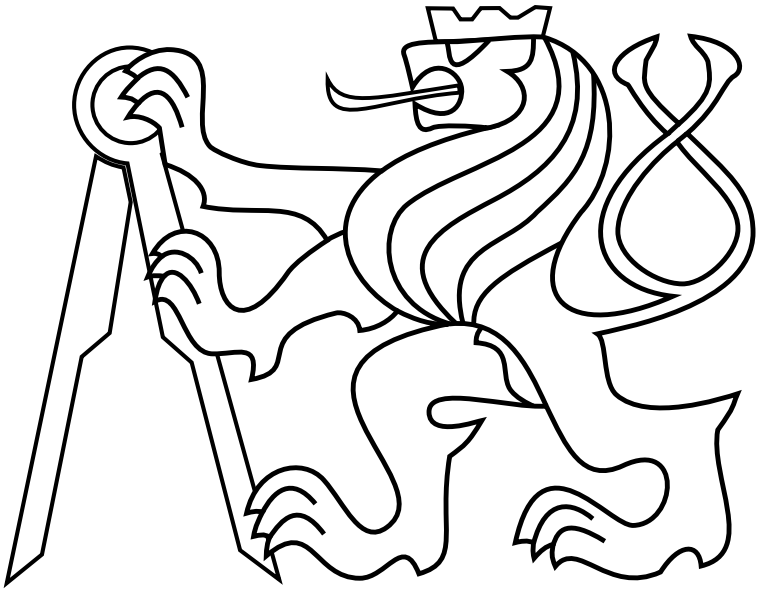
CZECH TECHNICAL UNIVERSITY IN PRAGUE

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**

Prohlášení autora práce

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne ....
podpis autora práce

Author statement for undergraduate thesis

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date ....
signature

BACHELOR'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Rozsypálek Zdeněk** Personal ID number: **457216**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Control Engineering**

Study program: **Cybernetics and Robotics**

Branch of study: **Systems and Control**

II. Bachelor's thesis details

Bachelor's thesis title in English:

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

Bachelor's thesis title in Czech:

Aktivní 3D mapování pomocí laserového dálkoměru s řiditelnými měřicími paprsky

Guidelines:

1. Consider the problem of active 3D mapping using a laser rangefinder with steerable measuring rays [1] in reinforcement learning setting, with actions encompassing the selected rays and the current 3D map (occupancy of voxels) built from past measurements, and reward corresponding to the map accuracy.
2. Briefly review state-of-the-art reinforcement learning algorithms implemented in OpenAI Baselines [2], such as [3, 4, 5], and select suitable algorithm(s) for the problem.
3. Design, implement, and train one or more agents using these (possibly adapted) algorithms in the Lidar gym environment [6] based on the publicly available KITTI dataset [7].
4. Evaluate the trained agent(s) and discuss the results, including learning curves and ROC curves for the final predicted occupancy. Compare the results to those of [1].

Bibliography / sources:

[1] Zimmermann, K. et al. Learning for active 3D mapping. In The IEEE International Conference on Computer Vision (ICCV), Oct. 2017, pp. 1548-1556. doi:10.1109/ICCV.2017.171
[2] Dhariwal, P. et al. OpenAI Baselines. In GitHub. URL: https://github.com/openai/baselines
[3] Mnih, V. et al. Human-level control through deep reinforcement learning. In Nature 518, pp. 529-533. doi:10.1038/nature14236
[4] Wang, Z. Sample Efficient Actor-Critic with Experience Replay. In International Conference on Learning Representations (ICLR), April 2017. URL: https://openreview.net/forum?id=HyM25Mqel
[5] Schulman, J. et al. Proximal Policy Optimization Algorithms. 2017. arXiv:1707.06347. URL: https://arxiv.org/abs/1707.06347
[6] Rozsypálek, Z. Lidar gym - OpenAI gym training environment for agents controlling solid-state lidars based on Kitti dataset. URL: https://gitlab.fel.cvut.cz/rozsyzde/lidar-gym
[7] Geiger, A. et al. Vision meets robotics: The KITTI dataset. In The International Journal of Robotics Research, 32(11), pp. 1231-1237.

Name and workplace of bachelor's thesis supervisor:

Ing. Tomáš Petříček, Vision for Robotics and Autonomous Systems, FEE

Name and workplace of second bachelor's thesis supervisor or consultant:

Date of bachelor's thesis assignment: **30.01.2018** Deadline for bachelor thesis submission: **25.05.2018**

Assignment valid until: **30.09.2019**

Ing. Tomáš Petříček	prof. Ing. Michael Šebek, DrSc.	prof. Ing. Pavel Ripka, CSc.
Supervisor's signature	Head of department's signature	Dean's signature

Acknowledgements

I would like to express my appreciation to Ing. Tomáš Petříček for his valuable and constructive suggestions during the planning and development of this thesis. I would also like to thank to the Department of Cybernetics of the Czech Technical University and to Michal Němec for the provided hardware. Finally, I wish to thank my family for support throughout my study.

Abstract

Bakalářská práce se zaměřuje na ovládání *solid-state* lidarů s omezeným počtem natáčecích paprsků. Kromě plánování směrů paprsků se práce věnuje i rekonstruování 3D mapy z řídkých měření těchto lidarů. V práci se pro rekonstruování a plánování používají hluboké neuronové sítě. Plánovací část využívá *reinforcement learning* metody pro trénink neuronových sítí. Bylo vytvořeno trénovací prostředí implementující framework pro trénování *reinforcement learning* agentů. Za pomocí stochastických metod se podařilo navrhnout agenta, který nabízí dostatečnou škálovatelnost a překonává náhodný plánovač.

Abstrakt

Bachelor thesis aims at control of the solid-state lidar sensor with a limited number of steerable rays. Besides planning the directions of the rays, thesis is also devoted to creating dense 3D maps from sparse measurements. The theses use a deep neural networks for planning the rays and reconstructing the dense maps. Planning part exploits the reinforcement learning concept for training of the neural network. An environment implementing a framework for training of reinforcement learning agents was created. The agent proposed in this thesis is using stochastic methods to achieve sufficient scalability in the challenging environment.

Keywords: Lidar, reinforcement learning, deep neural network, 3D mapping, voxel map.

Contents

List of Figures

1 Introduction

Lidar sensors offer an accurate distance measurement, which can be used for mapping surrounding space. There is much utilization of volumetric space reconstructions in different fields. For example, the 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 a broader implementation of these sensors is a relatively high price. Breakthrough in this field is a 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 a limited number of rays in chosen directions per timestamp. Zimmermann et al. [2] proposed a mapping agent which creates dense reconstructions from sparse measurements. They also proposed prioritized greedy planning for choosing the 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 the 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 a 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 the great success, they still suffer from a lack of sample efficiency - they need for training a lot of interactions with the environment. This inefficiency makes creating an 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 the RL is discussed in the first part of this thesis. In the second part are methods from the 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 by Markov decision process (MDP), where $S_t \in \mathcal{S}$ is a state from a set of possible states $\mathcal{S}$ in which the environment could be located in time t. In the environments with the observable MDP, an agent can observe the state of the environment and make the action accordingly. An action is a probabilistic transition between the states. Every action $A_t \in \mathcal{A}$ moves the environment from S_t to S_{t+1}. If we consider the lidar planning task, the action is sending the rays in certain direction and the state is given by the position of the sensor and by the already executed measurements. The environment evaluates every action and returns appropriate reward R_t (see Figure 1). In RL the set $\mathcal{A}$ is often called an action space and the set $\mathcal{S}$ an observation space. A return G_t is a sum of the discounted future rewards [3].

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

where $\gamma \in [0, 1]$ is a discount factor. The RL methods define how experience from interacting with the environment will change the policy. The major issue is that maximizing of the 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 the future states and rewards. The goal of the agent is to find policy π which maximizes the expected return. The agent use the value function $V_\pi(S_t)$ which assesses how advantageous is being in the state S_t with the policy π.

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

An optimal policy π^* is then defined for all $S_t \in \mathcal{S}$ as

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

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 a continuous action or observation space. Therefore, the modern methods use neural networks as function estimators.

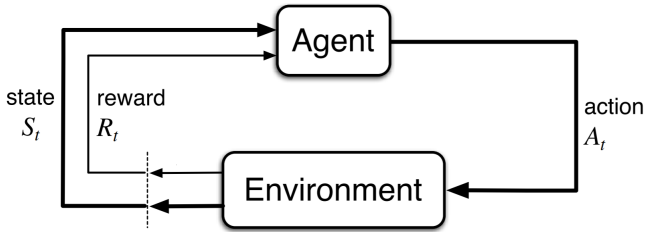


Figure 1: Illustration of RL concept. Source - [3].

2.1 Temporal difference learning

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

$$\delta_t = R_t + \gamma V(S_{t+1}) - V(S_t) \tag{4}$$
$$V(S_t) \leftarrow V(S_t) + \lambda\delta_t \tag{5}$$

where δ_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 the Q value, which refers to a quality of action in a particular state instead of the quality of the state itself. When we rewrite the TD learning (4) to the Q-learning we get:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \lambda[R_t + \gamma \max_{A_{t+1}} Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]. \tag{6}$$

The policy here is to take the action with the maximum Q value. That is called the 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 the ϵ-greedy policy.

Algorithm 1 ϵ-greedy policy in pseudocode

```
1: function ChooseAction
2:     ε ← ε · ε_d
3:     if ε > random ∈ (0, 1) then
4:         action ← random ∈ A
5:     else
6:         action ← max_{A_t} Q(S_t, A_t)
7:     return action
```

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

2.3 Prioritized experience replay

The prioritized experience replay is a 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 a high TD should have the higher priority. It is thus necessary to calculate the priority p from the TD error:

$$p = (|\delta_t| + \eta)^\rho \tag{7}$$

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

$$P(i) = \frac{p_i}{\sum_{j \in \mathcal{I}} p_j}, \tag{8}$$

where $\mathcal{I}$ is the set of all experience in the buffer. It is now possible to sample a batch of experience for training using this probability. It removes a correlation in the observation sequence and improves the sample efficiency of the DQN. It is feasible to store all the experience in a buffer sorted by the 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 the usage is in the 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)
```

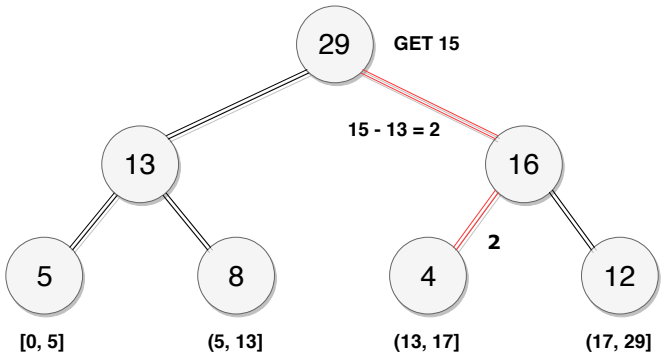


Figure 2: Simple example of sum tree.

3 Deep neural networks in RL

As was stated in the previous chapter, the tabular methods are very inefficient in the large environments. In these cases, it is possible to use the deep neural networks which can replace the tables. Deep Q networks (DQN) proposed by Google Deepmind [4] outperformed all previous RL algorithms in playing Atari games. With the 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 the neural networks.

3.1 Deep Q network

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

$$\delta_t = R_t + \gamma \max_{A_{t+1}} Q^\theta(S_{t+1}, A_{t+1}) - Q^\theta(S_t, A_t) \tag{9}$$

$$\theta_{t+1} = \theta_t + \lambda \delta_t \nabla_\theta Q^\theta(S_t, A_t). \tag{10}$$

The weights are updated in proportion to the TD δ_t. Unfortunately, this simple DQN agent suffers from a lack of the sample efficiency and often does not converge well. There are many techniques which can help to the DQNs to achieve satisfying results.

3.2 Target network

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

$$\theta^- = \tau\theta + (1 - \tau)\theta^-, \tag{11}$$

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

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

The target network stabilizes the training since the 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 a Double Q-learning which decompose the max operation into action selection and action evaluation [10].The TD is then computed by the following equation.

$$\delta = R_t + \gamma Q^{\theta^-}(S_{t+1}, \underset{A_{t+1}}{\text{argmax}} Q^{\theta}(S_{t+1}, A_{t+1})) - Q^{\theta}(S_t, A_t). \tag{13}$$

The double DQN outperforms the DQN in terms of the value accuracy and the policy quality.

4 Policy gradient

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

$$J = \mathbf{E}_{\pi}[G_t|S_t, A_t, \theta_t] \tag{14}$$

$$\theta_{t+1} = \theta_t + \lambda\widehat{\nabla_{\theta} J} \tag{15}$$

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

4.1 Actor-Critic

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

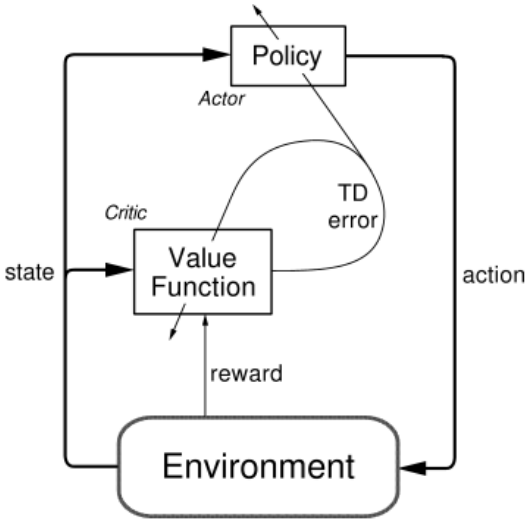


Figure 3: Actor-Critic framework. Source - [3].

Consider the critic using the Q-values for an update. θ and ω denote the trainable weights of actor and critic, respectively. Critic update is similar to DQN:

$$\delta_t = r_t + \gamma Q^{\omega}(S_{t+1}, \mu^{\theta}(S_{t+1})) - Q^{\omega}(S_t, A_t) \tag{16}$$

$$\omega_{t+1} = \omega_t + \lambda \delta_t \nabla_{\omega} Q^{\omega}(S_t, A_t). \tag{17}$$

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

$$\theta_{t+1} = \theta_t + \lambda \nabla_{\theta} \mu^{\theta}(S_t) \nabla_a Q^{\omega}(S_t, A_t)|_{a=\mu^{\theta}(S_t)}. \tag{18}$$

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

$$\frac{\partial Q^{\omega}(S_t, A_t)}{\partial \theta} = \frac{\partial Q^{\omega}(S_t, A_t)}{\partial A_t} \frac{\partial A_t}{\partial \theta}. \tag{19}$$

The neural network of the actor is updated by gradients which change the action output to maximize the Q-value of the critic [12]. There are other approaches, which doesn't use the 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 a frequently used approach. In this method the actor outputs a parameters of a probability distribution and the action itself is sampled from the parameterized distribution. It is a standard to use a normal distribution and predict a mean and variance of the action. The biggest advantage of the normal distribution is that it can be adjusted to the use of a backpropagation [14]. Another benefit of the stochastic actor is that it does not need any other techniques for the action space exploration.

4.2.1 Beta distribution

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

$$f(x;\alpha,\beta) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{\alpha-1}(1-x)^{\beta-1}, \tag{20}$$

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

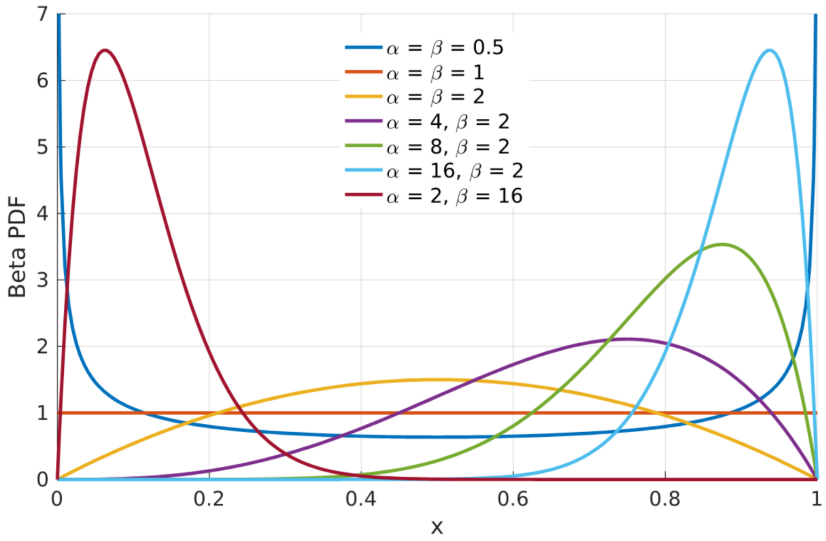


Figure 4: Probability density of beta distribution. Source - [15].

For the reinforcement learning is suitable only $\alpha, \beta \geq 1$. That makes the Beta distribution concave and unimodal. This can be ensured by using a 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 the stochastic actor predicts the distribution parameters and samples an action, the DDPG outputs the action directly. Silver [12] has shown that the deterministic policy can outperform its stochastic counterparts. A disadvantage of a deterministic approach is that it needs an additional policy to ensure the action space exploration. The exploration methods are discussed in the subsection 4.4.

4.4 Parameter and action space noise

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

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

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

Another approach is to apply the noise directly to the weights of the neural network of the actor. It can sometimes lead to more consistent exploration and richer behaviors [16].

$$\hat{\theta} = \theta + \mathcal{N}(0, \sigma^2) \tag{22}$$

where the policy using $\hat{\theta}$ is a so-called perturbed actor, which is interacting with the environment.

The major issue of the parameter space noise is that it is much harder to tune. When we use the action space noise, it is easier to estimate its impact on the actions (differences between both approaches can be seen in Figure 5). Because of an unpredictable influence of the parameter space noise is necessary to use an adaptive noise scaling.

$$d = |\mu^{\hat{\theta}}(S_t) - \mu^{\theta}(S_t)|_2 \tag{23}$$

$$\sigma_{t+1} = \begin{cases} \kappa\sigma_t & \text{if } d \leq T \\ \frac{1}{\kappa}\sigma_t & \text{otherwise} \end{cases} \tag{24}$$

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

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

$$\hat{A}_t = \mu^{\theta}(S_t) + \nu(\rho - \mu^{\theta}(S_t)) + \phi\mathcal{N}(0, 1), \tag{25}$$

where $\nu, \phi \in [0, 1]$ are constants of the random process and ρ is mean value around which we want to explore the action space. When $\nu = 0$ it is a basic exploration as in the expression (21).

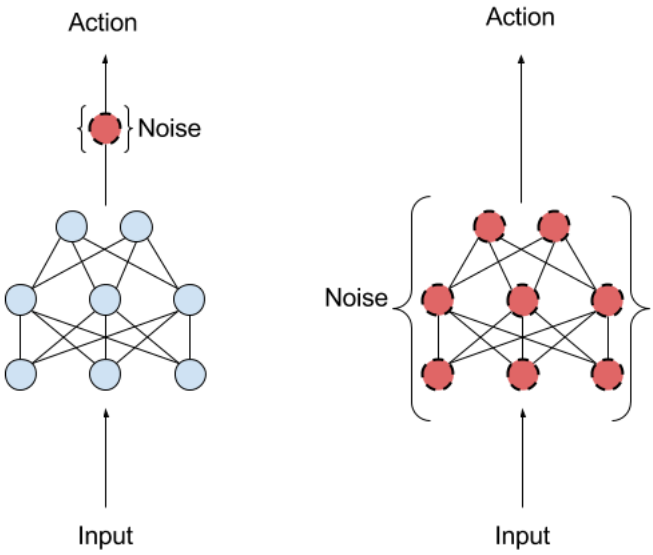


Figure 5: Left - action space noise, Right - parameter space noise. Source - [16].

5 Experiment

The experiment aims at using reinforcement learning algorithms for controling the solid-state lidar with a limited number of steerable rays. For purposes of the experiment it was neccessary to implement an environment, where an agent can learn and be evaluated [6]. The lidar-gym environment is written in Python 3 based on OpenAI gym interface [17]. It uses point clouds from the KITTI dataset drives[5]. One episode of the learning in the environment corresponds to a drive in the KITTI dataset. The large point clouds from the drives are processed into 3D voxel maps by C++ package [18], which also provides a 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.

$$\begin{aligned} c > 0 \quad & \text{occupied voxel} \\ c = 0 \quad & \text{unknown occupancy} \\ c < 0 \quad & \text{empty voxel.} \end{aligned} \tag{26}$$

5.1 Environments

Lidar-gym implements several environments (visualized in Figure 6), which follow the same template with different sizes of the voxel maps.The observation space is a local cutout of the voxel map, which provides occupancies from sparse measurements of the sensor. The sensor is located in the quarter of x-axis and half of y-axis and z-axis of the local cutout. The action space is divided into two parts. The first part is the dense voxel map reconstructed from the observations (sparse measurements). The second part of the action space are directions of the measuring rays. Each ray has an own azimuth and elevation. The environment expects directions in the format of a 2D array of booleans, where true means a fired ray. The reward function of the environment is negative logistic loss $-L$ (27). Parameters of the 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

*Using GPU Nvidia 1080Ti.

The environments also offer a visualization of actions using Mayavi [19] and ASCII art. The agents use neural networks as function estimators, which are implemented in Tensorflow [20] and Keras [21].

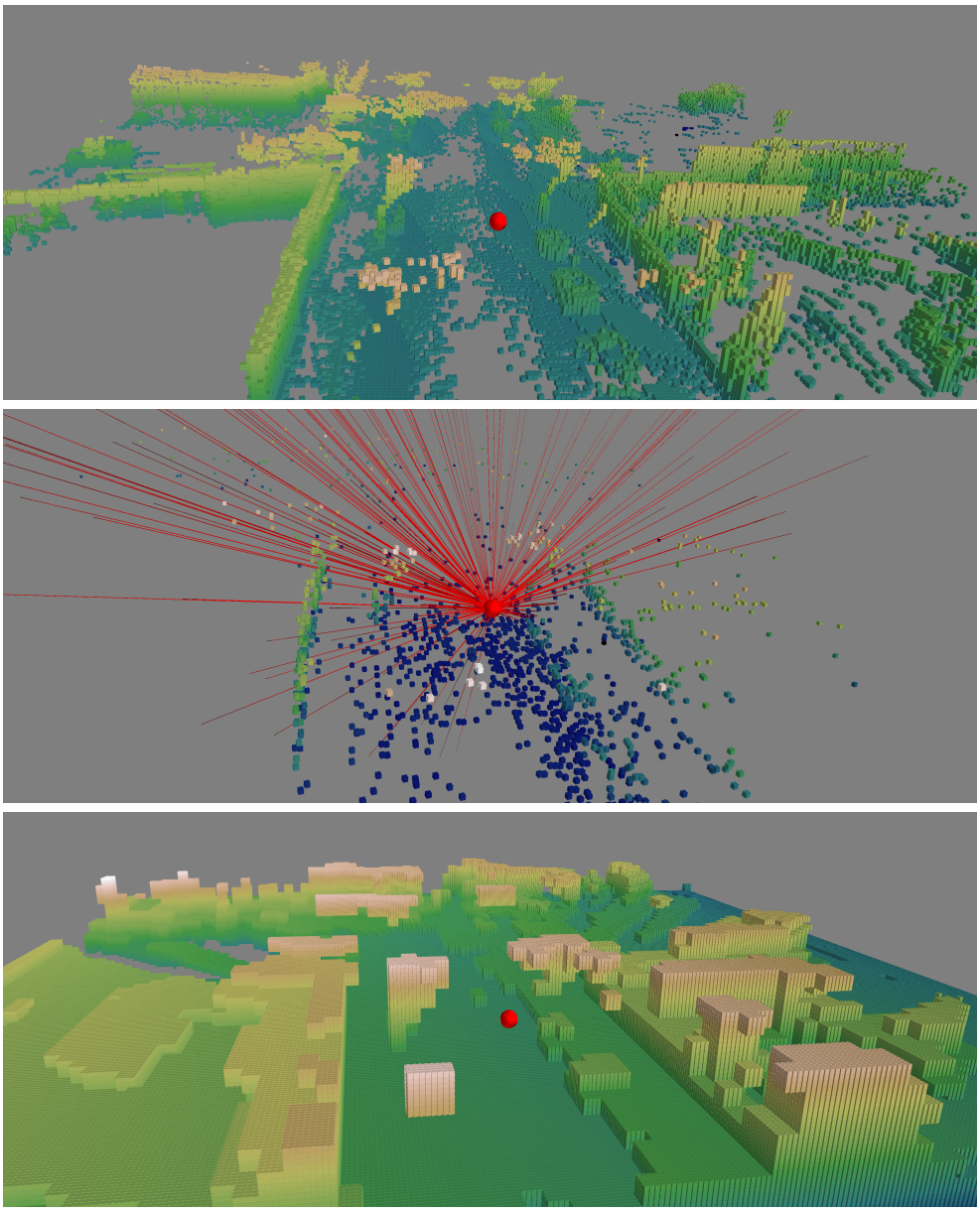


Figure 6: Visualization of the large environment: The first figure is the ground truth map, second is the voxel map of the sparse measurements and the third figure shows the dense reconstruction. The dense reconstruction in the third figure and the rays fired in the second figure are made by an agent using the random planner. Some known structures as cars and trees can be seen in the reconstructed map.

Due to the high time complexity, all experiments were conducted in the toy environment. The RL agents need significantly more training steps than supervised agents. Unlike the RL agents, for the supervised agent is known desired output, thus it can be learned by a gradient descent on the loss function between made and desired outputs. There are the RL agents trained for over million epochs in OpenAI baselines [22]. A 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 the dense map from the sparse measurements. 3D convolutional layers are used to learn features and max-pooling layers are used to avoid overfitting. The CNN architecture is described in Figure 7.

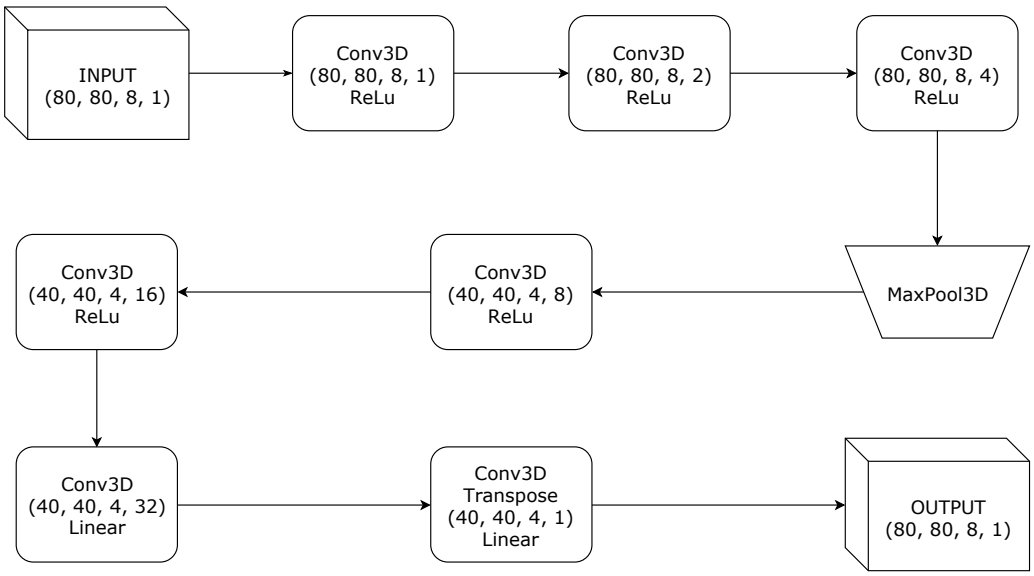


Figure 7: Input of the supervised mapping agent is voxel map containing the sparse measurements. The output is the dense reconstruction of the input.

Gradient descent is made by Adam optimizer [23]. Optimizer uses a logistic loss L between a ground truth map Y and predicted dense map $\hat{Y}$.

$$L(Y, \hat{Y}) = \sum_i w_i \log(1 + \exp(-Y_i \hat{Y}_i)) \tag{27}$$

where w are weights which balance importance of the 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, the following modified loss was used [24]

$$\begin{aligned} a_i &= -Y_i \hat{Y}_i \\ b_i &= \max(0, a_i) \\ L &= \sum_i w_i (b_i + \log(\exp(-b_i) + \exp(a_i - b_i))). \end{aligned} \tag{28}$$

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

5.3 Discrete planning agent

Since the action of the environment A_t for directions of the rays is 2D binary array, first try is to use a discrete agent. The DQN is the most used option for discrete action space, but in this use case, it requires some tweaks. Note that the number of the possible actions is extremely large. Even in the toy environment it is $\binom{40\times30}{15} \approx 10^{34}$ of actions. Thus it is necessary to emphasize the action space exploration. Further arises the problem with the ϵ-greedy policy, because we are unable to process all the possible actions and pick the one with the biggest Q-value. We consider only one ray as action to resolve this issue. For K rays is the TD from (9) now computed as:

$$\begin{aligned} q(S_t, A_t) &= \max_{A_t}^{K} Q^\theta(S_t, A_t) \\ \delta_t &= R_t + \gamma \overline{q}(S_{t+1}, A_{t+1}) - \overline{q}(S_t, A_t) \end{aligned} \tag{29}$$

where $\overline{q}$ is an average Q value over K actions with maximum Q values. The 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. The exploration is ensured by the action space noise. The 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 λ.

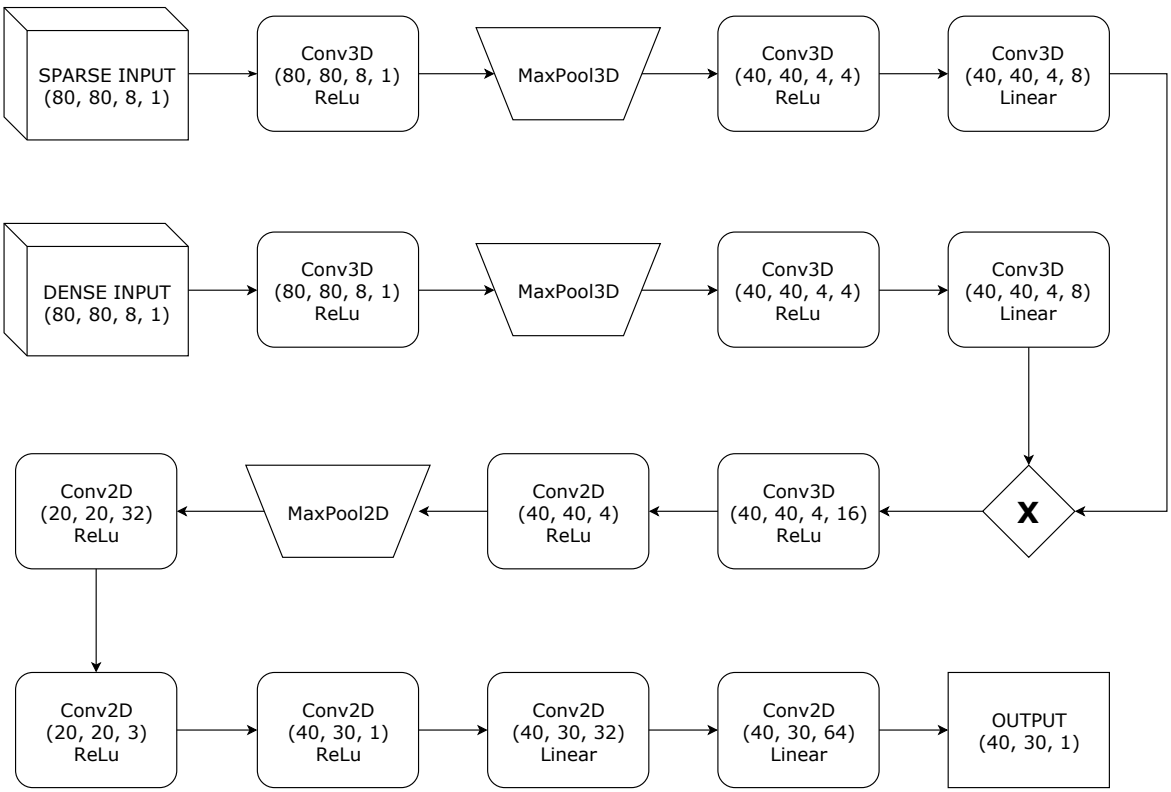


Figure 8: The DQN outputs an estimate of the Q-value. The network takes as the input two voxel maps. In conducted experimets was significantly better to merge these two inputs using multiplication than using addition.

5.4 Continuous planning agent

The discrete action output was substituted by a continuous action, which is then mapped into the 2D binary array. This will allow the agent to avoid the extremely large discrete action space. Thank to this substitution it is possible to exploit the actor-critic framework. The output of the actor is now 2 by K array where the first row is the elevation and the second row 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 Figure 9.

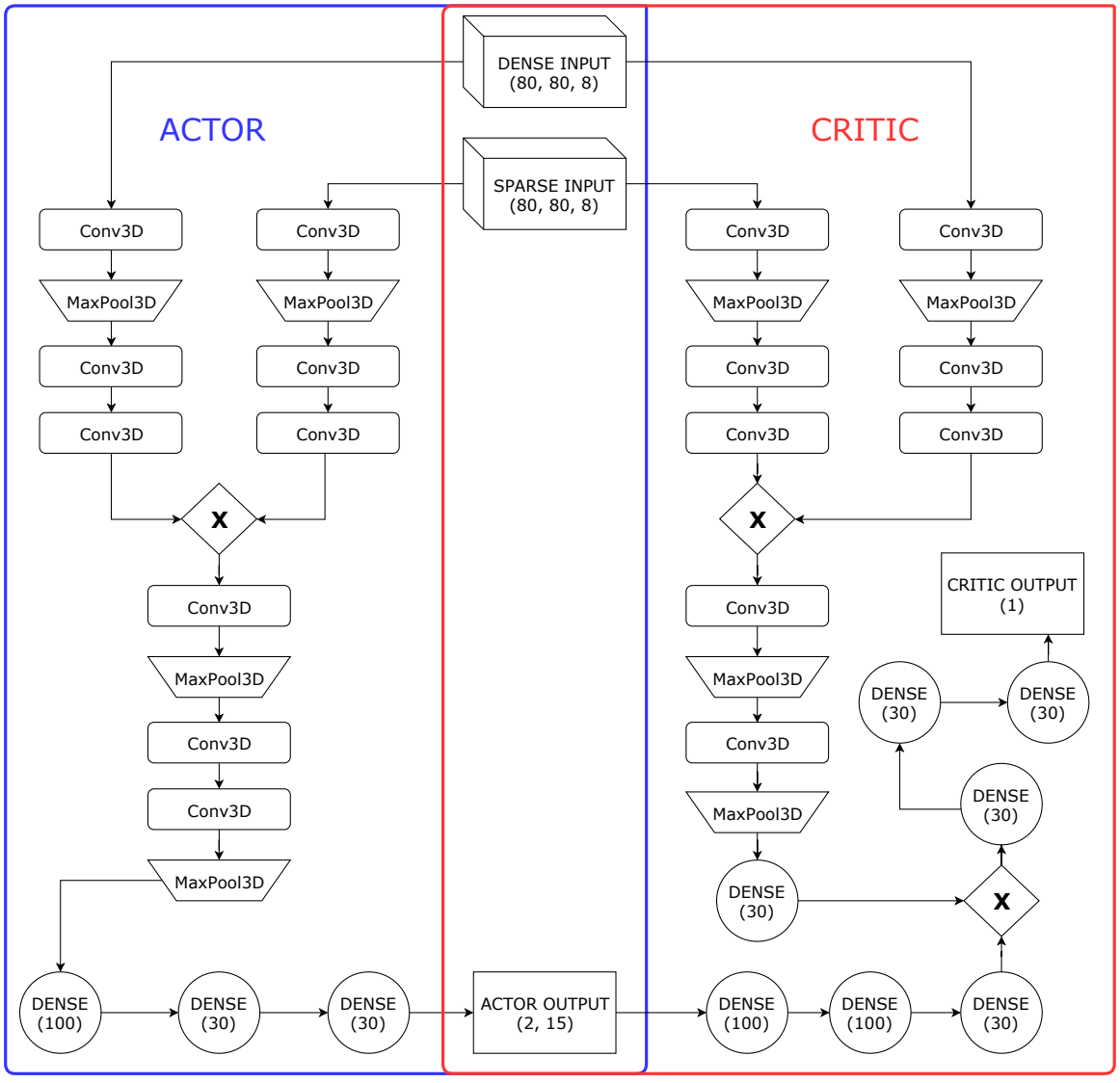


Figure 9: The architecture of the actor is on the left side and the critic is on the right side. In the middle can be seen shared inputs and outputs.

To explore the action space in our experiments correctly, 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 λ_a, λ_c. The target networks are used to stabilize the learning of the actor and critic. The continuous agent also uses prioritized experience replay.

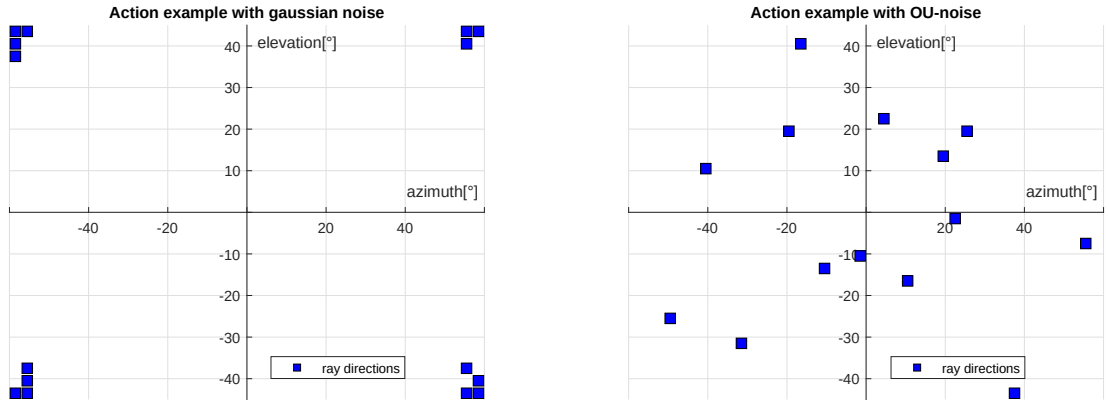


Figure 10: The difference between actions made by agents with the different exploration methods. Left figure is an example of the action made by an agent with Gaussian noise used for the exploration. The action in the right figure is made by the agent trained using Ornstein-Uhlenbeck noise.

5.5 Stochastic planning agent

There is an obvious issue with the deterministic actor. For efficient exploring of the 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 the 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 the Beta distribution and preserves the Actor-Critic framework. The architecture of stochastic agent is similar to the DDPG from the previous subsection, the only difference is that the agent now outputs only two values - the distribution parameters. The action is then sampled with distribution probabilities. Stochastic actor output can be seen in Figure 11.

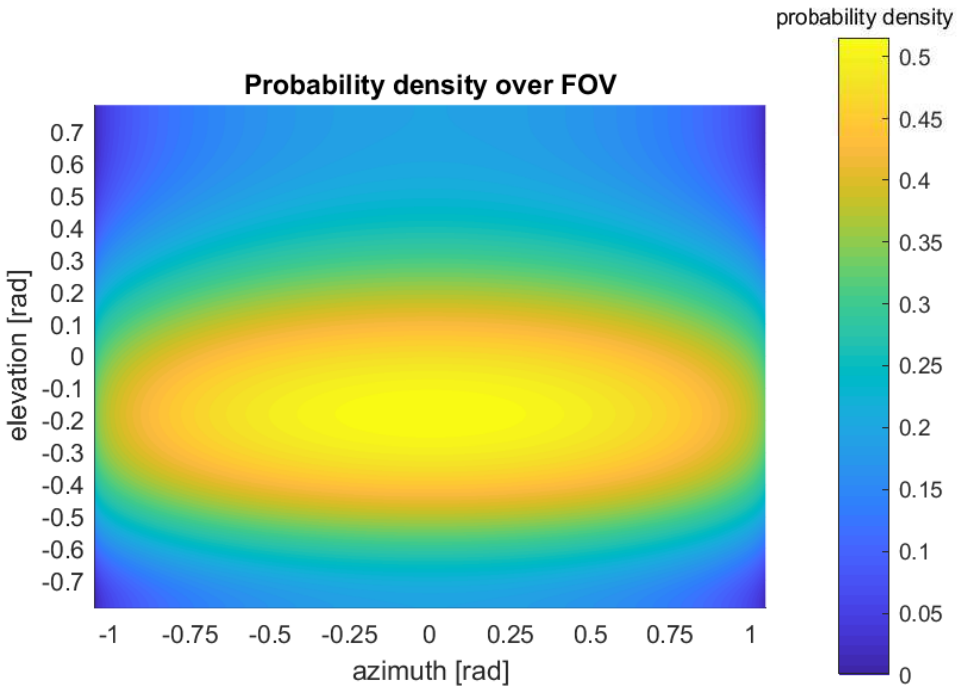


Figure 11: The example of stochastic agent output - probability density of choosing a ray with specific azimuth and elevation.

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 (29), did not help the agent to converge. The continuous DDPG agent also did not converge successfully. For DDPG agent is hard to change direction of the rays efficiently in the subsequent epochs and it often makes several same actions in a row which degrades performance a lot. Only the stochastic agent was able to outperform a random agent. Summary of the agents performances is made in Figure 12. It turns out that it is very difficult to get satisfying results. The stochastic agent converges, but it does not use any kind of the advanced strategy. The output of the DDPG actor does not even allow making any sophisticated action. Therefore were also conducted experiments with an extended stochastic agent, which uses the Beta distribution for each ray separately, but with no significant success. Another advantage of the simple stochastic agent is its scalability. This agent can be adjusted to the large environment much easier than any other used agent. The parameters used for the training of these agents are described in Table 2.

Parameter	Value
γ	0.09
λ_c	0.001
λ_a	0.0001
τ	0.01
Memory size	4096
Batch size	8

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

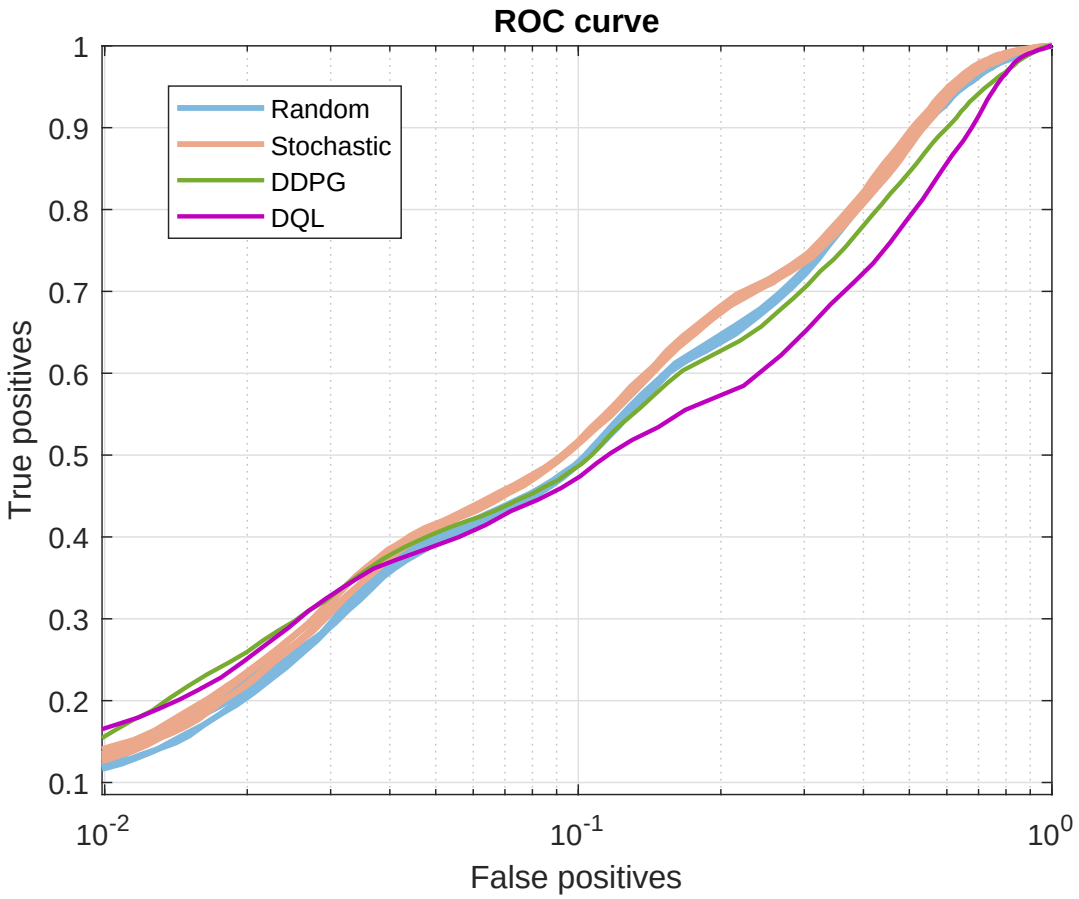


Figure 12: False positives were calculated using the ground truth map and true positives using all voxels, which could be possibly hit by the sensor. Five different evaluations are displayed for the non-deterministic agents.

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 the 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

We propose further experiments with an agent, which stands between the simple and the extended stochastic agent. The extended stochastic agent has the 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 the network architecture used in the experiments. On the other hand, when only one distribution is outputted for all rays, it does not allow the agent to create an advanced policies, because the Beta distribution considered in this thesis is always unimodal. A solution could be to output for example three different distributions, each describing five rays. That would allow agent to output a density function with multiple local maxima.

Another improvement could be achieved by adjusting the neural network architecture. Especially splitting the network into two subnetworks before the output or different types of merging the input layers can have a significant impact on performance. Finally, the reinforcement learning agent can be almost always improved by a better reward function, but we find very difficult to improve the existing reward function.

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] 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.

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

[18] 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.

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

[20] 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.

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

[22] 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.

[23] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. *ArXiv e-prints*, December 2014.

[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
ctu_thesis	latex source codes
lidar-gym	OpenAI gym environment

Table 3: CD Content

Appendix B List of abbreviations

In Table 4 are listed abbreviations used in this thesis.

Abbreviation	**Meaning**
CNN	Convolutional neural network
DDPG	Deep deterministic policy gradients
DQN	Deep Q-learning
MDP	Markov decision process
RL	Reinforcement learning
ReLu	Rectified linear unit
TD	Temporal difference

Table 4: Lists of abbreviations