CZECH TECHNICAL UNIVERSITY IN PRAGUE

Faculty of Electrical Engineering

**BACHELOR'S THESIS**

Zdeněk Rozsypálek

# Active 3D mapping using laser range finder with steerable measuring rays

**Department of Control Engineering**

Thesis supervisor: **Ing. Tomáš Petříček**

## Acknowledgements

I would like to thank my adviser for his advice.

## *Abstract*

The abstract in English.

## *Abstrakt*

Abstrakt cesky.

# Contents

# List of Figures

# 1 Introduction

Lidar sensors offer an accurate distance measurements, which can be used for mapping srounding space. There is much utilization of volumetric space reconstructions in different fields. For example, lidar sensors are nowadays essential equipment for a large variety of autonomous vehicles. The sensor can help autonomous vehicles to orient itself in an environment. One of the most significant issues which prevent broader implementation of these sensors is a relatively high price. Breakthrough in this field is solid-state lidar. These lidars do not have moving parts, and their price should be circa hundreds of dollars [1]. Solid-state lidar can send limited number of rays in chosen directions per timestamp. Zimmermann et al. [2] proposed mapping agent which creates dense reconstructions from sparse measurements. They also proposed prioritized greedy planning for choosing directions of these rays. The objective of this thesis is to apply reinforcement learning (RL) methods to learn planning of the rays and contribute to methods of controlling these sensors. RL is a field of study based on concepts of behavioral psychology, especially the trial and error method, and has in recent years experienced a rapid development due to the growth of computational power and neural networks improvement. Richard Sutton has made a helpful summary of RL concepts in his book [3]. One of the biggest achievements was playing Atari games by a RL agent without any prior knowledge of the environment [4]. Soon after was introduced an RL agent, able to solve simple continuous problems such as balancing inverse pendulum on a cart. Today state-of-the-art methods can solve complex problems with infinite action spaces. Although these methods reach great success, they still suffer from lack of sample efficiency - they need for training a lot of interactions with environment. This inefficiency makes creating agent controlling lidar very challenging, since training large neural networks is very time-consuming. The agent is divided into two parts - mapping and planning. The mapping part should create the best possible reconstruction from sparse measurements, while the planning part is focused on picking rays that will maximize reconstruction accuracy. Agents are trained using a publicly available dataset which contains drives of a car equipped with Velodyne lidar [5]. Theoretical background of RL is discussed in first part of this thesis. In the second part are methods from first part used to solve the Lidar-gym environment [6].

# 2 RL basics

Firstly, an environment where an agent can operate must be defined. The environment can be described as Markov decision process (MDP), where $S_t \in \mathcal{S}$ is a state from a set of possible states $\mathcal{S}$ in which environment is located in time $t$. In environments with observable MDP, an agent can observe the state of the environment and take action accordingly. An action is a probabilistic transition between states. Every action $A_t \in \mathcal{A}$ moves the environment from $S_t$ to $S_{t+1}$. The environment evaluates every action and returns appropriate reward $R_t$ (Figure 1). In RL set $\mathcal{A}$ is often called action space and set $\mathcal{S}$ observation space. Return $G_t$ is a sum of discounted future rewards [3].

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k} \tag{1}$$

where $\gamma \in [0, 1]$ is discount factor. RL methods define how experience from interacting with the environment will change the policy. The major issue is that maximizing immediate reward is often not an effective approach to maximize the sum of discounted rewards $G_t$. This greedy policy can take the agent into a very disadvantageous state. Thus, the agent must take into account future states and rewards. The goal of the agent is to find policy $\pi$ which maximizes expected return. The agent use the value function $V_\pi(S_t)$ which assesses how advantageous is being in state $S_t$ with policy $\pi$.

$$V_\pi(S_t) \doteq \mathbf{E}_\pi[G_t|S_t]. \tag{2}$$

Optimal policy $\pi^*$ is then defined as

$$\pi^*(S_t) \doteq \max_\pi V_\pi(S_t), \tag{3}$$

for all $S_t \in \mathcal{S}$. In the past, agents used big tables to estimate the value function. That is possible in environments with small action and observation spaces but is very memory consuming for larger environments and even impossible for continuous action or observation space. Therefore, modern methods use neural networks as function estimators.

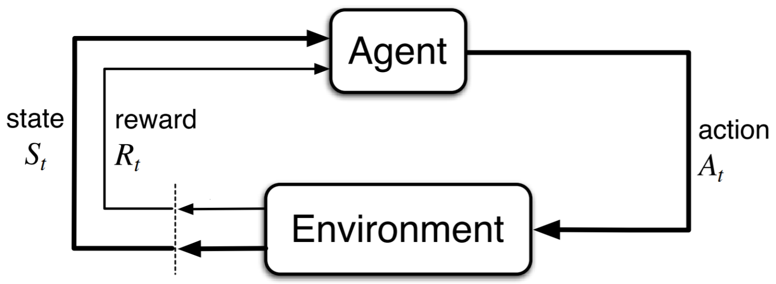

Figure 1: RL concept. Source - [3].

## 2.1 Temporal difference learning

Temporal difference (TD) learning combines the ideas of Monte Carlo methods and dynamic programming. It can learn directly from the experience obtained by interactions with the environment without any prior knowledge of the said environment. TD learning is done by following assignment in each timestamp [3]

$$\delta_t = R_t + \gamma V(S_{t+1}) - V(S_t) V(S_t) \leftarrow V(S_t) + \lambda \delta_t \tag{4}$$

where $\delta_t$ i the TD and $\lambda \in \mathbb{R}^+$ is step size.

## 2.2 Q-learning

Q-learning is a type of TD learning developed by Watkins [7]. The state value $V$ from the previous subsection is replaced by $Q$ value, which refers to a quality of action in a particular state instead of the quality of the state itself. When we rewrite TD learning (4) to Q-learning we get:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \lambda[R_t + \gamma \max_{A_{t+1}} Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]. \tag{5}$$

Our policy here is to take action with maximal $Q$ value. That is called greedy policy. An obvious drawback of greedy policy is that it does not allow to explore the whole environment properly because an action with the highest $Q$ value is always chosen. A solution to this problem is to make sometimes a random action, and explore the environment. This policy is often referred to as $\epsilon$-greedy policy.

**Algorithm 1** $\epsilon$-greedy policy in pseudocode

```
1: function ChooseAction
2:     $\epsilon \leftarrow \epsilon \cdot \epsilon_d$
3:     if $\epsilon >$ random $\in (0,1)$ then
4:         action $\leftarrow$ random $\in \mathcal{A}$
5:     else
6:         action $\leftarrow \max_{A_t} Q(S_t, A_t)$
7:     return action
8: end function
```

It is common to set $\epsilon = 1$ at the beginning of the training and decay rate $\epsilon_d$ close to one. The general idea behind this policy assumes that it is needed to explore the environment first and then exploit agent's experience.

## 2.3 Prioritized experience replay

Prioritized experience replay is biologically inspired mechanism introduced by Schaul et al. [8] which stores all experience (specifically: $S_t$, $A_t$, $R_t$, $S_{t+1}$) in a buffer and assigns priority to every experience. The main idea is that experience with high TD should have higher priority. It is thus necessary to calculate priority $p$ from TD error:

$$p = (|\delta_t| + \eta)^\rho \tag{6}$$

where $\rho$ indicates how much we prefer experience with higher priority and $\eta \ll 1$ is a constant which helps to avoid priorities very close to zero. Considering a greedy selection would abandon experience with low priority, a better approach is to choose experience $i \in \mathcal{I}$ with probability:

$$P(i) = \frac{p_i}{\sum\limits_{j \in \mathcal{I}} p_j}, \tag{7}$$

where $\mathcal{I}$ is set of all experience in the buffer. It is possible now to sample a batch of experience for training using this probability. It removes correlation in the observation sequence and improves sample efficiency of DQN. It is feasible to store all experience in a buffer sorted by priority, but a more efficient implementation is a sum tree. That is a binary tree, where the value of each root is equal to the sum of its children values (see Figure 2). Example of usage is in algorithm 2.

**Algorithm 2** Retrieve node from sum tree in pseudocode

```
1: function GetChild(parent, value)
2:     if parent.left is None then return parent
3:     if value ≤ parent.left.value then
4:         return GetChild(parent.left, value)
5:     else
6:         return GetChild(parent.right, value - parent.left.value)
7: end function
```

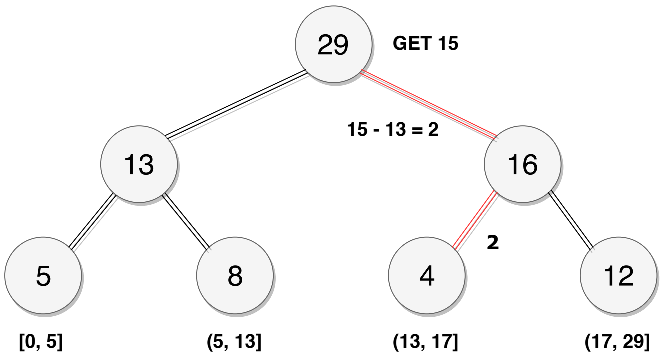

Figure 2: Simple example of sum tree.

# 3 Deep neural networks in RL

As was stated in the previous chapter, tabular methods are very inefficient in large environments. In these cases, it is possible to use deep neural networks which can replace tables. Deep Q networks (DQN) proposed by Googles Deepmind [4] outperformed all previous RL algorithms in playing Atari games. With neural networks also grew the popularity of policy gradient methods where function estimator outputs an action instead of Q values. Note that most of these methods are general and not necessarily tied to neural networks.

## 3.1 Deep Q network

The neural network takes current state as input and outputs Q value for each possible action. The network is trained using gradients of Q-value in the current state with respect to trainable weights $\theta$ of our neural network.

$$\delta_t = R_t + \gamma \underset{A_{t+1}}{\max} Q^\theta(S_{t+1}, A_{t+1}) - Q^\theta(S_t, A_t) \tag{8}$$

$$\theta_{t+1} = \theta_t + \lambda \delta_t \nabla_\theta Q^\theta(S_t, A_t). \tag{9}$$

Weights are updated in proportion to TD $\delta_t$. Unfortunately, this simple DQN agent suffers from a lack of sample efficiency and does not converge well. There are many techniques which can help DQNs to achieve satisfying results.

## 3.2 Target network

Target network is a technique which improves the convergence of DQN learning [4]. It uses two neural nets instead of one. The first is trained online network on a batch of data and the second target network is used for predictions during training. After the completion of training on a batch of data, the target network is updated [9].

$$\theta^- = \tau\theta + (1-\tau)\theta^-, \tag{10}$$

where $\theta^-$ is set of trainable weights of the target network, $\theta$ indicates online network weights and $\tau \ll 1$ is constant. TD $\delta$ is now calculated using target network:

$$\delta_t = R_t + \gamma \underset{A_{t+1}}{max} Q^{\theta^-}(S_{t+1}, A_{t+1}) - Q^\theta(S_t, A_t). \tag{11}$$

Target network stabilizes training since predicting network does not change after each training step.

## 3.3 Double Q-learning

Classic Q-learning algorithm tends to overestimate actions under certain conditions. Hasselt et al. propose the idea of Double Q-learning which decompose the max operation into action selection and action evaluation [10]. TD is then computed by the following equation.

$$\delta = R_t + \gamma Q^{\theta^-}(S_{t+1}, \underset{A_{t+1}}{\text{argmax}} Q^{\theta}(S_{t+1}, A_{t+1})) - Q^{\theta}(S_t, A_t). \tag{12}$$

Double DQN outperforms DQN in terms of value accuracy and in terms of policy quality.

# 4 Policy gradient

By this section, the goal of the neural network was predicting values by which the policy was determined. In policy gradient methods neural networks approximate the policy itself.

$$J = \mathbf{E}_\pi[G_t|S_t, A_t, \theta_t] \tag{13}$$

$$\theta_{t+1} = \theta_t + \lambda\widehat{\nabla_\theta J} \tag{14}$$

where $J$ is performance measure with respect to our neural network parameters and $\widehat{\nabla J(\theta_t)}$ is stochastic estimate of the gradient of the performance measure. In other words, this method is basically doing stochastic gradient ascent of $J$ with respect to $\theta$ [11]. Policy gradient methods are outperforming DQNs, especially in continuous action spaces, because it does not have to estimate Q-value for every possible action.

## 4.1 Actor-Critic

Thanks to predicting action directly, it is much easier to predict in continuous action space, but the Q-value which assessed the quality of action in a certain state has been lost. That is why the Actor-Critic framework was created. It uses two separate neural networks - actor which predicts action and critic which assesses action advantage. Concept is visualised in the Figure 3.

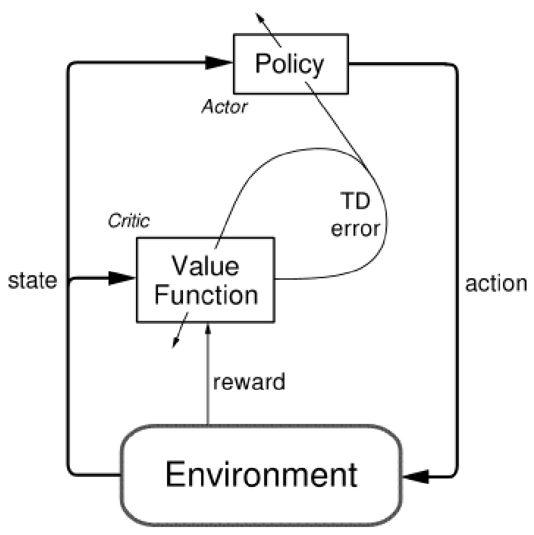

Figure 3: Actor-Critic framework. Source - [3].

Consider critic using Q-values for the update. $\theta$ and $\omega$ denote trainable weights of actor and critic, respectively. Critic update is similar to DQN:

$$\delta_t = r_t + \gamma Q^\omega(S_{t+1}, \mu^\theta(S_{t+1})) - Q^\omega(S_t, A_t) \tag{15}$$

$$\omega_{t+1} = \omega_t + \lambda \delta_t \nabla_\omega Q^\omega(S_t, A_t). \tag{16}$$

Note that instead of $A_{t+1}$ is now used function $\mu^\theta(S)$, which is an action estimate by actor neural network. Actor update rule is not so straightforward.

$$\theta_{t+1} = \theta_t + \lambda \nabla_\theta \mu^\theta(S_t) \nabla_a Q^\omega(S_t, A_t)|_{a=\mu^\theta(S_t)}. \tag{17}$$

This equation uses chain rule for derivatives to obtain the gradient of Q-values with respect to trainable weights $\theta$. Namely:

$$\frac{\partial Q^\omega(S_t, A_t)}{\partial \theta} = \frac{\partial Q^\omega(S_t, A_t)}{\partial A_t} \frac{\partial A_t}{\partial \theta}. \tag{18}$$

Actor neural network is updated by gradients which change action output to maximize Q-value of the critic [12]. There are other approaches, which doesn't use Q-value as critic assessment, but they rather use so-called advantage [13]. These methods are beyond the scope of this thesis.

## 4.2 Stochastic Actor-Critic

A stochastic Actor-Critic method is frequently used approach. In this method actor outputs parameters of distribution and action itself is sampled within the parameterized distribution. It is standard to use normal distribution and predict the mean and variance of action. The biggest advantage of the normal distribution is that it can be adjusted to use of backpropagation [14]. Another benefit of the stochastic actor is that it does not need any other techniques for action space exploration.

### 4.2.1 Beta distribution

On the other hand, obvious drawback of the normal distribution is that there is always some small probability of sampling an outlier. There is also an issue for bounded action space. When mean value of the normal distribution is close to the boundary, an agent can experience not negligible bias. A solution for both problems is to use beta distribution as a stochastic policy [15]. The beta distribution is defined by the following function:

$$f(x; \alpha, \beta) = \frac{\Gamma(\alpha + \beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{\alpha-1}(1-x)^{\beta-1}, \tag{19}$$

where $\alpha, \beta \in R_0^+$ are distribution parameters and $x \in [0, 1]$. $\Gamma$ is Euler's gamma function, which extends factorial into the set of real numbers. Beta distribution is shown in the Figure 4. The biggest advantage is that Beta distribution is bounded by definition and does not need additional clipping.

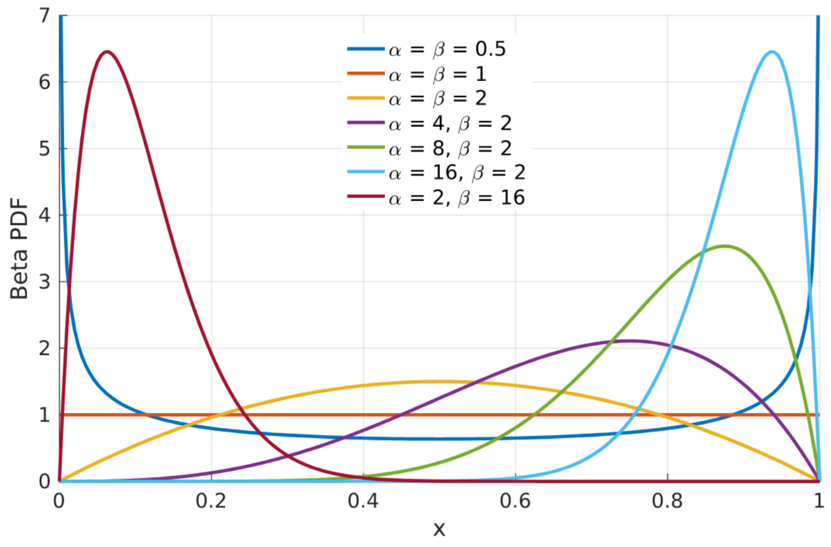

Figure 4: Probability density of beta distribution. Source - [15].

For reinforcement learning is suitable only $\alpha, \beta \geq 1$. That makes beta distribution concave and unimodal. This can be ensured by using softplus activation and adding one at the end of actor-network.

## 4.3 Deterministic policy gradients

Deep deterministic policy gradient (DDPG) is one of the methods for exploiting the Actor-Critic framework. Whereas stochastic actor predicts distribution parameters and samples an action, the DDPG outputs the action directly. Silver [12] has shown that deterministic policy can outperform its stochastic counterparts. A disadvantage of deterministic approach is that it needs an additional policy to ensure action space exploration. Exploration methods are discussed in subsection 4.4.

## 4.4 Parameter and action space noise

In the large action space is crucial to emphasize agents exploration. Bad exploration can cause that agent converges prematurely and ends up in a local optimum. DDPG commonly uses stochastic policy to modify actors actions slightly.

$$\hat{A}_t = \mu^\theta(S_t) + \mathcal{N}(0, \sigma^2) \tag{20}$$

where $\mathcal{N}$ is the normal distribution with mean value equal to zero and variance, which is reducing during the training and $\hat{A}_t$ is perturbed action. Action space noise helps agent to explore the environment.

Another approach is to apply noise directly to actor's weights. It can sometimes lead to more consistent exploration and richer behaviors [17].

$$\hat{\theta} = \theta + \mathcal{N}(0, \sigma^2) \tag{21}$$

where policy using $\hat{\theta}$ is a so-called perturbed actor, which is interacting with an environment. The major issue of parameter space noise is that it is much harder to tune. When we use action space noise, it is easy to estimate its impact on actions (differences between both approaches can be seen in the Figure 5). Because of an unpredictable influence of parameter space noise is necessary to use adaptive noise scaling.

$$d = |\hat{A}_t - \mu^\theta(S_t)|_2 \tag{22}$$

$$\sigma_{t+1} = \begin{cases} \kappa\sigma_t & \text{if } d \leq T \\ \frac{1}{\kappa}\sigma_t & \text{otherwise} \end{cases} \tag{23}$$

where $\kappa$ is scaling factor slightly bigger than one and $T$ is the threshold value, which has to be tuned to a specific environment.

When it is necessary to explore action space near to some specific action or include momentum of an environment, it is possible to use Ornstein-Uhlenbeck random process [9],

$$\hat{A}_t = \mu^\theta(S_t) + \nu(\rho - \mu^\theta(S_t)) + \phi\mathcal{N}(0, 1), \tag{24}$$

where $\nu, \phi \in [0, 1]$ are constants of the random process and $\rho$ is mean value around which we want to explore the action space. When $\nu = 0$ it is basic exploration as in expression (20).

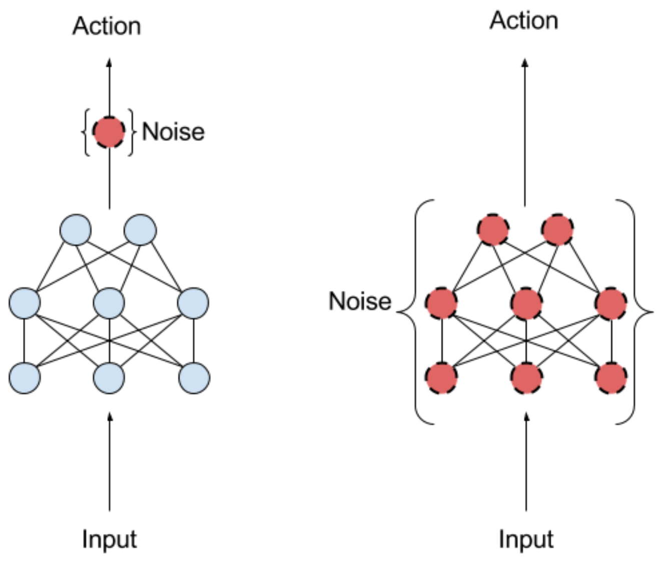

Figure 5: Left - action space noise, Right - parameter space noise. Source - [17].

# 5 Experiment

The experiment aims at using reinforcement learning algorithms for controling solid-state lidar with limited number of steerable rays. For purposes of the experiment it was neccessary to implement environment, where an agent can learn and be evaluated [6]. The lidar-gym environment is written in Python 3 based on OpenAI gym interface [18]. It uses point clouds from the KITTI dataset drives[5]. One episode of learning in the environment corresponds to one drive in the KITTI dataset. Large point clouds from drives are processed into 3D voxel maps by C++ package [19], which also provides ray tracing engine for the environment. Every voxel map is a 3D array containing real numbers which correspond to the occupancy confidence $c$ of each voxel.

$$\begin{aligned} c > 0 &\quad \text{occupied voxel} \\ c = 0 &\quad \text{unknown occupancy} \\ c < 0 &\quad \text{empty voxel.} \end{aligned} \tag{25}$$

The environment also offers visualization of actions using Mayavi [20] and ASCII art. Agents use neural networks as function estimators, which are implemented in Tensorflow [21] and Keras [22].

## 5.1 Environments

Lidar-gym implements several environments (visualized in Figure 6), which follow the same template with different sizes of voxel maps and action spaces. Observation space is a local cutout of voxel map, which provides occupancies from sensor's sparse measurements. The sensor is located in the quarter of x-axis and half of y-axis and z-axis of a local cutout. Action space is divided into two parts. First part is dense voxel map reconstructed from observations (sparse measurements). The second part of action space are directions of measuring rays. Each ray has own azimuth and elevation. Environment expects directions in the format of a 2D array of booleans, where true means fired ray. Reward function of the environment is negative logistic loss $-L$ (26). Parameters of environments are described in Table 1.

| Name of environment | Large | Small | Toy |
|---|---|---|---|
| Voxel map size [voxels] | 320 × 320 × 32 | 160 × 160 × 16 | 80 × 80 × 8 |
| Lidar FOV [°] | 120 × 90 | 120 × 90 | 120 × 90 |
| Densitiy of rays | 160 × 120 | 120 × 90 | 40 × 30 |
| Lidar range [m] | 42 | 42 | 42 |
| Number of rays | 200 | 50 | 15 |
| Voxel size [m] | 0.2 | 0.4 | 0.8 |
| Episode training time [min]* | 120 | 15 | 1.5 |

Table 1: Description of environments

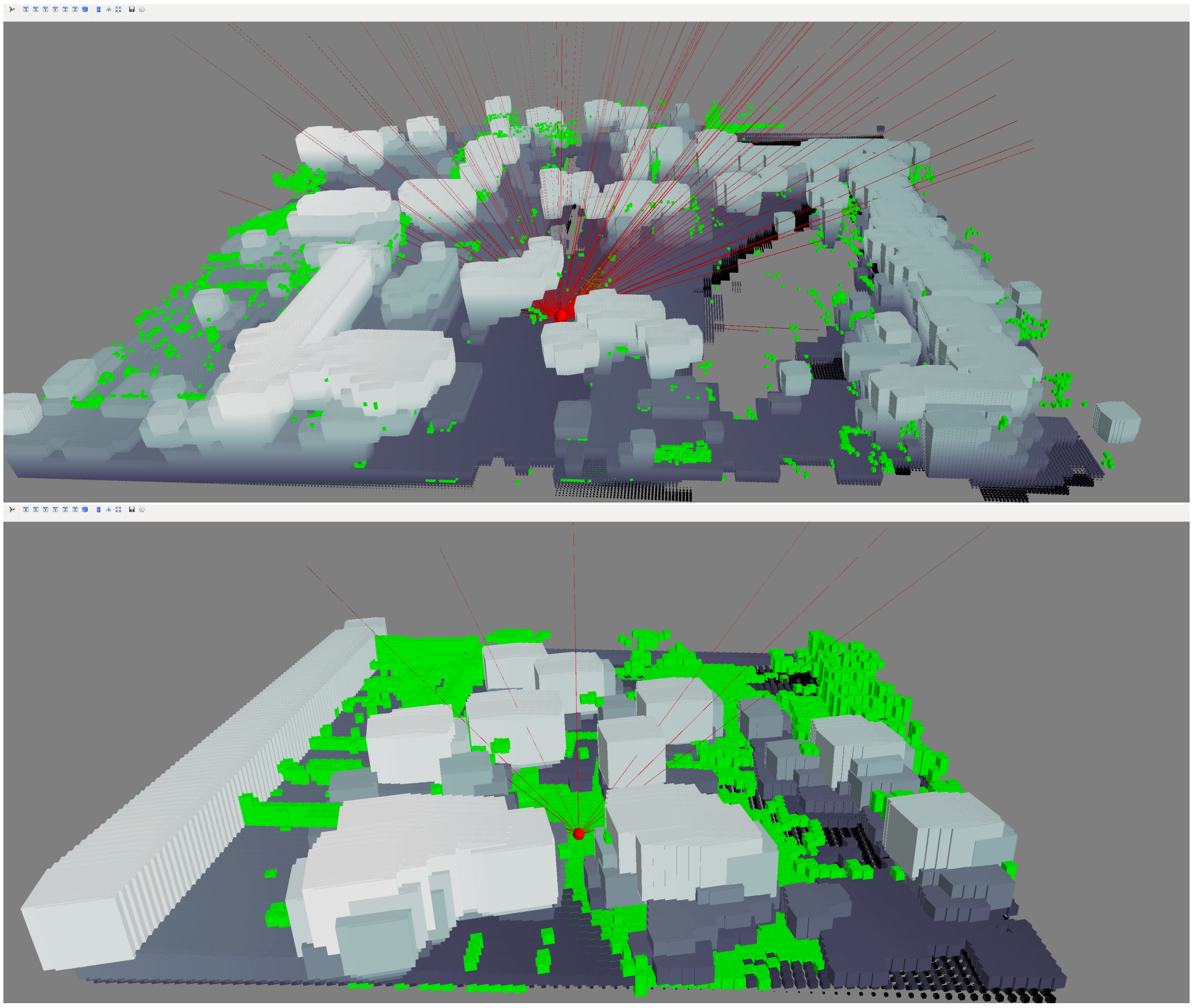

Figure 6: Visualization of two environments: Top - Large, bottom - Toy. Ground truth map is green, reconstructed map is in grey palette, sensor position and rays are red.

*Using GPU Nvidia 1080Ti.

Due to the high time complexity, all experiments were conducted in toy environment. The RL agents need significantly more training steps than supervised agents. Unlike the RL agents, for supervised agent is known desired output, thus it can be learned by a gradient descent on the loss function between made and desired output. There are RL agents trained for over million timestamps in OpenAI baselines [23]. One drive in the KITTI dataset has on average 200 epochs. All agents were trained and evaluated on different drives from the city category of the dataset.

## 5.2 Mapping agent

The mapping agent is based on work of Zimmermann et al. [2]. It uses a convolutional neural network (CNN) for reconstructing dense map from sparse measurements. 3D convolutional layers are used to learn the features and max-pooling layers to avoid overfitting. Whole CNN architecture is described in Figure 7.

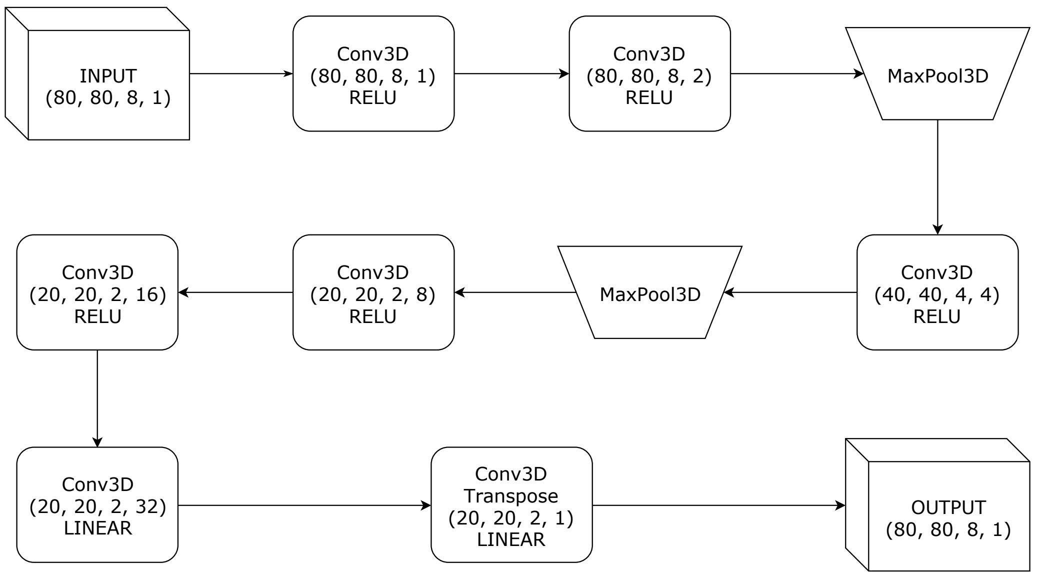

Figure 7: Input of the supervised mapping agent is voxel map containing sparse measurements. Output is dense reconstruction of the input.

Gradient descent is made by Adam optimizer [**?** ]. Optimizer uses logistic loss $L$ between ground truth map $Y$ and predicted dense map $\hat{Y}$.

$$L(Y, \hat{Y}) = \sum_i w_i \log(1 + \exp(-Y_i \hat{Y}_i)) \tag{26}$$

where $w$ are weights which balance importance of occupied and unoccupied voxels. Unfortunately, a naive implementation of this loss function is computationally inconvenient and often cause numerical issues as overflow. To stabilize training, following modified loss was used [24]

$$\begin{aligned} a_i &= -Y_i\hat{Y}_i \\ b_i &= \max(0, a_i) \\ L &= \sum_i w_i(b_i + \log(\exp(-b_i) + \exp(a_i - b_i))). \end{aligned} \tag{27}$$

At first, supervised mapping agent with random ray planning is trained. Reconstructions of the supervised agent are then used for training RL planning agents and after that is mapping agent retrained with RL agent picking the rays.

## 5.3 Discrete planning agent

Since the action of the environment $A_t$ for a direction of the rays is 2D binary array, first try is to use a discrete agent. DQN is the most used option for discrete action space, but in this use case, it requires some tweaks. Note that number of possible actions is extremely large. Even in toy environment it is $\binom{40\times30}{15} \approx 10^{34}$ of actions. Thus it is necessary to emphasize action space exploration. Further arises the problem with the $\epsilon$-greedy policy, because we are unable to process all possible actions and pick one with the biggest Q-value. We consider only one ray as action to resolve this issure. For $K$ rays is TD from (8) now computed as:

$$\begin{aligned} q(S_t, A_t) &= \overset{K}{\max_{A_t}} Q^\theta(S_t, A_t) \\ \delta_t &= R_t + \gamma\overline{q}(S_{t+1}, A_{t+1}) - \overline{q}(S_t, A_t) \end{aligned} \tag{28}$$

where $\overline{q}$ is average $Q$ value over $K$ actions with maximum $Q$ values. DQN agent implements all features as Prioritized experience replay, target network, and double Q learning which are described in the theoretical part of this thesis. Exploration is ensured by action space noise. Neural network architecture is described in Figure 8. The agent uses values of parameters shown in Table 2. The weights are updated via stochastic optimizer Adam with learning rate $\lambda$.

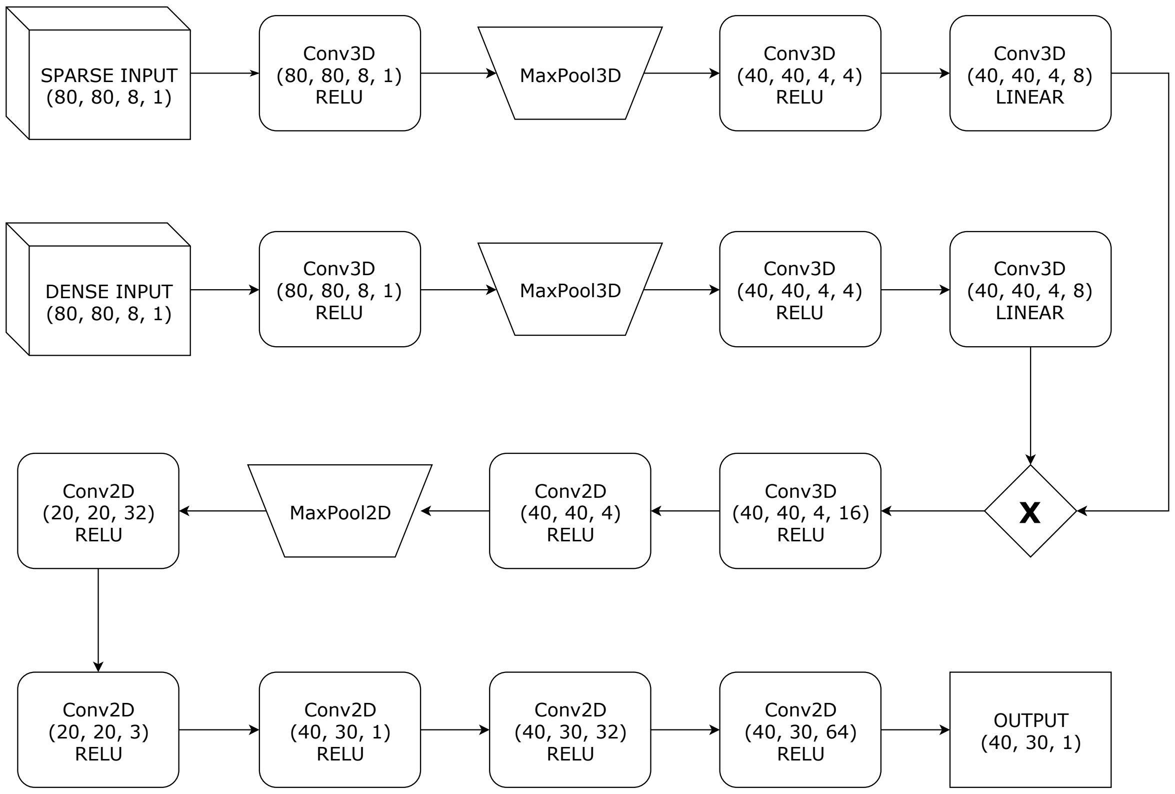
SPARSE INPUT (80, 80, 8, 1)
Conv3D (80, 80, 8, 1) RELU
MaxPool3D
Conv3D (40, 40, 4, 4) RELU
Conv3D (40, 40, 4, 8) LINEAR
DENSE INPUT (80, 80, 8, 1)
Conv3D (80, 80, 8, 1) RELU
MaxPool3D
Conv3D (40, 40, 4, 4) RELU
Conv3D (40, 40, 4, 8) LINEAR
X
Conv3D (40, 40, 4, 16) RELU
Conv2D (40, 40, 4) RELU
MaxPool2D
Conv2D (20, 20, 32) RELU
Conv2D (20, 20, 3) RELU
Conv2D (40, 30, 1) RELU
Conv2D (40, 30, 32) RELU
Conv2D (40, 30, 64) RELU
OUTPUT (40, 30, 1)

Figure 8: DQN architecture.

## 5.4 Continuous planning agent

The discrete action output was substituted by continuous action, which is then mapped into a 2D binary array. This will allow the agent to avoid extremely large discrete action space. Thank to this substitution it is possible to exploit actor-critic framework. The output the of actor is now 2 by $K$ array where the first row is the elevation and second is the azimuth of each ray. The last layer of the actor-network is tanh function, so its output is an element of $[-1, 1]$. As training method is used DDPG. The neural network architecture is described in the Figure 9.

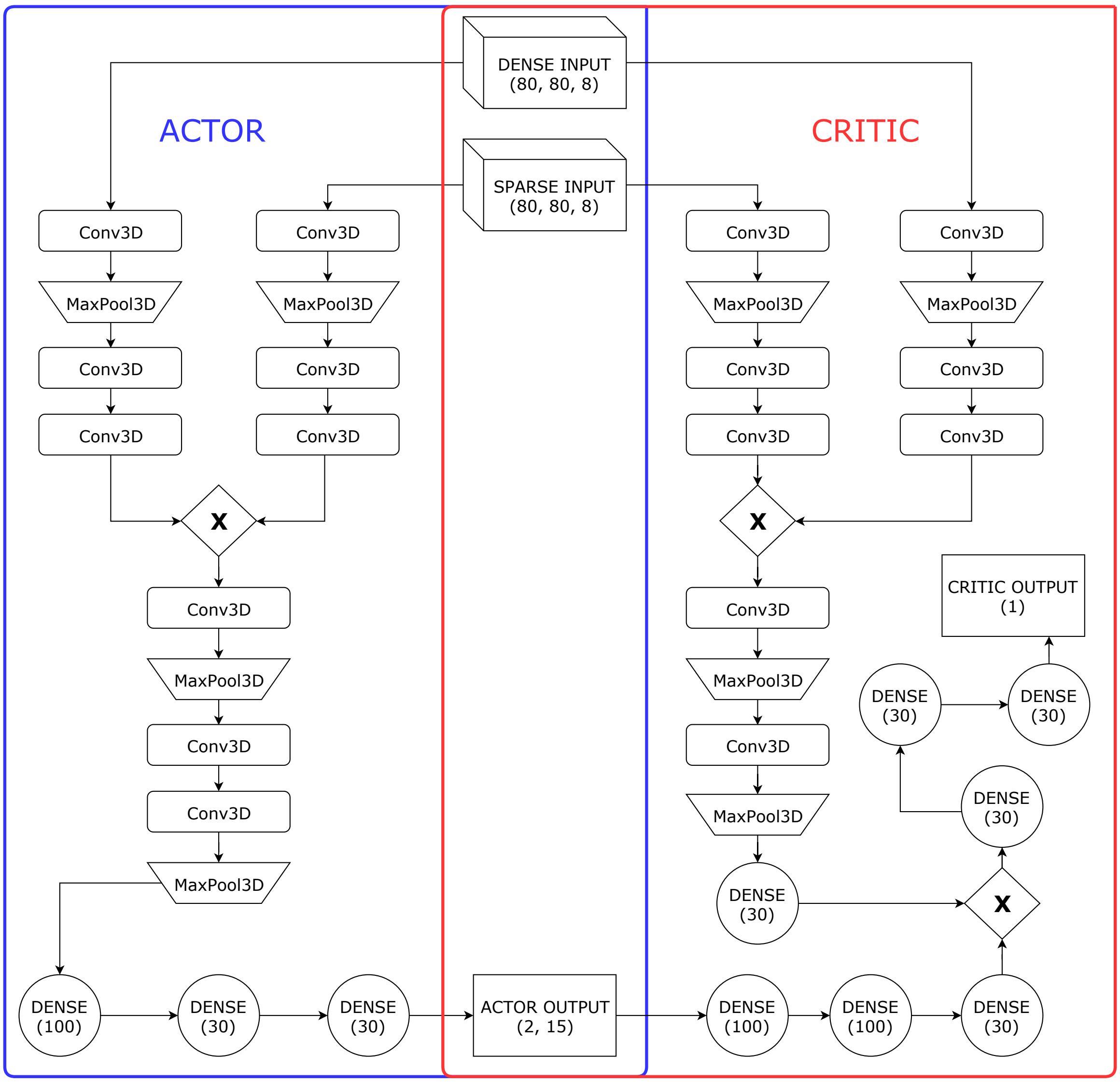

Figure 9: Architecture of the actor is on the left side and critic is on the right side. In the middle can be seen shared inputs and outputs.

To explore action space correctly in our experiments, it is necessary to apply Ornstein-Uhlenbeck random process. When only Gaussian noise is added, actions tend to converge into the corners very fast as in Figure 10. For actor's and critic's neural network is used Adam optimizer with learning rates $\lambda_a$, $\lambda_c$. The target networks are used to stabilize learning of actor and critic. The continuous agent also uses prioritized experience replay.

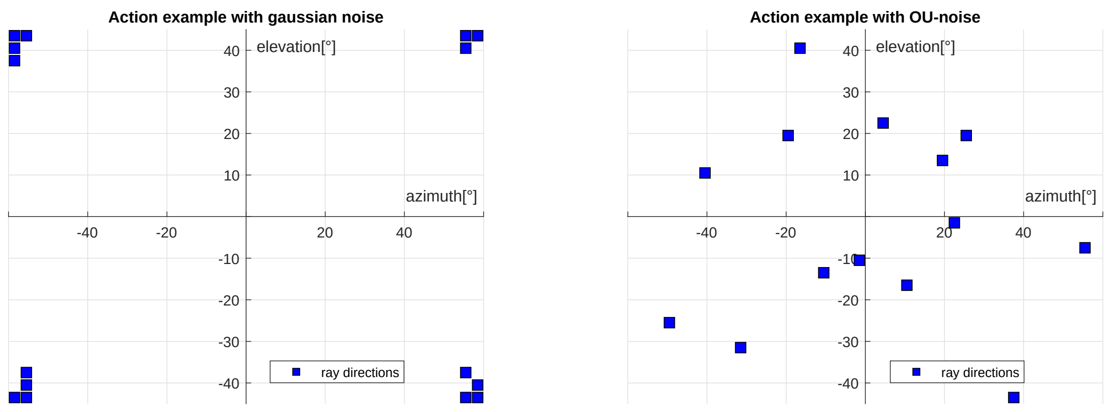

Figure 10: Difference between actions made by agents with different exploration methods. Left figure is an example of action made by an agent with Gaussian noise used for exploration. The action in the right figure is taken by agent trained using Ornstein-Uhlenbeck noise.

## 5.5 Stochastic planning agent

There is an obvious issue with the deterministic actor. For efficient exploring of ground truth map, it is required to hit as many unique voxels as possible. Thus making several similar actions in a row is not a good strategy. Unfortunately, except first few epochs of every drive, there is not a big difference between two subsequent observations. It is hard for a neural network to make two different outputs for two similar inputs. The solution to this problem could be a stochastic agent. The stochastic agent outputs parameters of beta distribution and preserves actor-critic framework. The architecture of stochastic agent is similar to DDPG from the previous subsection, the only difference is that the agent now outputs only two values - the distribution parameters. Action is then sampled with distribution probabilities. Stochastic actor output can be seen in Figure 11.

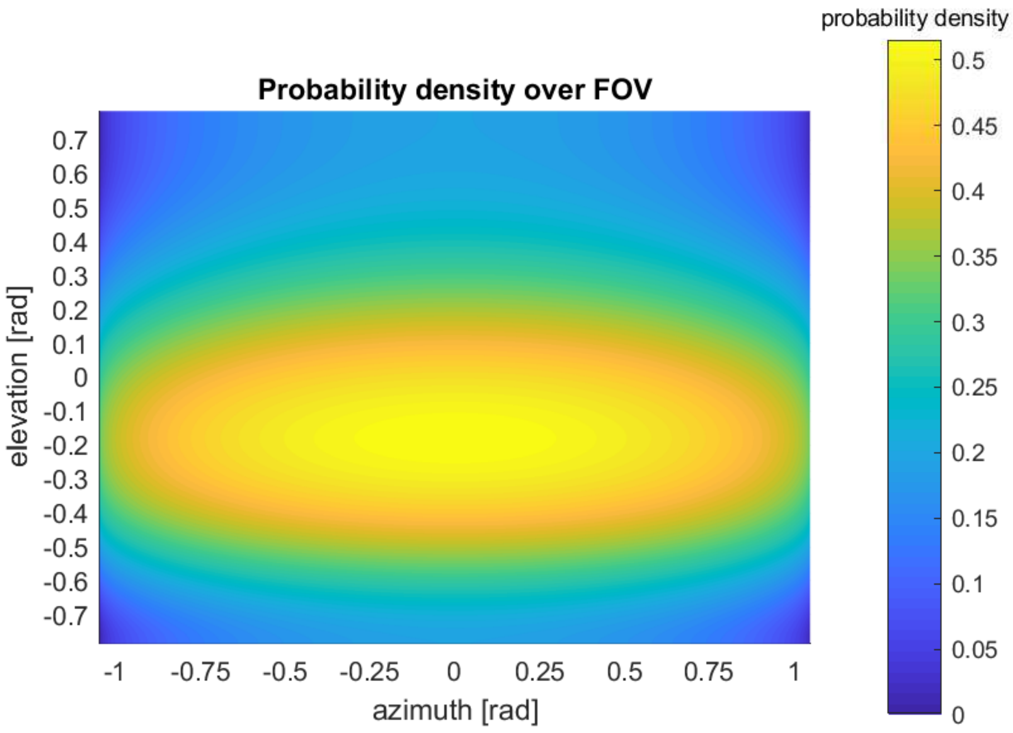

Figure 11: Probability density of picking ray, with specific azimuth and elevation.

## 5.6 Comparison of methods

Several methods were tried. Unfortunately, most of the methods do not perform well in the Lidar-gym environment. The DQL had the worst performance. That is probably caused by the extremly large action space. The workaround made in formula (28), did not help the agent to converge. Continuous DDPG agent also did not converge successfully. For DDPG agent is hard to change direction of rays efficiently in subsequent timestamps and it often makes several same actions in a row, that degrades performance a lot. Only the stochastic agent was able to outperform a random agent. Summary of agents performances is made in Figure 12. It turns out that it is very difficult to get satisfying results. Stochastic agent converges, but it does not use any kind of advanced strategy. Output of the DDPG actor does not even allow him to make any sophisticated action. Therefore were also conducted experiments with an extended stochastic agent, which used beta distribution for each ray separately, but with no significant success. Another advantage of simple stochastic agent is its scalability. This agent can be adjusted to the large environment much easier than any other used agent. Parameters used for these agents are described in Table 2.

| Parameter | Value |
|---|---|
| $\gamma$ | 0.09 |
| $\lambda_c$ | 0.001 |
| $\lambda_a$ | 0.0001 |
| $\tau$ | 0.01 |
| Memory size | 4096 |
| Batch size | 8 |

Table 2: Parameters used for learning. DQN uses $\lambda = \lambda_c$

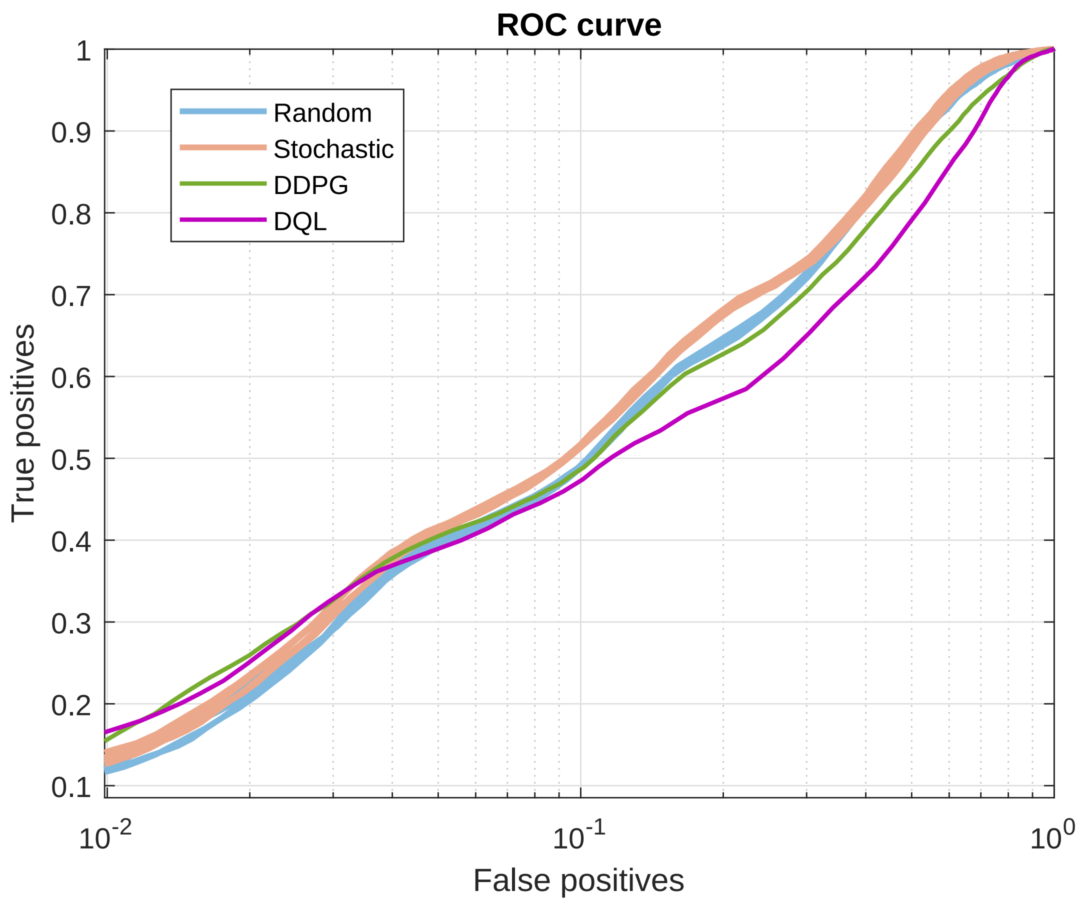

Figure 12: False positives were calculated using ground truth map and true positives using all voxels, which could be possibly hit by sensor. Five different evaluations are displayed for non-deterministic agents.

# 6 Conclusion

First, we overviewed reinforcement learning concepts and described several methods which help convergence of the learning process. Then, we addressed the challenging multi-dimensional control task of selecting depth-measuring rays for 3D mapping. Various agents and model architectures were implemented and compared. All deterministic agents performed poorly in this specific task. The stochastic agent successfully outperformed the random planner. Action space size and time-complexity were two major blockers during the training. None of the trained RL agents can compete with the prioritized greedy planner proposed by Zimmermann et al. [2].

## 6.1 Future work

Experiment which would be worth conducting is an agents, which stands between simple and extended stochastic agent. The extended stochastic agent has action space consisting of 60 real numbers (15 rays with azimuth and elevation and for each alpha, beta parameters). That is very likely too much for network architecture used by agent. On the other hand, when only one distribution is outputted for all rays, it does not allow agent to create advanced policies. Solution could be to output for example three different distributions, each describing five rays. Another improvement could be achieved by adjusting neural network architecture. Especially splitting and merging layers of neural network, can have significant impact on performance.

## References

[1] Evan Ackerman. Quanergy announces $250 solid-state lidar for cars, robots, and more. `https://spectrum.ieee.org/cars-that-think/transportation/sensors/quanergy-solid-state-lidar`, 2016.

[2] K. Zimmermann, T. Petricek, V. Salansky, and T. Svoboda. Learning for Active 3D Mapping. *ArXiv e-prints*, August 2017.

[3] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: an introduction*, volume 2. The MIT Press, 2012.

[4] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, and Georg et al. Ostrovski. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[5] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets robotics: The kitti dataset. *International Journal of Robotics Research (IJRR)*, 2013.

[6] Zdeněk Rozsypálek. Lidar-gym, training environment in openai interface. `https://gitlab.fel.cvut.cz/rozsyzde/lidar-gym`, 2018.

[7] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. In *Machine Learning*, pages 279–292, 1992.

[8] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized Experience Replay. *ArXiv e-prints*, November 2015.

[9] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *ArXiv e-prints*, September 2015.

[10] H. van Hasselt, A. Guez, and D. Silver. Deep Reinforcement Learning with Double Q-learning. *ArXiv e-prints*, September 2015.

[11] Richard S. Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Proceedings of the 12th International Conference on Neural Information Processing Systems*, NIPS'99, pages 1057–1063, Cambridge, MA, USA, 1999. MIT Press.

[12] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. 1, 06 2014.

[13] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal Policy Optimization Algorithms. *ArXiv e-prints*, July 2017.

[14] N. Heess, G. Wayne, D. Silver, T. Lillicrap, Y. Tassa, and T. Erez. Learning Continuous Control Policies by Stochastic Value Gradients. *ArXiv e-prints*, October 2015.

[15] Po-Wei Chou, Daniel Maturana, and Sebastian Scherer. Improving stochastic policy gradients in continuous control with deep reinforcement learning using the beta distribution. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 834–843, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR.

[16] G. Dulac-Arnold, R. Evans, H. van Hasselt, P. Sunehag, T. Lillicrap, J. Hunt, T. Mann, T. Weber, T. Degris, and B. Coppin. Deep Reinforcement Learning in Large Discrete Action Spaces. *ArXiv e-prints*, December 2015.

[17] M. Plappert, R. Houthooft, P. Dhariwal, S. Sidor, R. Y. Chen, X. Chen, T. Asfour, P. Abbeel, and M. Andrychowicz. Parameter Space Noise for Exploration. *ArXiv e-prints*, June 2017.

[18] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. OpenAI Gym. *ArXiv e-prints*, June 2016.

[19] Tomáš Petříček. Voxel map, simple c++ header-only library with matlab and python interfaces for dealing with 3-d voxel maps. `https://bitbucket.org/tpetricek/voxel_map`, 2017.

[20] P. Ramachandran and G. Varoquaux. Mayavi: 3D Visualization of Scientific Data. *Computing in Science & Engineering*, 13(2):40–51, 2011.

[21] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[22] François Chollet et al. Keras. `https://keras.io`, 2015.

[23] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Openai baselines. `https://github.com/openai/baselines`, 2017.

[24] A. Vedaldi and K. Lenc. Matconvnet – convolutional neural networks for matlab. In *Proceeding of the ACM Int. Conf. on Multimedia*, 2015.

# Appendix A CD Content

In Table 3 are listed names of all root directories on CD.

| **Directory name** | **Description** |
|---|---|
| thesis | the thesis in pdf format |
| thesis_sources | latex source codes |

Table 3: CD Content

# Appendix B List of abbreviations

In Table 4 are listed abbreviations used in this thesis.

| **Abbreviation** | **Meaning** |
|---|---|
| **API** | application programming interface |

Table 4: Lists of abbreviations