM model (square marker) performs better than SL_LSTM model (triangle marker) in 13 of 15 training scenarios. The navigation success rate difference ranging from 22% to -9%, SL_LSTM model performs better only when train on one Simple_map and then test on Office_map and Forest_map. This result adhere to the model differences explanation, that supervised learning might contain addition error coming from data generation step. Which leads to lower navigation success rate in SL_LSTM model.

Table 6.1: *Success rate differences (A3C_LSTM over SL_LSTM)*

	Office_map	Street_map	Forest_map
One simple_map (best one)	**-2%**	11%	**-9%**
One complex_map (best one)	12%	3%	0%
Two simple_maps (best one)	10%	22%	5%
Three simple_maps	6%	6%	4%
Same map (Top line)	3%	0%	12%

6.2.2 Collision ratio (C%)

Two failure navigation cases for qaudrotor are collision with obstacles or oscillation at certain location. From the table below, it shows that SL_LSTM model has higher collision ratio in 13 out of 15 navigation scenarios. This result originates from the characteristics of training techniques in two models. The reward function in reinforcement learning model clearly separate the navigation outcomes into three categories:

- 1. Reach the goal ($Reward = 1.0$)
- 2. Hit the obstacle ($Reward = -1.0$)
- 3. None of above cases, which means oscillation ($Reward = 0$).

RL model understands the difference between collision and oscillation explicitly, and realizes collision is better than oscillation. Even reinforcement learning model fails to learn a control policy which is able to reach the goal, it will just oscillate at some locations rather than hit the obstacle. Nonetheless, supervised learning model is not able to recognize the difference between collision and oscillation. It simply maps a environment state into most correct action learned from the oracle. In its world, there are only two outcomes of navigation, either reach the goal or not reach the goal.

Table 6.2: *Collision rate (C%) differences (SL_LSTM over A3C_LSTM)*

	Office_map	Street_map	Forest_map
One simple_map (best one)	56%	36%	82%
One complex_map (best one)	70%	54%	**-19%**
Two simple_maps (best one)	92%	50%	63%
Three simple_maps	83%	78%	**-4%**
Same map (Top line)	30%	0%	81%

6.2.3 Training time

When train the navigation model in Cylinder_map using Intel Core i7-4790K CPU @ 4.00GHz, it takes $3h40m$ with $3k$ steps for A3C_LSTM model's reward value converging to 1.0. However, when train a SL_LSTM model under same setup, it takes only $53m$ with $40k$ steps to converge. Supervised learning model is converging four times faster than reinforcement learning model.

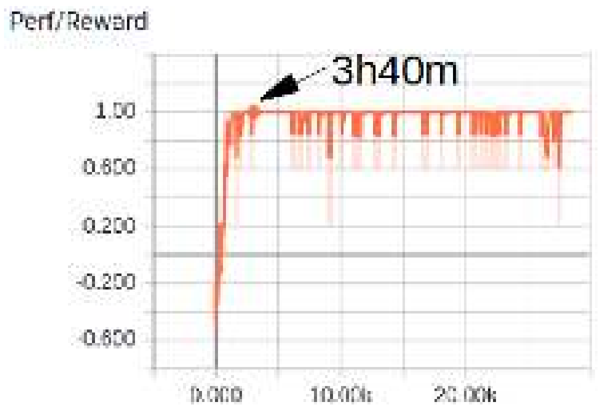

Figure 6.2: *A3C_LSTM*

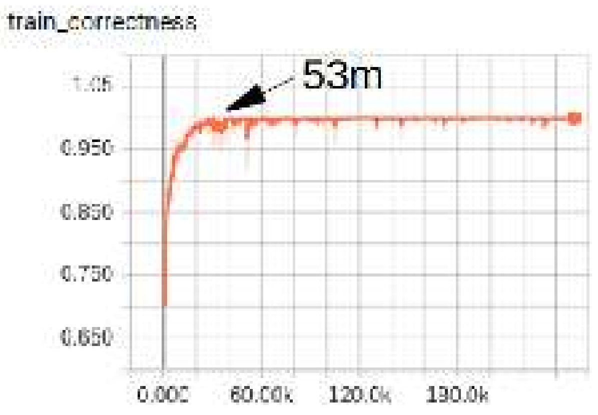

Figure 6.3: *SL_LSTM*

Figure 6.4: *Training time comparison*

6.3 Training on One Simple_map vs Multiple Simple_maps

As the figure below, when increase the number of training maps, the trend is getting higher success rate. This is true when test on all three complex maps, and also true on both A3C_LSTM (square marker) and SL_LSTM (triangle marker) models. The reason is that more training maps give the model more environment knowledge about different shapes of obstacles, so it is able to perform better in another unseen map. Also, the success rate of training on three simple maps are close to top red lines (the success rate of training and testing on same map). We assume it is the highest possible success rate using same navigation model. This result evidently demonstrates the environment generalization ability for training on diverse simple maps.

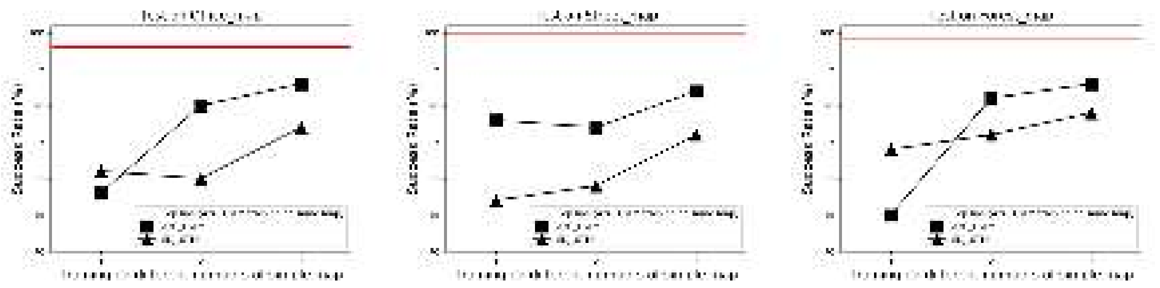
Figure 6.5: *The trend of increasing number of training maps*

6.4 Closed vs Open Environment

As mentioned in Section 5, surrounding walls provide map pattern knowledge to model, and also give more environment information to model. In a open environment, range finder can only provide the relative positions between quadrotor and obstacles, which is not enough to train the model and fail to perform a 2D navigation task.

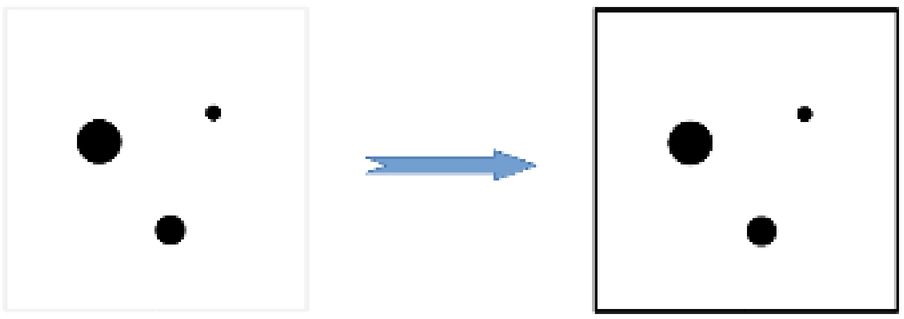
Figure 6.6: *From bounded space to open space*

Table 6.3: *Success rate for A3C_LSTM model in closed and open environment*

	Success rate	Training steps	Training time
Closed env.	99%	20k	13h 16m
Open env.	<10%	20k	19h 51m

6.5 Long vs Short Laser Range

As mentioned in Section 5, short laser range is another reason why navigation model fails to perform a 2D navigation. When reduce the laser range from $10m$ to $2m$, most of the distance readings become useless, because they cannot detect anything. That's why we increased the range of LIDAR in 2D complex navigation.

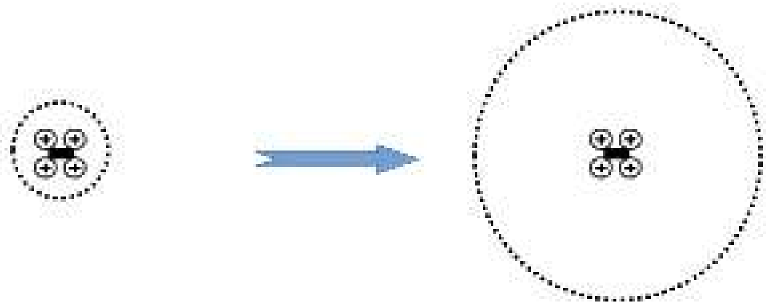

Figure 6.7: *From short laser range to long laser range*

Table 6.4: *Success rate for A3C_LSTM model in long and short laser range environments*

	Success rate	Training steps	Training time
Long Laser Range env.	99%	6k	6h 15m
Short Laser Range env.	<10%	6k	21h 39m

6.6 Grid vs Continuous Environment

When quadrotor fly in the simulation environment, it is only allowed to move between each intersection point of the grid. But now we would like to see whether the navigation model can also works in a continuous space, which is more similar to real world environment. Therefore, we changed the action space from 8 grid positions to 8 circular positions as the figure below. It means the quadrotor moves with same distance $(0.2m)$ in each action, and it can fly through any position in the space, not only on intersection of the grid anymore.

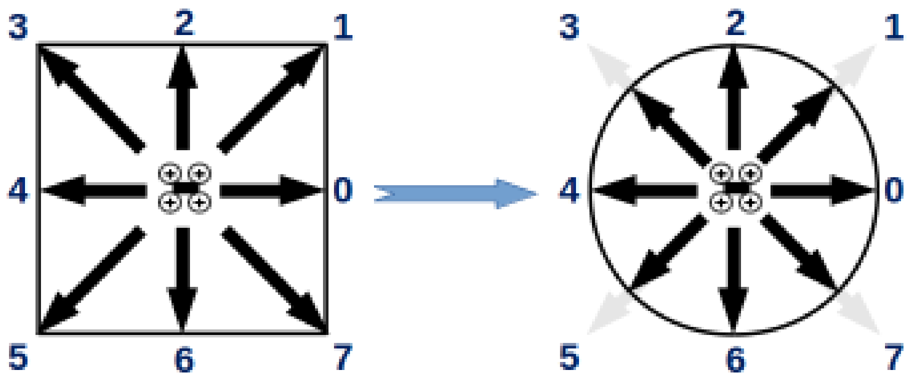

Figure 6.8: *From grid space to continuous space*

Here we compare following three results in grid and continuous simulation environment using A3C_LSTM model after training $20k$ steps.

- Perf/Length: Average time steps to finish one episode.
- Perf/Reward: Average rewards in each episode.
- Perf/Value: Average value function in each episode.

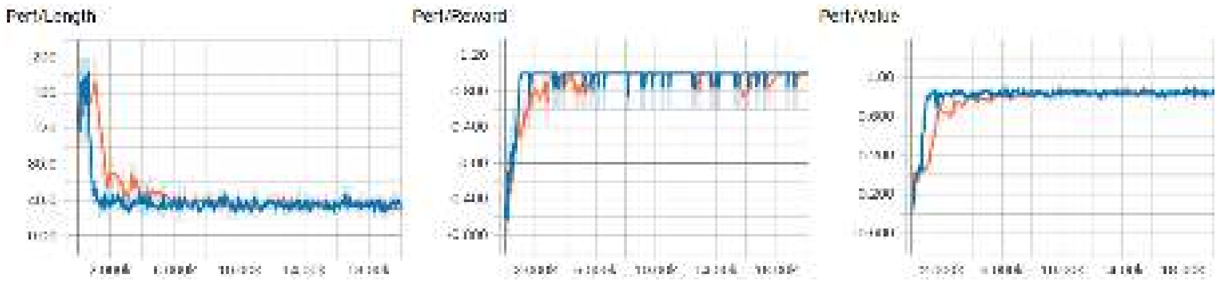

Figure 6.9: *Grid(orange) vs Continuous(blue) environment*

The success rate is same for gird environment and continuous environment, but it takes less time to train under continuous environment.

Table 6.5: *Success rate for A3C_LSTM model in grid and continuous environment*

	Success rate	Converging steps	Converging time
Grid env.	99%	4k	5h
Continuous env.	99%	1.4k	2h 30m

7

Conclusion

Fully connected deep reinforcement learning algorithm (A3C_FC) successfully performs an end-to-end mapless 1D easy forward navigation task. This model can be transferred from virtual to real world without any fine-tuning. In 2D complex navigation, physical bounded box and longer laser range help navigation model to acquire more environment information such as map pattern and obstacle arrangements. These extra environment knowledge highly improve the navigation success rate.

We also demonstrate the environment generalization ability for navigation models. Training merely on three simple maps with different obstacle features, A3C_LSTM model achieves navigation success rate up to 93% in unseen complex maps, while SL_LSTM model is able to achieve up to 89%. The A3C_LSTM model tends to oscillate when fails in navigation, but it is opposite for SL_LSTM model. The reason is reinforcement learning model has ability to explicitly distinguish that oscillation is better than collision, but supervised learning model not.

Reinforcement learning achieves higher navigation success rate with low collision ratio, while supervised learning required less training time. To further improve navigation model, we are investigating the possibility of training a reinforcement learning model on top of supervised learning to tackle the oscillation issue by combining the navigation knowledge from a motion planner oracle. It's also interesting to explore such simple to complex environment generalization method to other robotics tasks. If a model can manage the unseen complex tasks from some simple instructions, this would make training more efficient and model more generalized.

Bibliography

Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.

Arthur Juliani. Simple reinforcement learning with tensorflow part 8: Asynchronous actor-critic agents (a3c), 2016.

Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, 2006.

Slawomir Grzonka, Giorgio Grisetti, and Wolfram Burgard. A fully autonomous indoor quadrotor. *IEEE Transactions on Robotics*, 2012.

M.W. Mueller and R. D'Andrea. A model predictive controller for quadrocopter state interception. *ECC*, 2013.

Lei Tai, Giuseppe Paolo, and Ming Liu. Virtual-to-real deep reinforcement learning: Continuous control of mobile robots for mapless navigation. In *IEEE International Conference on Intelligent Robots and Systems*, 2017.

Mark Pfeiffer, Samarth Shukla, Matteo Turchetta, Cesar Cadena, Andreas Krause, Roland Siegwart, and Juan I. Nieto. Reinforced imitation: Sample efficient deep reinforcement learning for map-less navigation by leveraging prior demonstrations. *CoRR*, abs/1805.07095, 2018.

Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. End-to-end training of deep visuomotor policies. *CoRR*, abs/1504.00702, 2015.

Pieter Abbeel, Adam Coates, Morgan Quigley, and Andrew Y Ng. *An application of reinforcement learning to aerobatic helicopter flight*. PhD thesis, 2007.

Stéphane Ross, Geoffrey J. Gordon, and J. Andrew Bagnell. No-regret reductions for imitation learning and structured prediction. *CoRR*, abs/1011.0686, 2010.

Joshua Tobin, Wojciech Zaremba, and Pieter Abbeel. Domain randomization and generative models for robotic grasping. *CoRR*, abs/1710.06425, 2017.

J. Hwangbo, I. Sa, R. Siegwart, and M. Hutter. Control of a quadrotor with reinforcement learning. *IEEE Robotics and Automation Letters*, 2(4):2096–2103, Oct 2017.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

Bram Bakker. Reinforcement learning with long short-term memory. In *In Advances in Neural Information Processing Systems 15 (NIPS-2002*, 2003.

Piotr Mirowski, Razvan Pascanu, Fabio Viola, Hubert Soyer, Andrew J. Ballard, Andrea Banino, Misha Denil, Ross Goroshin, Laurent Sifre, Koray Kavukcuoglu, Dharshan Kumaran, and Raia Hadsell. Learning to navigate in complex environments. *CoRR*, abs/1611.03673, 2016.

Slawomir Grzonka, Giorgio Grisetti, and Wolfram Burgard. A fully autonomous indoor quadrotor. *IEEE Transactions on Robotics*, 2012.

S. Stevšić, T. NÃd'geli, J. Alonso-Mora, and O. Hilliges. Sample efficient learning of path following and obstacle avoidance behavior for quadrotors. *IEEE Robotics and Automation Letters*, 3(4):3852–3859, Oct 2018.

Mark Pfeiffer, Michael Schaeuble, Juan Nieto, Roland Siegwart, and Cesar Cadena. From Perception to Decision: A Data-driven Approach to End-to-end Motion Planning for Autonomous Ground Robots. Technical report.

Fereshteh Sadeghi and Sergey Levine. RL: Real Single-Image Flight Without a Single Real Image Training entirely in simulation Test in real world. In *RSS*, 2017.

Yuke Zhu, Roozbeh Mottaghi, Eric Kolve, Joseph J. Lim, Abhinav Gupta, Li Fei-Fei, and Ali Farhadi. Target-driven visual navigation in indoor scenes using deep reinforcement learning. In *Proceedings - IEEE International Conference on Robotics and Automation*, 2017.

V. Mnih, A. Puigdomènech Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. *ArXiv e-prints*, February 2016.

E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1959.

Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Mg. ROS: an open-source Robot Operating System. *IEEE Conference on Robotics and Automation (ICRA)*, 2009.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

J. Tobin, L. Biewald, R. Duan, M. Andrychowicz, A. Handa, V. Kumar, B. McGrew, J. Schneider, P. Welinder, W. Zaremba, and P. Abbeel. Domain Randomization and Generative Models for Robotic Grasping. *ArXiv e-prints*, October 2017.