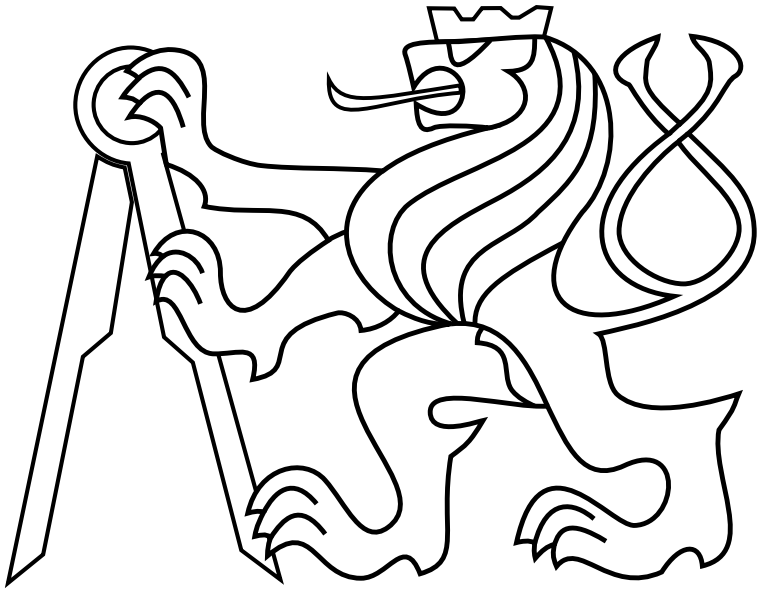
CZECH TECHNICAL UNIVERSITY IN PRAGUE

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 [9]. One of the biggest achievements was playing Atari games by an RL agent without any prior knowledge of the environment [4]. Soon after an 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 [1]. Price of this sensors should be circa hundreds of dollars in contrast to Velodyne lidars with price tag above $5000. 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 and his team, which proposed a supervised learning agent for mapping and a prioritised greedy policy for planning rays [12]. Thesis is motived mainly by need of autonomous vehicles to do efficient 3D volumetric mapping using low cost sensors. Agents are trained using publicly available dataset which contains drives of car equipped with Velodyne lidar [3].

2 RL basics

Firstly, environment where an agent is able 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 observe the state of the environment and take action accordingly. Action is a 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. In RL set $\mathcal{A}$ is often called action space and set $\mathcal{S}$ observation space. The main goal of the agent is to maximise a expected reward.

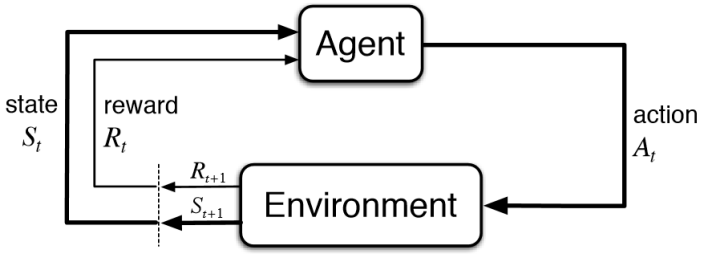


Figure 1: RL concept

Major issue is that maximising immediate reward is often not an effective approach to maximising the overall reward. This greedy policy can take the agent into very disadvantageous state. Thus, the agent must take into account future states and rewards. In the past agents used to contain big tables which stored information about the quality of every action in every state. 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.

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 an assignment in each timestamp [9]

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

where V is so called state value, which shows how good is being in a particular state with the current policy. $\alpha \in \mathbb{R}^+$ is step size and $\gamma \in (0, 1)$ is discount factor.

2.2 Q-learning

Q-learning is type of TD learning developed by Watkins [1989]. 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 (1) to Q-learning we get:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_t + \gamma \underset{A_{t+1}}{max} Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)]. \tag{2}$$

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: $\epsilon \leftarrow \epsilon \cdot \epsilon_d$
3: **if** $\epsilon >$ random $\in (0, 1)$ **then**
4: action $\leftarrow$ random $\in \mathcal{A}$
5: **else**
6: action $\leftarrow \underset{A_t}{max} Q(S_t, A_t)$
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. 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 deep neural networks which can replace tables come into effect. Deep Q networks (DQN) proposed by Google's Deepmind [4] outperformed all previous RL algorithms in playing Atari games. With neural networks, also grew the popularity of policy gradient methods, where neural network 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.

$$\delta_t = R_t + \gamma \underset{A_{t+1}}{max} Q^\theta(S_{t+1}, A_{t+1}) - Q^\theta(S_t, A_t) \tag{3}$$

$$\theta_{t+1} = \theta_t + \alpha \delta_t \nabla_\theta Q^\theta(S_t, A_t). \tag{4}$$

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 [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

$$\theta^- = \tau\theta + (1 - \tau)\theta^- \tag{5}$$

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

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

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. Thus is necessary to calculate priority p from TD error:

$$p = (|\delta_t| + \beta)^\alpha \tag{7}$$

where α indicates how much we prefer experiences with higher priority and $\beta << 1$ 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:

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

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 [11]. Target value is then computed by following equation.

$$Y = R_t + \gamma Q(S_{t+1}, argmax_{A_{t+1}} Q(S_{t+1}, A_{t+1}; \theta); \theta^-). \tag{9}$$

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.

$$\theta_{t+1} = \theta_t + \alpha\widehat{\nabla J(\theta_t)} \tag{10}$$

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 [10]. 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 asses action advantage and usually predicts the Q-value of action.

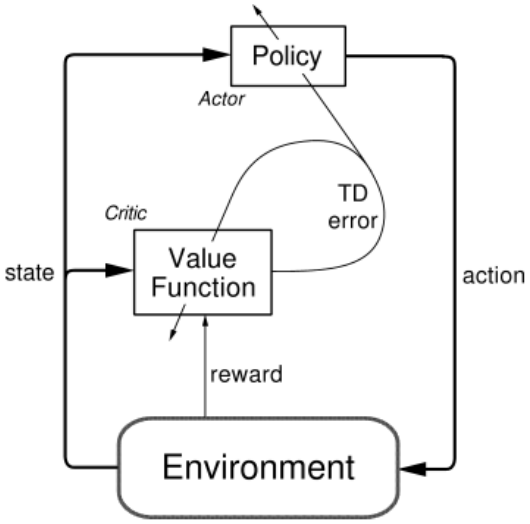


Figure 2: Actor-Critic framework

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:

$$\delta_t = r_t + \gamma Q^\omega(S_{t+1}, \mu^\theta(S_{t+1})) - Q^\omega(S_t, A_t) \tag{11}$$

$$\omega_{t+1} = \omega_t + \alpha\delta_t \nabla_\omega Q^\omega(S_t, A_t). \tag{12}$$

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.

$$\theta_{t+1} = \theta_t + \alpha \nabla_\theta \mu^\theta(S_t) \nabla_a Q^\omega(S_t, A_t)|_{a=\mu^\theta(S_t)}. \tag{13}$$

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

$$\frac{\partial Q^\omega(S_t, A_t)}{\partial \theta} = \frac{\partial Q^\omega(S_t, A_t)}{\partial A_t} \frac{\partial A_t}{\partial \theta}. \tag{14}$$

DDPG significantly outperforms its stochastic counterparts, especially in big continuous action spaces [8].

4.3 Wolpetinger policy

Actor-Critic methods and DDPG work well in continuous action spaces, but there is a lot of problems with large discrete action spaces, such as recommender systems or lidar planning. Wolpetinger policy is approach how to utilize DDPG in discrete action space [2]. Actor doesn't predict action directly, but it predicts so called proto-action $\tilde{A}_t$.

$$\tilde{A}_t = \mu^\theta(S_t). \tag{15}$$

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.

$$\mathcal{A}_{knn} = \underset{a \in \mathcal{A}}{argmin^N} |a - \tilde{A}_t|_2. \tag{16}$$

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

$$A_t = \underset{a \in \mathcal{A}_{knn}}{argmax} Q^\omega(S_t, a) \tag{17}$$

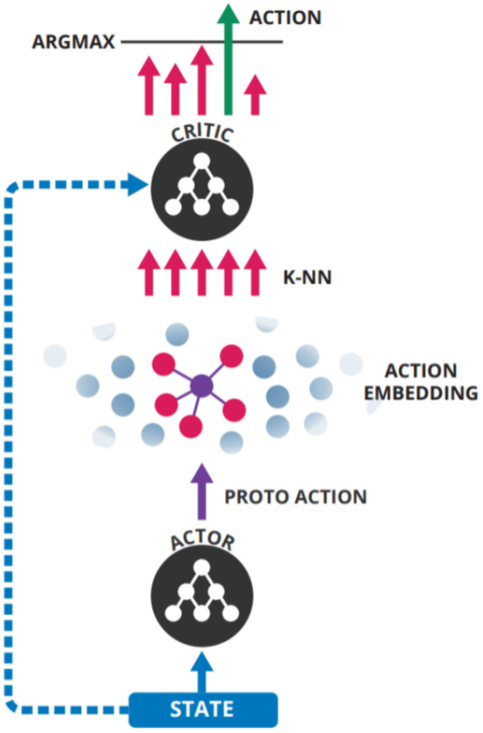


Figure 3: Wolpetinger policy illustration

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 optima. DDPG commonly use stochastic policy to slightly modify actors actions.

$$\hat{A}_t = \mu^{\theta}(S) + \mathcal{N}(0, \sigma^2) \tag{18}$$

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 [5].

$$\hat{\theta} = \theta + \mathcal{N}(0, \sigma^2) \tag{19}$$

where $\tilde{\theta}$ 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. Whereas parameter space noise has very unpredictable results. Thus it is necessary to use adaptive noise scaling.

$$d = |\hat{A}_t - \mu^\theta(S)|_2 \tag{20}$$

$$\sigma_{t+1} = \begin{cases} \kappa\sigma_t & \text{if } d \leq T \\ \frac{1}{\kappa}\sigma_t & \text{otherwise} \end{cases} \tag{21}$$

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

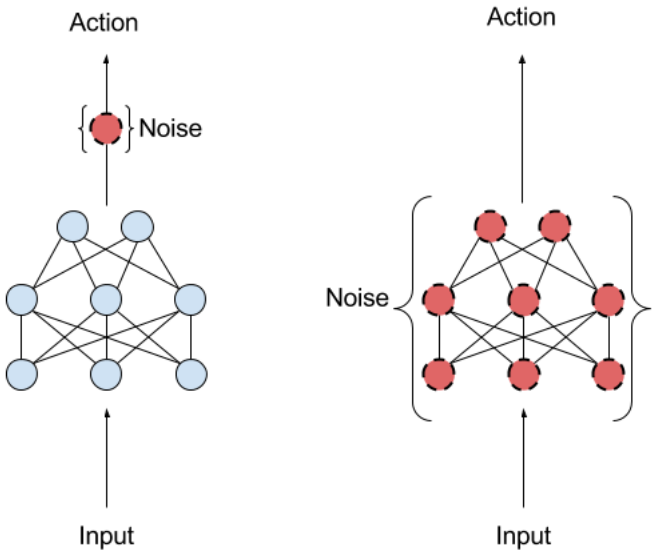


Figure 4: Action vs. parameter space noise

5 Experiment

Blah

6 Conclusion

BLAH BLAH BLAH

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] 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.

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

[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] 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.

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

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

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

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

[10] 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.

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

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

Appendix A CD Content

In Table 1 are listed names of all root directories on CD.

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

Table 1: CD Content

Appendix B List of abbreviations

In Table 2 are listed abbreviations used in this thesis.

Abbreviation	**Meaning**
API	application programming interface

Table 2: Lists of abbreviations

List of Figures