Goal Directed Obstacle Avoidance for Mobile Robots Using Deep Deterministic Policy Gradient

Robin Dietrich
Final Project - Report
CS533 - Intelligent Agents and Decision Making
dietriro@oregonstate.edu

Abstract

Path planning and obstacle avoidance for mobile robots, especially in unknown environments, is a crucial problem in robotics and as been an area of research for years. There are a variety of approaches including deterministic as well as randomized search methods like A and RRT*, respectively. Although some of these algorithms are complete and have low computational upper bounds, they all require a reliable map of the environment. This means, that a local map of the robots environment needs to be build at all times with e.g. SLAM.*

As an opposing solution to this problem, this report presents an approach of implicit local path planning/navigation for mobile robots by using a Deep Deterministic Policy Gradient algorithm to directly learn action (velocity) commands. This algorithm has proven to work on both high dimensional (image) and low dimensional input (1D data). Therefore this report describes and evaluates two different methods, one learns the algorithm based on a map image including all necessary information (robot pose, goal position, sensor data), the other one based on the plain sensor data. The evaluation shows that the latter approach seems to work significantly better and faster, although it contains less information, since it doesn't include information about the static map but the lower dimension of the data allow the network to learn faster.

1. Introduction

The area of path planning has been within the field of interest for researchers especially in robotics for years now, since it is essential to either navigate a mobile robot through an environment or fulfill a task with a robotic arm. There are numerous solutions to this problem, both deterministic [3] and randomized [1]. For these approaches the robot generally plans on its static global map, which is a gray-scale image that represents the environment as a discretized two dimensional array with values per pixel indicating whether the cell is occupied or free. In the case where there are dynamic obstacles in the world, this approach fails, since the global plan expects the area within the obstacle to be free. This leads to a situation as shown in Figure 1, where the robot needs to plan around the obstacle. Although standard path planning methods can be used for local path planning, nature and physics inspired algorithms like potential fields are rather used for this area, since they work better with dynamic obstacles, especially when they are moving. But all these algorithms need some preprocessing of the data in form of a discretized cost map that they can work on.

This is the reason why this report introduces a new approach of goal directed obstacle avoidance for mobile robots. Within the last years the success of deep learning for image classification, recognition or segmentation tasks has also influenced other fields like reinforcement learning. The development of the "Deep Q Network" (DQN) [6] led to numerous successes, mostly in learning to play video games solely based on the pixel data of the game as an input to the algorithm. The drawback of this method is, that it is just able to learn a mapping to a discretized action space and not to a continuous one. Therefore this method is not suitable for a lot of real world problems including the robotics domain and e.g. autonomous driving. A combination of Deep Q-Learning and the Deterministic Policy Gradient (DPG) [7] algorithm on the other hand make exactly this possible. In [4] the authors present their approach of Deep DPG (DDPG). The standard DPG approach is here extended by DQN advancements, such as a target network and a replay buffer for more consistent updates and learning. Due to its success, the A3C algorithm [5] was also considered by the author but since this algorithm requires multiple instantiations of the simulation environment, which is not possible in the ROS framework used for the simulation, it was not possible to use it for this particular problem.

This report implements a simulation environment as well as the DDPG algorithm to teach a mobile robot to avoid obstacles and drive towards a specified goal. For this purpose,

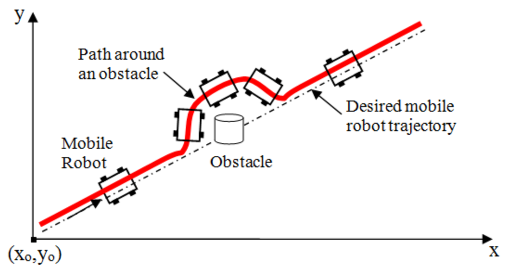


Figure 1: Visualization of the obstacle avoidance problem for mobile robots.

the problem is first formally defined as a Markov Decision Process (MDP) in Section 2. After that, the two implemented approaches are described in Section 3. For the first one, the learning is based on the plain sensor and positional data of the robot as well as the specified goal. In addition to that a second version is implemented, which uses a constructed map including the original static map, the robots position and orientation, its sensor data as well as the goal position. These approaches are then evaluated in Section 4, before Section 5 draws a conclusion of the implemented method Section 6 elaborates on the future work planned on this field.

2. MDP Problem Definition

In this section the problem of goal oriented obstacle avoidance for mobile robots is defined as an MDP to show that it can be solved using Q-Learning techniques. As part of that the state space S, the action space A and the reward function R are defined. The transition function is hidden in the simulation environment and will therefore be part of the learning process. A visualization of the problem domain can be seen in Figure 1. The image shows a robot that is trying to follow a global path which is however blocked by a dynamic obstacle. The robot therefore needs to plan its way around the obstacle. The data that is available to the robot for accomplishing this task is its sensor data, in this case a 360 degree laser scanner, its own current position as well as orientation, the position of the specified goal and the static global map. As already stated, there will be two different approaches presented in this report, one with and one without the static global map data. The state of the MDP for these two problems looks as follows:

$$S_{sensor} = \{sensordata[360], robotpose[3], goalposition[2]\} \quad (1)$$

$$S_{map} = \{globalmap[64,64]\{sensordata[360], robotpose[3], goalposition[2]\}\} \quad (2)$$

Algorithm 1 The Rules of the Reward Function

```
1: procedure GET_REWARD(map, ranges, pose, goal, action, d, a, v, r, t)
2:     $reward \leftarrow 0$
3:
4:     $reward \leftarrow reward + d/\text{euclid\_distance}(pose, goal)$
5:
6:     $ang\_error \leftarrow \text{angle\_dif}(\text{o}(pose), \text{o}(goal))$
7:     if $ang\_error > 0.5\pi$ then
8:         $ang\_error \leftarrow ang\_error * a$
9:     $reward \leftarrow reward + 2\pi - ang\_error$
10:
11:    if $action.v < 0$ then
12:        $reward \leftarrow reward + action * v$
13:
14:    $short\_ranges \leftarrow t - ranges[ranges < t]$
15:    $reward \leftarrow reward - \text{sum}(short\_ranges)$
       return reward
```

Where S_{sensor} specifies the state for the approach using plain sensor and positional input and S_{map} the one including the global map data. In the latter case, the sensor and positional information is integrated into the map which is a $64x64$ matrix. The state space is therefore a lot larger for the second approach. The action space of the two approach is defined as:

$$A = \{v, \omega\} \quad (3)$$

Where v defines the linear velocity in x direction and ω the angular velocity around the z axis. Both are continuous values and can be positive or negative.

The reward function is used for both approaches and is specified through the rules shown in Algorithm 1. The first reward is assigned in Line 4 of the procedure and adds a term to the reward that is based on the ratio of a specified parameter d and the euclidean distance between the current position of the robot and its goal. Next, a value that represents the difference in the angle between the current orientation of the robot and the desired orientation towards the goal. This value is positive for an absolute difference $<= 0.5\pi$ and negative for a difference $> 0.5\pi$. Here again the parameter a defines how large the reward is supposed to be in both positive and negative direction. The next term adds a negative reward to the overall reward in case the linear velocity of the robot is negative. This term is supposed to keep the robot from moving backwards too much, since this is generally not a desired behaviour with a mobile robot. The degree of negativity can be specified through the parameter v. The last reward that is added to the overall sum, depends on the ranges measured by the laser scanner. The negative sum of all ranges that are below a certain

```
1: Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$
2: Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$
3: Initialize replay buffer $R$
4:
5: for episode = 1, M do
6:     Initialize a random process $N$ for action exploration
7:     Receive initial observation state $s_1$
8:     for episode = 1, M do
9:         Select action $a_t = \mu(s_t|\theta^\mu) + N_t$ according to the current policy and exploration noise
10:        Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
11:        Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
12:        Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
13:        Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
14:        Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
15:        Update the actor policy using the sampled policy gradient:
16:
17:        $\delta_{\theta^\mu} \approx \frac{q}{N}\delta_a Q(s,a|\theta^Q)|_{s=s_i,a=\mu(s_i)}\delta_{\theta_\mu}\mu(s|\theta^\mu)|_{s_i}$
18:
19:        Update the target networks:
20:
21:        $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
22:        $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
```

Figure 2: DDPG Algorithm [4]

threshold, specified by the parameter t, is added to the total reward. This should keep the robot from driving into obstacles. This very simple reward specification leaves definitely room for improvement, which will be elaborated in more detail in the result section. Nevertheless this reward function definition led to very reasonable rewards as far as the author could tell during a test drive around the environment.

3. DDPG Obstacle Avoidance

This section describes the DDPG algorithm used for training the robot on the data as well as the simulation environment used to generate the data. First the DDPG algorithm is formally described and compared to similar approaches. After that the implementation of the simulation environment as well as the algorithm itself are elucidated.

3.1. DDPG Algorithm

The DDPG algorithm [ref] that was implemented for this project is shown in Figure 2. It uses an actor-critic network based on deterministic policy gradient to learn a policy. The *actor* $\mu(s|\theta^\mu)$ learns a policy by adjusting the parameters θ^μ. This gives a deterministic mapping from states to actions. The Q-function needed for the updates of the *actor* network is approximated by the *critic* $Q(s,a)$ using temporal-difference learning. To obtain consistent targets during the temporal-difference backups target Q-networks are defined for both, *actor* and *critic*. The parameters of these networks incorporate changes from the actual *actor* and *critic* networks very slowly to increase consistency in the targets. Furthermore a replay buffer is introduced, that trains the network off-policy with samples from the buffer. Using this learning method leads to a minimized correlation between the respective samples and increases the learning stability greatly. These improvements from recent DQN advances make it, according to the authors, possible to stabilize the learning even for complex tasks and continuous action spaces.

3.2. Implementation

In this part of the report the details of the implementation are explained covering the simulation environment and the structure that was chosen for the neural network layers.

3.2.1 Simulation Environment

To implement reinforcement learning in general a simulation environment is required. This simulation environment must offer the possibility to input an action and run the simulator for one time step. It is also necessary to have a method to reset the simulation environment to its initial state. Since none of the ”simple” ROS Simulators (stage/stdr) offers these options, the simulator stage-ros was extended with these methods. Part of the submitted code is a

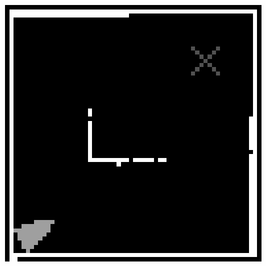
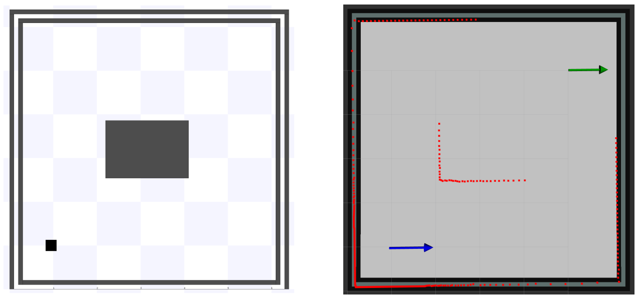


Figure 3: The representation of the current state created by the simulation environment as an input for the network.

ROS package called stage-ros that contains the source code of the standard stage-ros package, extended by a ros-service that enables another program to run the simulator with the current input for just one time step (100ms). On top of that a new *SimulationEnvironment* class was created for python. This class encapsulates the knowledge of the environment and offers methods to control it. It maintains a representation of the environment as class *Map* that contains information about the static obstacles (global map), the dynamic obstacles (laser scan), the current location (robot pose) and the goal position. This map is a discretized gray-scale representation of the environment and serves as input to the second approach of this project using CNNs. All obstacles, whether dynamic or not, are represented in this environment by the highest value, 255. As a result of that all cells that define a free space have the value 0. This representation makes input matrix for the network sparse, since in most environments the amount of free space is a lot higher than the amount of occupied cells. The position and orientation of the robot is included by drawing a triangle at the robots position, pointed in the direction of the robots orientation. All cells that are marked with the robots triangular position have a value of 80. The goal position is simply marked as a cross through the x and y coordinates, with all cells having a value of 160. This makes it easy for the network to distinguish the different kinds of information and it eliminates the need of different color channels, which would add a huge amount of complexity, since the input dimension would increase by an amount of three. An example of this created map can be seen in Figure 3. This image in form of a 2D matrix will be the input for the second approach presented in the next part of this section.

3.2.2 Neural Network Layer

For the two different approaches implemented during this project, two different neural networks were defined, each serving the purpose of the input shape. The first approach works, as already introduced, directly on the sensor input of the robot, its position/orientation and the goal position. This data input is 365 dimensional (360/3/2) and therefore



(a) ROS-Stage Simulator. (b) ROS-Rviz visualization.

Figure 4: The environment modeled for this report shown in ROS-stage simulator and rviz visualization tool.

a simple network composed of two fully-connected layers is used with a size of 500 and 1000, respectively.

As described in the previous subsection, the input for the second approach is a 2D matrix that includes a map like representation of the environment that includes all necessary information. An example for that can be seen in 3. To process this input, on top of the previously described two fully connected layers, two new convolutional layers are now introduced with a 32 and 64 filters of size 3x3. Max pooling layers as well as a dropout layer are included into the concolutional part of the network to decrease the chance of overfitting.

4. Experiments and Results

This part of the report evaluates the results obtained by training the presented networks. First, the used environment including simulator, robot and map are specified. Afterwards a comparison between the two introduced approaches is presented.

4.1. Simulation Environment

As already described in Section 3.2.2, the simulator used for learning the network is called Stage and is part of the ROS framework and was extended to fulfill the requirements of reinforcement learning. The robot used for this project is based on the model of the Turtlebot regarding size and motion model. This makes it easy to test the system on a real robot in a real world environment in the future and it also is a very common robot. The standard sensor of the robot was replaced by a 360 degree sensor, since it was just capable of delivering a small angular range of 90 degrees and with that the robot would not have seen anything behind or around it. The map that was developed for the following experiences can be seen in Figure 4. The image in 4a shows the simulation environment, so the world how it really (in the simulated world) is. The map consists of a simple squared room with an obstacle in the center. The obstacle is dynamic, that means it is not part of the static global

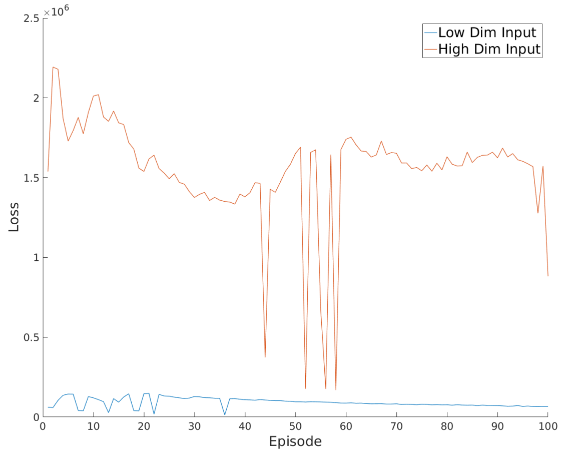


Figure 5: The resulting total loss of both networks per episode and 800 steps each episode.

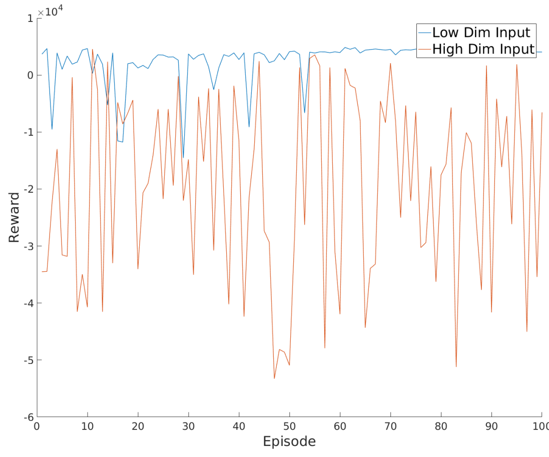


Figure 6: The resulting total reward of both networks per episode and 800 steps each episode.

map that the robot usually relies on. Therefore a global plan would initially go right through it and the robot has to plan locally around it. In Figure 4b the state of the world is displayed in the way the robot perceives it. The image displays the robots position in blue, its laser scan data in red and the goal in green. Here the sensor data detects the obstacle that is not part of the global map.

4.2. Low vs. High Dimensional Data DDPG

This part of the evaluation compares the results obtained for learning on low dimensional data (sensor, pose, goal) with the ones obtained from high dimensional data (map/matrix). The training parameters used for this experiment are shown in Table 1. The learning rate for the higher dimensional state space was chosen to be smaller than the one from the lower dimensional space, since it was generally harder to learn and it with higher values for the learning rates, the network got sometimes stuck in local minima. Due to insufficient time, both networks were just run for 100 episodes, 800 steps each. After 20 episodes the goal changed for both of them so that they learn to generalize. The goals in the 6.4x6.4m world are defined as follows: ([5.0, 5.0], [5.0, 0.5], [1.5, 2.5], [2.5, 5.0], [3.0, 2.0]). Where the origin of the map lies in the lower left corner of Figure 3 and 4. The robot starts at position [1.0, 1.0] with its orientation aligned with the x-axis as can be seen in Figure 4b as the blue arrow.

The results of the experiment are shown in Figures 5 and 6, where the first one displays the loss of the two networks and the second one the total reward after each period. Unfortunately these graphs don't reveal a real learning curve of the agent. In the case of the high dimensional input data, the loss decreases slightly over time, but the reward is not stabilizing at all over time. It is mostly negative and if so it is in highly negative regions. The robot very often gets stuck at a wall. This could be due to an insufficient amount of time for the learning agent. It might be the case that if it would run for thousands of episodes, the loss would eventually decrease over time and the reward would stabilize as well but since the learning of these 100 episodes already took hours of computational time, there was not enough time to let it run for much longer. A huge bottleneck of this computation is the ROS framework. Since it is a message based system, the program always has to wait until messages like sensor data arrive and this can take some time. After ten thousands of steps, this adds a huge amount to the computational demands. A look at the real behaviour of the robot after learning with the higher dimensional data didn't reveal any improvement over random behaviour.

Using the low dimensional data, the total reward collected increases a lot. The loss stabilizes and decrease slightly over time. The agent here learns very quickly to avoid running into walls. From the very beginning, the robot constantly tries to avoid running into any kind of obstacles. On the other hand it was very hard for the agent to learn a goal directed behaviour. This could be due to the fact, that it does not really identify the input goal as the

	γ	τ	LR actor	LR critic
Low Dim	0.99	0.001	0.0005	0.005
High Dim	0.99	0.001	0.0001	0.001

Table 1: The training parameters for the two different approaches implemented.

goal during the learning process. Maybe a longer amount of learning time would solve this as well. The comparison between these two approaches shows, how much harder it is to learn a network if the input size increases. The high dimensional network might need just more time or also a refinement in the network architecture. The author tried to add another convolutional layer but without significant improvements.

Another problem that the author experienced, is that the agent has a huge problem with learning to turn in place, e.g. if the goal is in a completely different direction from the very beginning. It almost always has some kind of positive or negative velocity in both, linear and angular direction. This makes it hard to turn in place or drive just straight. One approach the author took to counter counter this behaviour was to set either the linear or angular velocity to zero. The probability of setting it to zero decreased over time. Unfortunately, this approach did not really lead to an improvement in the robots behaviour, but it also didn't make it worse. A problem the authors in [4] already mention by testing their algorithm on the racing simulation TORCS is, that the algorithm is not really stable. In their results the network not always converged. According to them, sometimes it found a solution and sometimes not.

5. Conclusion

This report presented two approaches to learn goal directed obstacle avoidance for a mobile robot. The DDPG algorithm was implemented to train the two networks. The results have shown, that none of the two proposed approaches were able to really learn to avoid all obstacles and walls and also drive towards the goal. The approach using low dimensional input data (sensor, robot pose, goal) though worked significantly better, or faster. Due to insufficient amount of time the author was not able to learn the networks for much longer, although especially the high dimensional data would probably have performed a lot better after more learning time. Overall the experiments revealed, that it is possible to learn this kind of behaviour, since the robot avoided running into obstacles almost completely using the low dimensional network. But on the other hand it is hard to learn, since it was not able to always reach the goal with a high precision.

6. Future Work

Since the time was short for this project and it took a long time to find and extend the right simulator, time was the biggest problem in the implementation and evaluation of the proposed approaches. This is why the first point of future work is to run especially the high dimensional network for a couple of thousand episodes at least and see if it eventually converges. Another approach will be to refine the whole network structure, change the number and size of the layers. Another extension could be the use of different environments with and without obstacles to see the learning behaviour of the agent.

7. Software/Libraries

The code for this project is completely written for ROS and can be just used and executed with a full installation of ROS, Keras and Tensorflow. The DDPG algorithm was implemented using Keras and is based on an approach to learn the computer game TORCS [2]. The stage-ros package was extended based on its open source code but is not fully implemented by me.

References

[1] A. Bry and N. Roy. Rapidly-exploring random belief trees for motion planning under uncertainty. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 723–730. IEEE, 2011. 00168.

[2] B. Lau. TORCS DDPG. https://yanpanlau.github.io/2016/10/11/Torcs-Keras.html.

[3] M. Likhachev, G. J. Gordon, and S. Thrun. ARA*: Anytime A* with provable bounds on sub-optimality. In *NIPS*, pages 767–774, 2003. 00000.

[4] 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:1509.02971 [cs, stat]*, Sept. 2015. 00203 arXiv: 1509.02971 bibtex: lillicrap_continuous_2015.

[5] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. *arXiv:1602.01783 [cs]*, Feb. 2016. 00183 arXiv: 1602.01783 bibtex: mnih_asynchronous_2016.

[6] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb. 2015. 01174.

[7] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. Deterministic policy gradient algorithms. In *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pages 387–395, 2014. 00088.