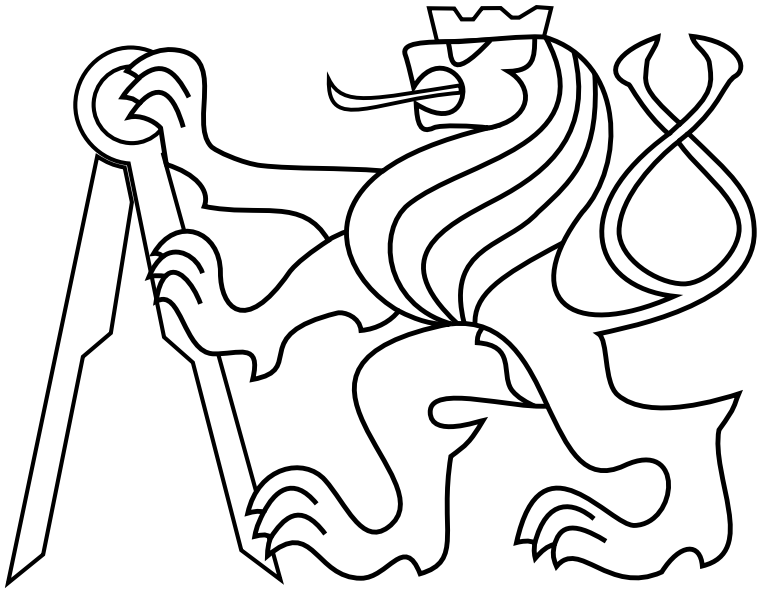
CZECH TECHNICAL UNIVERSITY IN PRAGUE

Faculty of Electrical Engineering

BACHELOR'S THESIS



Zdeněk Rozsypálek

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

Department of Cybernetics

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

1 Introduction

Reinforcement learning (RL) is a field of study consisting mainly of dynamic programming and machine learning. It is based on concepts of behavioural 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 [1]. One of the biggest achievements was playing Atari games by an RL agent without any prior knowledge of the environment [2]. Soon after the RL agent, able to solve simple continuous problems such as balancing inverse pendulum on a cart, was introduced. Today state-of-the-art methods can solve complex environments with infinite action spaces. The objective of this thesis is to apply these methods to control solid-state lidar sensor with very limited number of rays [3]. Price of this sensors should be circa hundreds of dollars in contrast to Velodyne lidars with price tag above $5000. Thesis is motived mainly by need of autonomous vehicles to do efficient 3D volumetric mapping using low cost sensors. The agent is divided into two parts - mapping and planning. The mapping part should create a best possible reconstruction from sparse measurements, while the planning part is focused on picking rays that will maximise reconstruction accuracy. This thesis is based on the work of Zimmermann et al, which proposed a supervised learning agent for mapping and a prioritised greedy policy for planning rays [4]. Agents are trained using publicly available dataset which contains drives of car equipped with Velodyne lidar [5].

2 RL basics

Firstly, environment where an agent is able to operate must be defined. Environment can be described as Markov decision process, 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. Agent can usually observe the state of the environment and take action accordingly. 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. The main goal of the agent is to find policy π which maximises expected return. Return G_t is a sum of discounted future rewards [1].

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

where $\gamma \in [0, 1]$ is discount factor. RL methods define how experiences from interacting with environment will change the policy. Major issue is that maximising immediate reward is often not an effective approach to maximise expected sum of discounted rewards. This greedy policy can take the agent into very disadvantageous state. Thus, the agent must take into account future states and rewards. This is done by value function $V_pi(S_t)$ which assesses how advantageous is being in state S_t with policy π.

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

Optimal policy π^* 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. This 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.

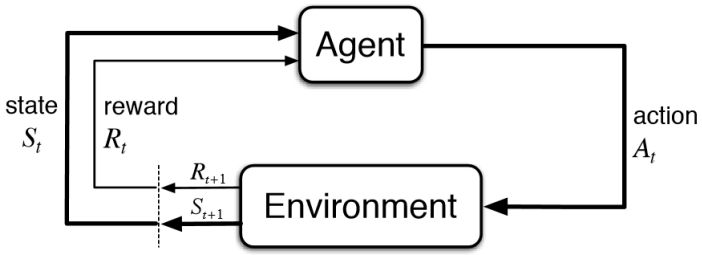


Figure 1: RL concept

2.1 Temporal difference learning

Temporal difference (TD) learning combines the ideas of Monte Carlo methods and dynamic programming. It is able to learn directly from experience obtained by interactions with an environment without any prior knowledge of said environment. TD learning is done by following assignment in each timestamp [1]

$$V(S_t) \leftarrow V(S_t) + \alpha[R_t + \gamma V(S_{t+1}) - V(S_t)] \tag{4}$$

where $\alpha \in \mathbb{R}^+$ is step size.

2.2 Q-learning

Q-learning is type of TD learning developed by Watkins [6]. The state value V from previous subsection is replaced by Q value, which refers to quality of action in a particular state instead of quality of the state itself. When we rewrite TD learning (4) to Q-learning we get:

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

Our policy here is to take action with maximal Q value. That is called greedy policy. 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 sometimes take random action to explore the environment. This policy is often referred to as ϵ-greedy policy.

Algorithm 1 ϵ-greedy policy

```
1: procedure 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:     end if
8:     return action
9: end procedure
```

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

3 Deep neural networks in RL

As was stated in previous chapter, tabular methods are very inefficient in large environments. In these instances is possible to use deep neural networks which can replace tables. Deep Q networks (DQN) proposed by Googles Deepmind [2] outperformed all previous RL algorithms in playing Atari games. With neural networks grew also the popularity of policy gradient methods where function estimator outputs an action instead of Q values. Note that the most of these methods are general and not necessarily tied to neural networks.

3.1 Deep Q network

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

$$\delta_t = R_t + \gamma \max_{A_{t+1}} Q^\theta(S_{t+1}, A_{t+1}) - Q^\theta(S_t, A_t) \tag{6}$$

$$\theta_{t+1} = \theta_t + \alpha \delta_t \nabla_\theta Q^\theta(S_t, A_t). \tag{7}$$

We are updating gradients in proportion to TD δ_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 convergence of DQN learning [2]. 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

$$\theta^- = \tau\theta + (1 - \tau)\theta^- \tag{8}$$

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

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

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

3.3 Prioritized experience replay

Experience replay is biologically inspired mechanism introduced by Schaul et al. [7] which stores all experiences (specifically: S_t, A_t, R_t, S_{t+1}) into a buffer and assigns priority to every experience. Main idea is that experiences with high TD should have higher priority. It is thus necessary to calculate priority p from TD error:

$$p = (|\delta_t| + \beta)^\rho \tag{10}$$

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

$$P(i) = \frac{p_i}{\sum_{j \in \mathcal{I}} p_j} \tag{11}$$

where $\mathcal{I}$ is set of all experiences in the buffer. It is possible now to sample a batch of experiences for training using this probability. It removes correlation in the observation sequence and improves sample efficiency of DQN. It is feasible to store all experiences in a buffer sorted by priority but a more efficient implementation is a sum tree.

3.4 Double Q-learning

Classic Q-learning algorithm tends to overestimate actions under certain conditions. Hasselt et al. propose idea of Double Q-learning which decompose the max operation into action selection and action evaluation [8]. TD is then computed by following equation.

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

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

4 Policy gradient

By this section the goal of neural network was predicting values on the basis of which we determined the policy. In policy gradient method neural network approximates the policy itself.

$$\theta_{t+1} = \theta_t + \alpha\widehat{\nabla J(\theta_t)} \tag{13}$$

where J is performance measure with respect to our neural network parameters and $\widehat{\nabla J(\theta_t)}$ is stochastic estimate which approximates gradient of performance measure. In other words, this method is basically doing stochastic gradient ascent of J with respect to θ [9]. Policy gradient methods are outperforming DQNs especially in continuous action spaces, because their output is directly continuous action instead of Q-value for every possible action.

4.1 Actor-Critic

Thanks to predicting action directly, we gain possibility to predict in continuous action space, but we lost the Q-value which assessed the advantage of action in certain state. That it 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 2.

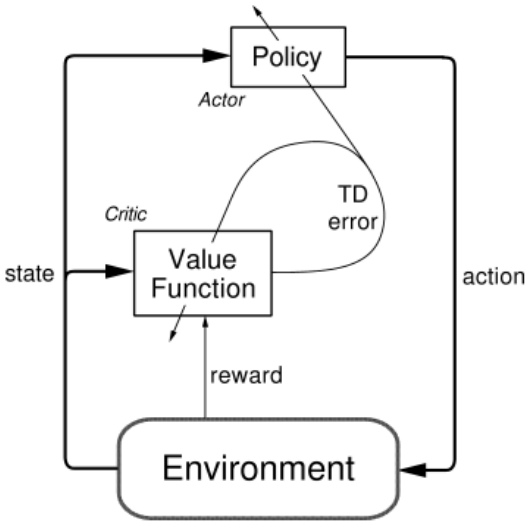


Figure 2: Actor-Critic framework

4.2 Deterministic policy gradients

Deep deterministic policy gradient (DDPG) is one of methods exploiting the Actor-Critic framework. Before DDPG was common practice to use stochastic actor, which predicts parameters of distribution (usually normal distribution). Action of stochastic actor is then a random sample from predicted distribution. Whereas deterministic actor uses distribution sampling only for exploration of action space. We denote θ and ω for trainable weights of actor and critic, respectively. Critic update is very similar to DQN:

$$\delta_t = r_t + \gamma Q^\omega(S_{t+1}, \mu^\theta(S_{t+1})) - Q^\omega(S_t, A_t) \tag{14}$$

$$\omega_{t+1} = \omega_t + \alpha\delta_t \nabla_\omega Q^\omega(S_t, A_t). \tag{15}$$

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.

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta \mu^\theta(S_t) \nabla_a Q^\omega(S_t, A_t)|_{a=\mu^\theta(S_t)}. \tag{16}$$

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

$$\frac{\partial Q^\omega(S_t, A_t)}{\partial \theta} = \frac{\partial Q^\omega(S_t, A_t)}{\partial A_t} \frac{\partial A_t}{\partial \theta}. \tag{17}$$

DDPG significantly outperforms its stochastic counterparts, especially in big continuous action spaces [10].

4.3 Wolpetinger policy

Actor-Critic methods and DDPG work well in continuous action spaces, but there is a lot of usecases with large discrete action spaces, such as recommender systems or lidar planning. Wolpetinger policy is approach how to utilize DDPG in discrete action space [11]. Whole policy is illustrated in figure 3. Actor doesn't predict action directly, but it predicts so called proto-action $\tilde{A}_t$.

$$\tilde{A}_t = \mu^\theta(S_t). \tag{18}$$

Proto action mostly isn't valid action $\tilde{A}_t \notin \mathcal{A}$. Thus it is necessary to find valid action corresponding to proto action. This is done by computing euclidean distance to every possible action.

$$\mathcal{A}_{knn} = \underset{a \in \mathcal{A}}{\overset{N}{\operatorname{argmin}}} |a - \tilde{A}_t|_2. \tag{19}$$

Usually policy choose N closest actions to the proto action. $\mathcal{A}_{knn}$ is the set of closest action to the proto action. Whole set is then assessed by critic and action with highest Q-value is finally picked.

$$A_t = \underset{a \in \mathcal{A}_{knn}}{\operatorname{argmax}} Q^\omega(S_t, a) \tag{20}$$

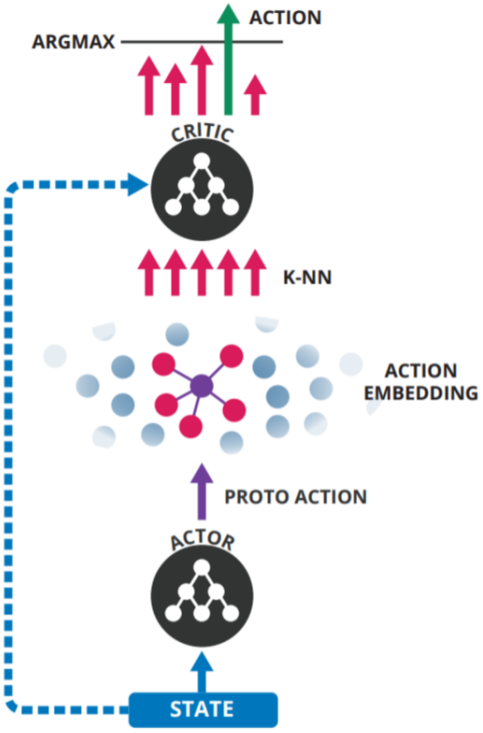


Figure 3: Wolpetinger policy illustration

4.4 Parameter and action space noise

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

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

where $\mathcal{N}$ is 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 actors weights. It can sometimes lead to more consistent exploration and richer behaviours [12].

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

where $\hat{\theta}$ is so called perturbed actor, which is interacting with environment. 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 4). Because of unpredictable influence of parameter space noise is necessary to use adaptive noise scaling.

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

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

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

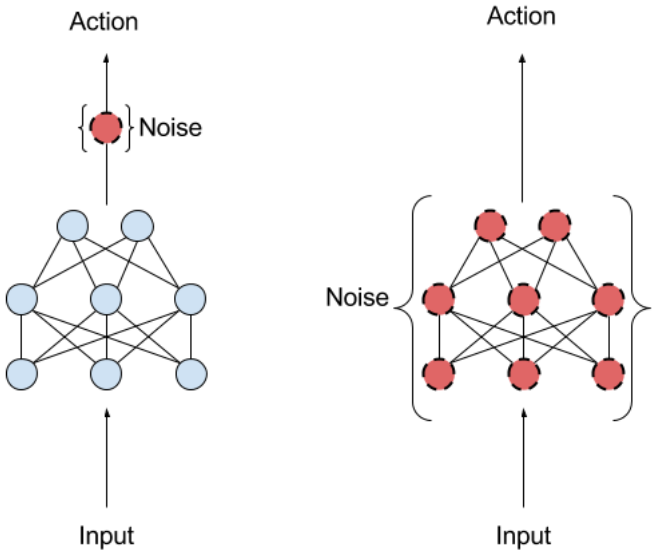


Figure 4: Action vs. parameter space noise

When is necessary to explore action space near to some specific action or include momentum of environment, it is possible to use Ornstein-Uhlenbeck random process [13].

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

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

5 Experiment

Experiment aims at using reinforcement learning algorithms for solid-state lidar with steerable rays and limited number of rays. At first it was necessary to create environment where the agent can learn and evaluate [14]. Lidar-gym environment is written in python 3 based on OpenAI gym interface [15]. It uses point clouds from Kitti dataset drives[5]. One episode of learning in environment corresponds to one drive in Kitti dataset. Large point clouds from drives are processed into 3D voxel maps by C++ package [16], which also provides ray tracing engine for environment. Every voxel map is 3D array containing real numbers which corresponds to the occupancy c of each voxel.

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

Environment also offers visualisation of actions using Mayavi [17] and ASCII art. Agents use neural networks as function estimators, which are handled by Tensorflow [18] and Keras [19].

5.1 Environments

Lidar-gym implements several environments, which follow same template with different sizes. Observation space is local cutout of voxel map, which provides occupancies from sensor sparse measurements. Sensor is located in quarter of x axis and half of y and z axis of local cutout. Action space is divided into two parts. First part is dense voxel map reconstructed from observations (sparse measurements). Second part of action space are directions of measuring rays. Each ray has own azimuth and elevation. Environment expects directions in format of 2D array of booleans, where true means fired ray. Environments reward is negative logistic loss $-L$ (27). Lidar-gym defines environments with parameters 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: Environment description

*Using GPU Nvidia 1080Ti.

Due to the high time complexity were all experiments conducted in toy environment. RL agents need significantly more training steps than supervised agents. In OpenAI baselines [20] are RL agents trained for over million timestamps. One drive in Kitti dataset has on average 200 timestamps. All agents was trained and evaluated on different drives from city part of dataset.

5.2 Mapping agent

Mapping agent is based on work of Zimmermann et al [4]. It uses 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 5.

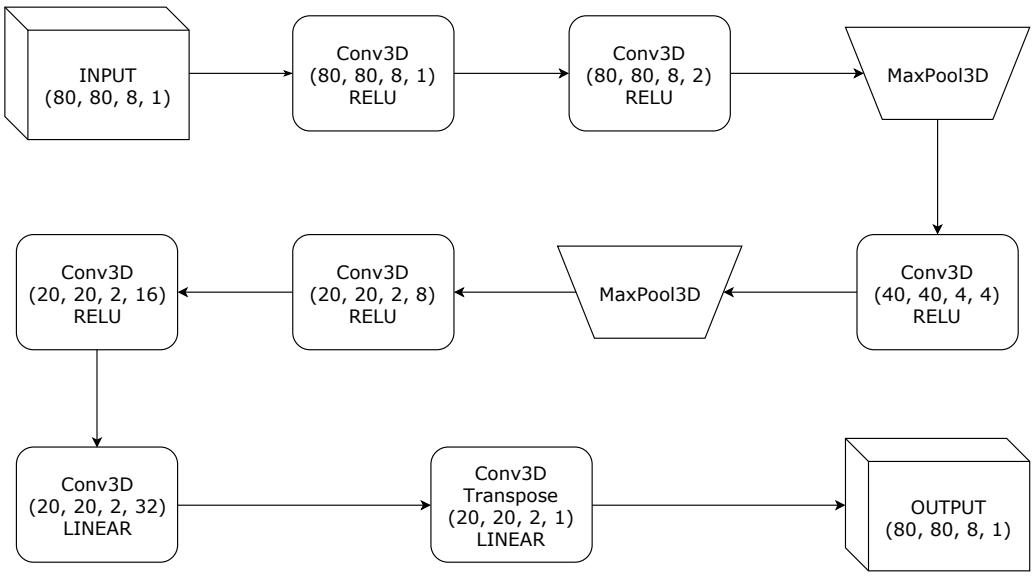


Figure 5: Mapping network architecture

For gradient descent is used logistic loss L between ground truth map Y and predicted dense map $\hat{Y}$.

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

where w are weights which balance importance of occupied and unoccupied voxels. Unfor-

tunately naive implementation of this loss function is computationally inconvenient and often cause numerical issues as underflow or overflow. To stabilize training following modified loss was used [21].

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

At first we train mapping agent with random ray planning. Reconstructions of 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

Insomuch as environment input for direction of the rays is 2D binary array, first try is to use 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 is necessary to emphasize on action space exploration. Further arises problem with ϵ-greedy policy, because we are unable to process all possible actions and pick one with the biggest Q-value. To solve this issue is considered one ray as an action and for K rays is TD from (6) now computed as:

$$q(S_t, A_t) = \max_{A_t}^{K} Q^{\theta}(S_t, A_t) \tag{29}$$

$$\delta_t = R_t + \gamma \overline{q}(S_{t+1}, A_{t+1}) - \overline{q}(S_t, A_t) \tag{30}$$

where $\overline{q}$ is average Q value over K actions with maximal Q values. DQN agent implements all available features described in theoretical part of this thesis as Prioritized experience replay, target network and double Q learning. Exploration is ensured by action space noise. Parameter values of agent are shown in table 2 and neural network architecture in figure 6. For gradient descent is used Adam optimizer.

Parameter	Value
γ	X
α	X
τ	X
ϵ_d	X

Table 2: Parameters of DQL agent

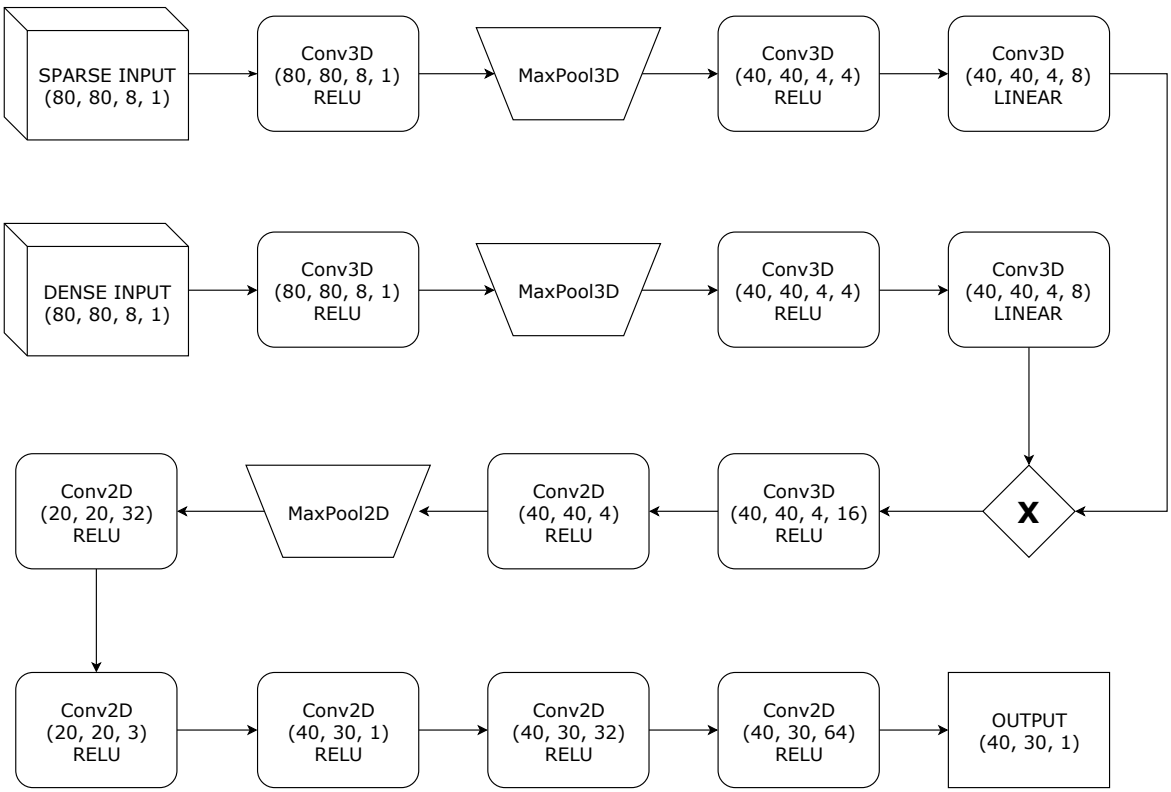


Figure 6: DQN architecture

5.3.1 Wolpetinger policy

Wolpetinger policy is currently state-of-the-art method for discrete action spaces, because it utilize actor-critic continuous methods. It was tested on recommender systems with over a million discrete actions [11]. That is large action space, but significantly smaller than action space of Lidar-gym environment. Obvious problem comes in place during action embedding in formula (19). It is not possible to compute KNN with so many possible actions. When is KNN substituted by picking rays with highest value, agent does not converge well. It abandons some rays very soon and get stuck in local optimum.

5.4 Continuous planning agent

To avoid extremely large actions was discrete action space substituted by continuous action, which is then mapped into 2D binary array. Thank to this change it is possible to exploit actor-critic framework. Output of actor is now 2 by K array where first row is the elevation and second is the azimuth of each ray. Last layer of the actor network is tanh function, so its output is element of $[-1, 1]$. As training algorithm is used DDPG. To explore action space correctly it is necessary to apply Ornstein-Uhlenback random process. When we use only normal distribution for action space noise, actions tend to converge into the corners very fast.

6 Conclusion

BLAH BLAH BLAH

References

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

[2] 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.

[3] 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.

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

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

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

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

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

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

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

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

[16] 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.

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

[18] 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.

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

[20] 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.

[21] 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

List of Figures