23rd *Computer Vision Winter Workshop*
Zuzana Kúkelová and Júlia Škovierová (eds.)
Český Krumlov, Czech Republic, February 5–7, 2018

Map-less goal-driven navigation based on reinforcement learning

Matej Dobrevski, Danijel Skočaj
Faculty of Computer and Information Science, University of Ljubljana
Večna pot 113, 1000 Ljubljana
matej.dobrevski@fri.uni-lj.si

Abstract. *Mobile robots that operate in real-world environments need to be able to safely navigate their surroundings. This problem is especially significant for robots operating in dynamic or new environments as the map of the environment might be unreliable or unavailable. We present an approach for learning map-less goal-driven navigation using an actor-critic deep reinforcement learning (RL) approach. The behavioral policy of the robot is represented as a deep neural network. The agent is trained in a simulated and highly randomized obstacle course. We evaluate the learned policy on a real robot.*

1. Introduction

Intelligent machines are expected to perform complex tasks in unstructured environments. Regardless of the specific assignment at hand such machines would need to be watchful for the safety and potential damages of the environment, itself, as well as any humans or agents in the vicinity. If the machine is mobile, the minimum operational requirement is the ability of planning its movement and avoiding obstacles.

If the environment is static then the usual approach is to build a map of the environment and use a path-planning algorithm in conjunction with a localization algorithm to safely navigate the working space. However, for some robotic scenarios (e.g. a robot operating in crowded environments, exposition areas where the exponents are changed) building a map of the environment can be unfeasible as we might want the robot to be operational momentarily, or the environment might change at a pace that makes maintaining an accurate map impractical.

For a robot operating in such environments a more reactive behavioral policy, one which does not rely on a map, can be more suitable. Motivated by the recent advances in reinforcement learning (RL), as well as the methodological applicability of RL to an agent which sequentially perceives an environment and produces an action, we develop a map-less goal-driven navigation policy. Our policy takes as input an observation of the world, represented by a range-scan of the environment, the angle and distance to the goal position, and outputs a movement command for the robot. We model the problem of navigating the environment as a Markov Decision Process [11] (MDP) and optimize a behavioral policy within the framework of reinforcement

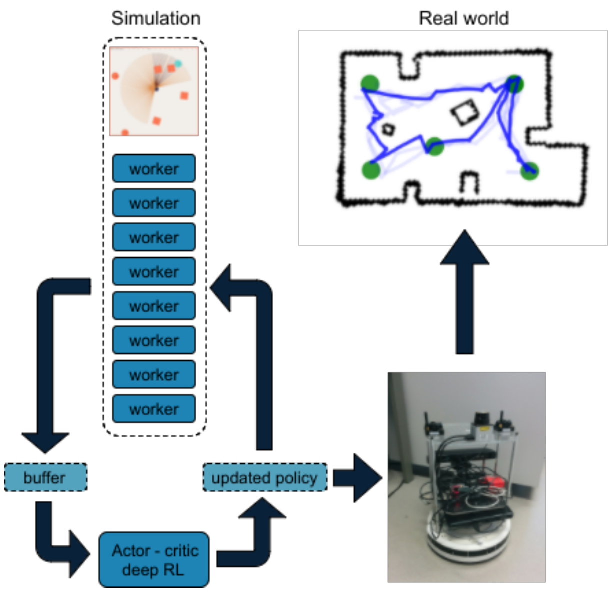


Figure 1: A behavioral policy was learned in simulation by training with deep reinforcement learning. The learned policy, modeled by a deep neural network, takes as input a range-scan and the relative position of the goal. It outputs a turning angle or a straight movement command. The policy is then implemented on a real robot.

learning, specifically the state-of-the-art Advantage Actor-Critic [10] (A2C) method.

Running experiments and training is expensive when working with real robots. Physical components suffer from wear and tear, resetting the system takes time and effort, batteries need charging. For the case of training mobile robots for obstacle avoidance and path-planning there is the added problem of gathering negative samples, as collisions can be critical and are always costly. Furthermore, RL algorithms typically demand large amounts of data to converge. Generating a large amount of data is possible using a fleet of robots (which is especially impracticable when dealing with mobile robots), learning in simulation using a very realistic simulator, or having a less realistic simulation, but vigorously randomizing parameters of the simulation during learning to achieve regularizing effects. We take the last approach.

We implement the learned policy on a custom built Turtlebot[1] robot equipped with a laser range-scanner and evaluate our map-less approach on an obstacle course. Our results are compared with the performance of the standard Turtlebot navigation package which uses an Adaptive Monte Carlo Localization [3] (AMCL) algorithm together with a Dynamic Window Approach [4] (DWA) local planner and a global planner based on the Dijkstra algorithm. We evaluate both approaches and show that our approach is more robust as the map for the navigation stack degrades.

In 2 we will present the related work, then we will present our model learning framework in 3, and in Section 4 we will present our results, the evaluation of the algorithm on a real robot and a discussion of the results. In Section 5 we present our conclusions.

2. Related work

Model-free methods for optimizing MDPs make it possible to find an optimal policy without explicitly modeling the dynamics of the underlying MDP. Q-learning [19] and methods based on Policy Gradients [15] are among the most significant such methods.

Recently, incredible results were presented by Mnih et al. [9] by their deep neural network implementation of the Q-learning algorithm, where they achieved a human level performance on a number of Atari games. Wang et al. [18] applied the same (DQN) optimization algorithm but designed a novel network architecture, separating the state-values, and the action-advantages named Dueling DQN. This architecture beat the state of the art on the Atari benchmark. Then van Hasselt et al. [17] presented a method named Double DQN which uses a second, *target* Q-network, and so circumventing the maximization bias in Q-learning and achieving better results in a number of domains.

While policy gradients have been around for quite some time, Mnih at al. developed the Asynchronous Advantageous Actor Critic (A3C) method [10] for a neural network implementation of a policy function that outperforms the standard DQN, and on some games the Double DQN and in numerous experiments demonstrated the superiority of policy gradient methods for some classes of problems. The training of these deep models is made practically viable by employing methods of experience replay [8] like Prioritized Experience Replay developed by Schaul et al. [13]. Employing experience replay significantly speeds up the convergence of the network by breaking down the high correlation between successive samples.

Applications of RL methods to obstacle avoidance have been done in various ways and settings. Some have used tabular methods and discretized the state space to make those method viable, while others have used neural networks as function approximators to represent the Q-function, the state-value function or the policy.

Discretized state space is usually done by approximating the working environment with a grid. While limiting the applicability, this has the advantage of computational simplicity. Adbel et al. [1] implement a tabular form of Q-learning for obstacle avoidance and assume a fully observable environment. A fully observable environment is often not available to a robot, and the tabular Q-learning can only be applied up to a certain environment-complexity, as the method suffers from the curse of dimensionality. Konar et al. developed Improved Q-learning (IQL) [7] as an extension of Extended Q-learning [2] and applied it to path-planing in an environment represented by a grid. The IQL algorithm presupposes a knowledge of the distance to the next-state (observation) which is not always available. Wen et al. [20] used a tabular implementation of the Q-learning algorithm merged with EKF-SLAM [14] for an obstacle avoidance system on a NAO robot. They use clustering methods to pre-

[1]`http://www.turtlebot.com/`

process the laser-scan data, which restricts the possible configurations of the obstacles.

Neural network function approximation fueled by the recent advancements in computation and deep networks has made possible the circumvention of some of the negative effects of drastic discretization—using powerful networks to approximate the state-value function, Q-function or directly the policy. Gupta at el. [6] implemented an end to end system on learning simultaneous mapping and planning for robotic navigation by applying a value iteration method. This method in general converges slower than either Q-learning or policy-gradient methods. A variant of Q-learning (fitted Q iteration) was applied to an in-simulation learning system [12], which through high randomization can to a certain degree of success, be directly applied to the target domain.

The work most closely related to ours is Lei at al. [16]. They also train a navigation policy for the Turtlebot using a Deep RL algorithm. However, they use the deterministic policy gradients approach, while we use an actor-critic approach. Secondly, they simulate the robot in a single environment randomizing only the goal location, while we randomize the whole environment on each run. As a result the trajectories we generate are not as smooth, but they are easier to transfer to a real-robot as the policy is not tightly coupled with the dynamics parameters of the robot. Our simulation does not rely on a 3D physics engine and consequently we are able to train a policy much faster.

3. Method

We model the problem of obstacle avoidance as a Markov Decision Problem (MDP), which is a common approach for transferring problems to the reinforcement learning domain.

3.1. Formal description

An MDP is defined by a set of states S, a set of actions A, transition probabilities $t(s,a) : S \times A \mapsto S$, a reward at each time-step R_k and a discount factor for the future rewards $\gamma \in [0,1]$. The agent is randomly initialized in the environment, after which it observes the state s_0 and performs an action a_0 according to a policy $\pi(s) : S \mapsto A$. This takes the agent to the state s_1 and it receives a reward of R_1. This process continues iteratively until a terminal state is reached, which in our implementation is either a state where the robot has reached the goal-state, or it has reached a state where it is in collision with an obstacle. Our goal becomes finding an optimal policy π^* such that we would maximize the cumulative discounted future reward $\sum_{k=1}^{k_{final}} \gamma^{k-1} R_k$.

In our approach a state is represented by a vector $s_k = \begin{bmatrix} l_1 & \dots & l_{30} & \alpha & d \end{bmatrix}^T$ where l_1 to l_{30} are laser-scan measurement of the environment around the robot, α is the angle to the goal and d is the distance to the goal. The action a_k is one of 7 available turning angles. The *value function* $V(s_k) = E[R_k + \gamma^k R_{k+1} + \dots | s_k]$ tell us how much reward we can expect to get when we are in a certain state, and the *quality function* or Q-function $Q(s_k, a_k) = E[R_k + \gamma^k R_{k+1} + \dots | s_k, a_k]$ tells us how much reward we can expect to get after taking a certain action in a certain state. Consequently, we can define an advantage function $A(s_k, a_k) = Q(s_k, a_k) - V(s_k)$ which tells us how much more reward we can expect for taking a certain action in a certain state (then the reward we would get by taking the average action).

3.2. Advantage Actor-Critic Approach

A2C is a model-free method which uses the advantage function in optimizing the policy. The advantage function is calculated with the critic which is trained to approximate the value function.

The actor network is the behavioral policy which is represented by a neural network which outputs the probability that each action is the best action in the current state:

$$\pi(a_k | s_k; \theta_a) = Pr(a = a_k | s = s_k, \theta = \theta_a) \quad (1)$$

where the vector θ_a are the weights of the neural network. Our goal becomes learning the weights θ_a on the basis of a performance measure so that our policy will approach the optimal policy – maximizing the cumulative discounted reward. The critic network approximates the expected future reward, given a state:

$$V(s_k; \theta_c) = E[R_k + \gamma R_{k+1} + \gamma^2 R_{k+2} \dots | s_k, \theta_c] \quad (2)$$

where θ_c are the weights of the critic network. It should be noted that the actor and critics network share the same weights on the lower layers.

The loss for the critic network is the mean-squared error:

$$Loss_{critic} = (V(s_k; \theta_c) - \sum_{i=0}^{i_{max}} \gamma^i R_i)^2 \quad (3)$$

The loss for the actor network, according to the Policy-Gradient theorem [15] is:

$$Loss_{actor} = log\pi(a_k|s_k;\theta_a)A(s_k,a_k;\theta_c) \quad (4)$$

While optimizing the actor and critic losses it is common that the agent will quickly start to converge to a sub optimal policy. To encourage exploration an entropy term is added to the loss function which prevents premature convergence of the learning method.

$$Loss_{ent.} = \pi(a_k|s_k;\theta_a)log\pi(a_k|s_k;\theta_a) \quad (5)$$

Therefore, the complete loss function we used is:

$$Loss = Loss_{actor} + v_c Loss_{critic} - v_e Loss_{ent.} \quad (6)$$

Where v_c and v_e are hyperparameters for the loss of the critic and the entropy respectively.

3.3. Implementation of the policy network

The actor-critic network is modeled with a deep neural network represented in Figure 2. The actor and the critic share the lower layers of the network. The network has a categorical output for the actor, and a linear output for the critic.

The architecture of the network is different to the one used in [10]. We replaced the convolutional layers with fully connected layers. The actor and critic streams of our network also split sooner, only the first three layers and shared, with each separate stream having three additional layers instead of the one.

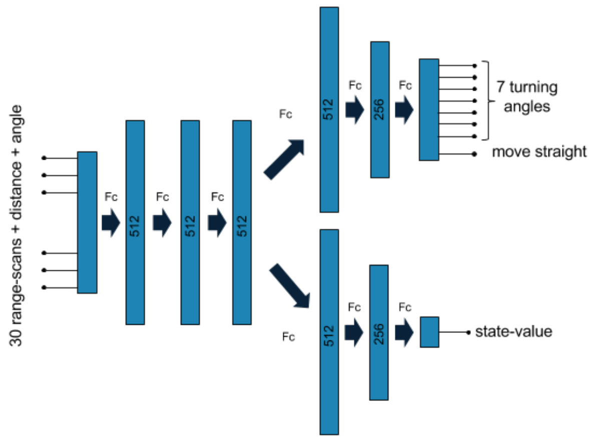


Figure 2: The design of the policy network with the state-value approximation. The network is split into two streams, one for the actor with a categorical output and one with a linear output for the critic. Fc stands for fully-connected layer.

The input to the network is a vector representing the state of the world. The state vector is generated by concatenating 30 range-scan measurements with the angle and distance to the goal. In simulation each range-scan measurement is generated by taking the minimum of 4 neighboring laser-scans. The complete laser-scan captures measurements in the range of $\pm 120°$ with respect to the forward direction of the robot. The actor outputs one of eight possible actions. The first seven actions are interpreted as a turning angle while the last output is interpreted as movement in a straight line.

The first three layers of the network are fully connected layers with 512 neurons each and a Rectified Linear Unit (ReLU) activation function. The network is then split into two streams, one for the actor and one for the critic. The actor stream has a further layer with 512 neurons, then a layer with 256 neurons and then the output layer with eight outputs. The critic stream contains one layer with 512 neurons, than a layer with 256 neurons and an output layer with a single linear output.

We used the RMSPropOptimizer as it was found to have the lowest variance among the tested optimizers during training. We used a learning rate of 0.007, decay of 0.99 and ϵ was set to 0.00005. The learning rate was decreased during the training as this also proved to stabilize the learning.

The neural network and the A2C are implemented in Tensorflow[2].

3.4. Training in simulation

Generating samples from a single simulated robot produces highly correlated samples. The use of such data is detrimental for the learning for RL methods based on policy gradients. Therefore, eight uncorrelated simulators were run in parallel to generate experience. Our simulator is built on top of the physics simulator Chipmunk[3]. Four visualizations of the simulator are represented in Figure 3.

All eight workers are run for 100 time-steps after which the produced samples were gathered in a common buffer from which the actor-critic network was updated. At the start of each episode the obstacles' and goal's positions are randomized. After a number of steps the agent encounters a terminal state or it performs 100 cycles of observing and acting after which the episode ends. A terminal state means that the agent has collided with an obstacle or it has

[2] `https://www.tensorflow.org/`.

[3] The Chipmunk physics engine `https://chipmunk-physics.net/`

reached a goal. Then the actor-critic network is updated with the samples collected in the buffer, and a new episode begins.

During training each action is selected by sampling from the outputs according to the predicted probability. During test time an action is selected as the action with the maximum output.

The reward function was designed such that the robot receives a reward of 100 for reaching the goal which is done through checking for a distance threshold, -50 for colliding with an obstacle and each time-step it additionally receives a reward proportional to change of distance to the goal with values in the range of ±1.5. Even without this intermediate reward the robot learns a useful policy, however, we found that this reward speeds up the training about twice.

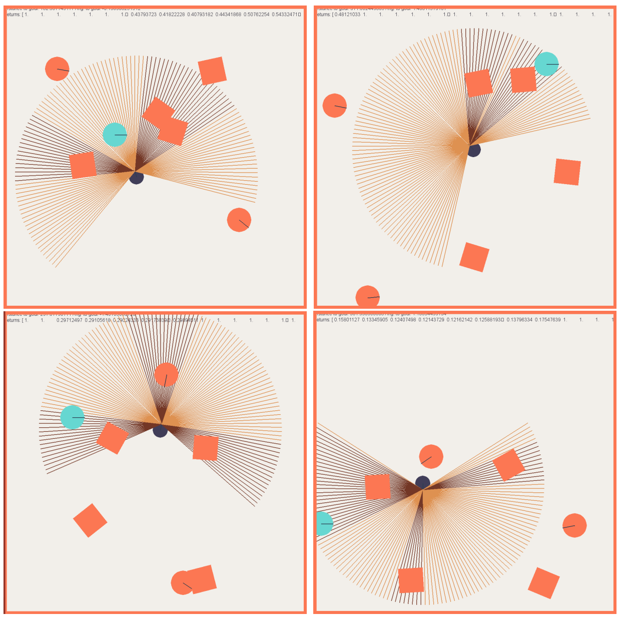


Figure 3: Four visualizations of the simulator used for training. The orange objects represent obstacles and the blue circle represents the goal position. The robot is represented with purple color.

4. Evaluation

Here we present the results for learning the policy in simulation, then the set-up that was used for the real-world experiments and at the end the evaluation of the real-world experiments.

4.1. Learning

The policy was trained for total of 10M time-steps, 1.125M time-steps per simulation. In other words each of the 8 simulators running in parallel played out 11250 episodes of 100 time-steps. We ran the simulation and trained the network three times. The progression of learning, represented as the running average of the reward in 100 episodes vs the time-steps is visualized in Figure 4.

Each of the eight simulations was run on a separate core of the Intel i7-6700 CPU processor, while the network was trained on a GeForce GTX 1070 graphics card. The total training time was about 2 hours.

As can be seen in Figure 4 the agent started to average a positive reward after about 0.5M time-steps. It reached a small plateau at about 1M steps (possibly due to learning a sub-optimal greedy strategy), after which the average reward quickly rises till about 3M time-steps, after which the learning slows down.

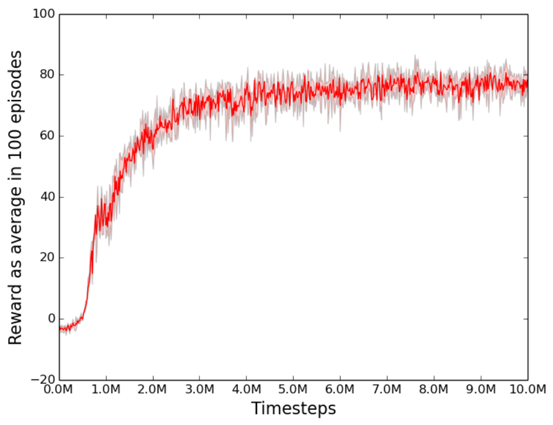


Figure 4: Progression of learning. Reward vs time-steps.

4.2. Experimental setup

The trained policy was implemented on a custom Turtlebot represented in Figure 5. The Turtlebot was equipped with a scanning laser range finder Hokuyo URG-04LX[4] and two ultrasonic localization beacons from the Marvelmind Indoor Navigation System[5] for localization.

The range-scanner on the robot generates 1024 range-scans in a radius of $\pm120°$. These range-scans were grouped into 30 bins of consecutive range-scans and for each bin we took the minimum reading as

[4]Hokuyo laser range-scanner `https://www.hokuyo-aut.jp/search/single.php?serial=165`

[5]Marvelmind indoor navigation system `https://marvelmind.com/shop/starter-set-hw-v4-9-plastic-housing/`

the value for that bin. The position of the robot in the world was calculated by taking the average x and y measurement from the two symmetrically placed beacons on top of the robot. Knowing its orientation the distance and angle to the goal position is calculated and together with the 30 range-scans an observation of the world is generated. The state was then fed through the network and the output is transformed into a simple movement command for the robot which is sent as a simple movement command through the exposed ROS[6] interface. The evaluation of the network as well as all the other processing was done on the on-board laptop.

The obstacle course which was used for evaluating the policy on the Turtlebot can be seen in Figure 6.

For the Turtlebot navigation stack a map of the course was constructed using OpenSlam's Gmapping[7] which is a Rao-Blackwellized [5] particle filter for learning grid maps from a range-scanner. This map was given to the navigation stack for all the experiments, although it was set to update this map while navigating the obstacle course.

In the first stage we evaluate the algorithms on the original obstacle course. In the second stage two of the obstacles were displaced and in the third stage the course was severely altered. In each stage the robot was sequentially given six navigation goals and each algorithm navigated the robot through the goals three times. We plot the trajectory that the robot executed as well as the cumulative distance traveled. The resulting trajectories and the cumulative distances are represented in Figure 7. From each of the three runs one trajectory is bolded for each navigation algorithm. The cumulative distances are represented as the mean of the three trajectories.

4.3. Experimental results

The first obstacle course with the traversed trajectories is represented in Figure 7(a). Both our learned policy and the navigation stack had no problems navigating to all six goals. We can see that our algorithm produces a more “choppy” trajectory which is also a slightly shorter trajectory, but both trajectories are of comparable length as can be seen in 7(d).

In the second obstacle course, visualized in Figure 7(b) two of the obstacles were moved. The navigation stack, having enough ground-truth from the original map, quickly updated the map and had no problem navigating the changed obstacle course. Our algorithm also did not have problems navigating this obstacle course. The traveled distances are are of comparable length as is be seen in Figure 7(e).

In the last obstacle course the obstacles were severely displaced, as well as some of the walls. The Turtlebot navigation stack was run five times in this course. Out of these it got lost two times, and managed to find all goals three times. After getting lost the navigation stack performed recovery behaviors but ultimately did not manage to localize itself. In

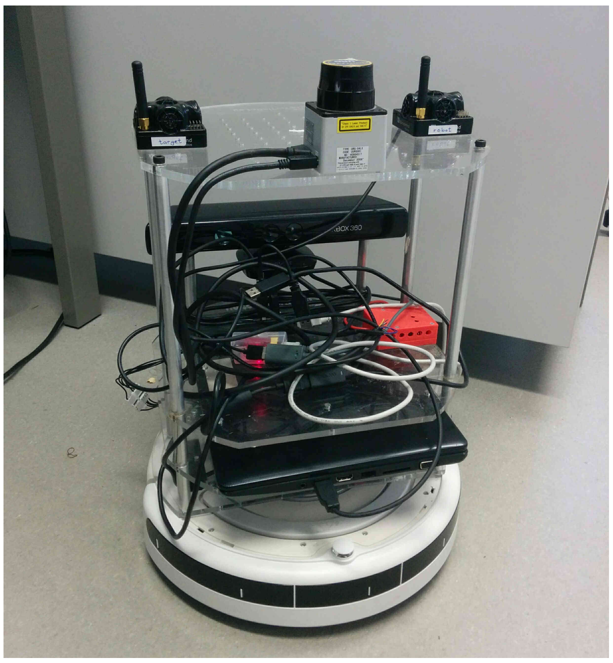


Figure 5: The custom built Turtlebot. Equipped with a laser range-scanner and two localization beacons.

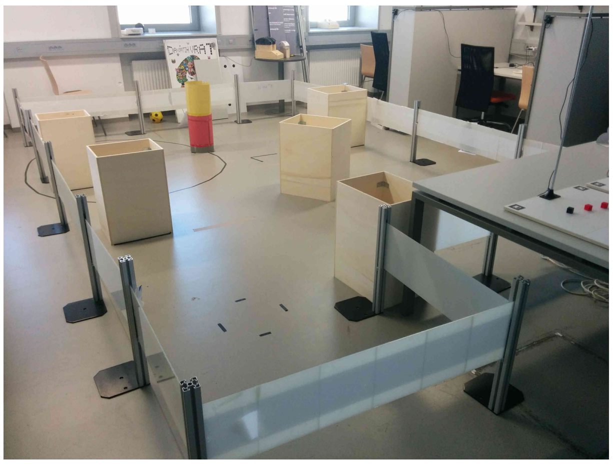
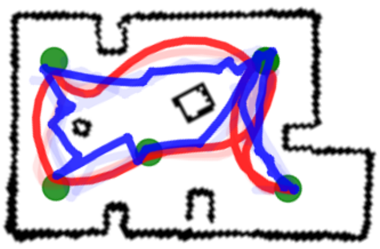


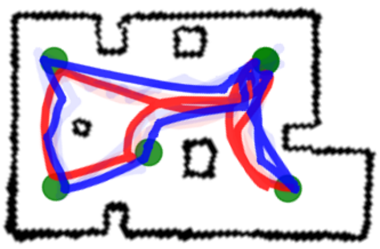
Figure 6: The obstacle course used for evaluating the navigation algorithms.

[6]The Robot Operating System `http://www.ros.org/`

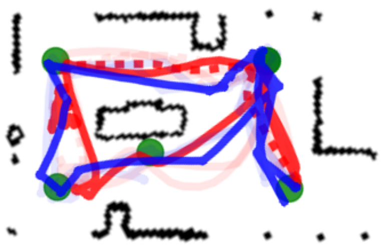
[7]OpenSlam's Gmapping `https://www.openslam.org/gmapping.html`



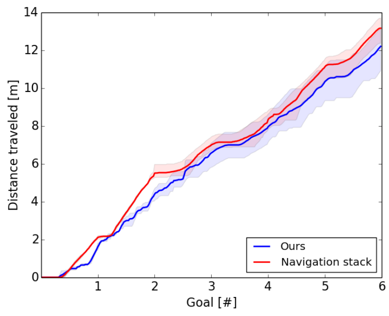
(a) Stage1: original polygon.



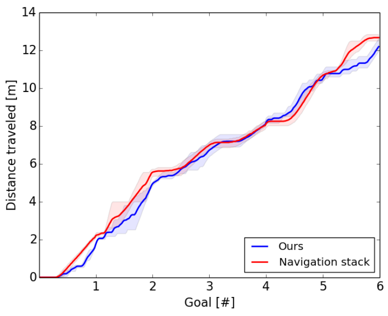
(b) Stage2: two displacements.



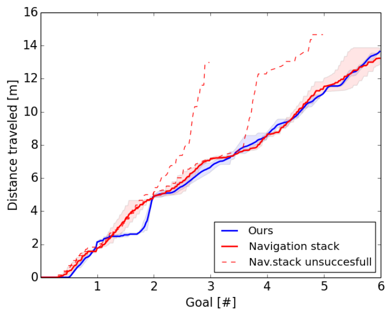
(c) Stage3: severely altered.



(d) Distance traveled for Stage 1.



(e) Distance traveled for Stage 2.



(f) Distance traveled for Stage 3.

Figure 7: Evaluation of the executed trajectories. Stage 1 is represented in (a). The second stage, where two of the obstacles were displaced is represented in (b). The third stage, where the obstacle course was severely altered is represented in (c). In (a),(b) and (c) the blue line represents a trajectory executed by our navigation policy and the red one the trajectories by the navigation stack. (d), (e) and (f) present the cumulative distance traveled for (a),(b) and (c) respectively.

the cases when it managed to find all goals the total distances traveled were again of comparable length. In the two cases when it got lost, the traveled distance was greatly increased by the recovery behaviors. As it can be seen in Figure 7(c) and (f) the performance of our method does not degrade as the obstacle course is changed.

5. Conclusion

We presented a framework for learning a map-less goal-driven navigation policy which at each time-step tries to get closer to the goal, while avoiding collisions with obstacles. To achieve this we used the state-of-the-art A2C reinforcement learning algorithm.

The training of this policy is possible because we train it completely in simulation and in this paper we showed that such a policy can be directly transferred to a real robot without any domain adaptation. We also showed that our method is robust and consistent in performance. Because our method does not rely on a map of the environment it is applicable in scenarios when many other methods are not.

However, having a map of the environment is of great utility, especially in complex environments, and we do not claim to replace such methods, but present an alternative approach which is feasible in scenarios when other methods are not.

Initial experiments suggest that we can further reduce the reliance on environmental measurements by excluding the distance to the target from the state-vector. This can further extend the applicability to scenarios when only the bearing to the target location is known.

We also believe that we can improve our method by addressing the partial observability of the problem at hand. We plan to utilize the history of the observations the robot has performed, and will do this by incorporating LSTM units in the policy network.

Acknowledgement This work was in part supported by the ARRS research programme P2-0214.

References

[1] M. Abdel, K. Jaradat, M. Al-rousan, and L. Quadan. Reinforcement based mobile robot navigation in dynamic environment. *Robotics and Computer Integrated Manufacturing*, 27(1):135–149, 2011. 2

[2] P. K. Das, S. C. Mandhata, H. S. Behera, and S. N. Patro. An Improved Q-learning Algorithm for Path-Planning of a Mobile Robot. *International Journal of Computer Applications*, 51(9):40–46, 2012. 2

[3] D. Fox. Adapting the sample size in particle filters through kld-sampling. *The International Journal of Robotics Research*, 22(12):985–1003, 2003. 2

[4] D. Fox, W. Burgard, and S. Thrun. The dynamic window approach to collision avoidance. *IEEE Robotics & Automation Magazine*, 4(1):23–33, 1997. 2

[5] C. S. Giorgio Grisetti and W. Burgard. Improved Techniques for Grid Mapping with Rao-Blackwellized Particle Filters. *IEEE Transactions on Robotics*, 23:34–46, 2007. 6

[6] S. Gupta, J. Davidson, S. Levine, R. Sukthankar, and J. Malik. Cognitive Mapping and Planning for Visual Navigation. *arXiv:1702.03920*, 2017. 3

[7] A. Konar, I. G. Chakraborty, S. J. Singh, L. C. Jain, and A. K. Nagar. A Deterministic Improved Q-Learning for Path Planning of a Mobile Robot. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 43(5):1141–1153, 2013. 2

[8] L.-j. Lin. Self-Improving Reactive Agents Based On Reinforcement Learning, Planning and Teaching. *Machine Learning*, 321:293–321, 1992. 2

[9] 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, 2015. 2

[10] V. Mnih, M. Mirza, A. Graves, T. Harley, T. P. Lillicrap, D. Silver, K. Kavukcuoglu, and G. Deepmind. Asynchronous Methods for Deep Reinforcement Learning. *International Conference on Machine Learning*, 48, 2016. 2, 4

[11] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1994. 1

[12] F. Sadeghi and S. Levine. (CAD)^{2}RL: Real Single-Image Flight without a Single Real Image. *arXiv:1611.04201*, 2016. 3

[13] T. Schaul, J. Quan, I. Antonoglou, D. Silver, and G. Deepmind. Prioritized experience replay. *International Conference on Learning Representations*, 2016. 2

[14] R. C. Smith and P. Cheeseman. On the representation and estimation of spatial uncertainly. *Int. J. Rob. Res.*, 5(4):56–68, Dec. 1986. 2

[15] R. S. Sutton, D. McAllester, S. Singh, and Y. 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. 2, 4

[16] L. Tai, G. Paolo, and M. Liu. Virtual-to-real deep reinforcement learning: Continuous control of mobile robots for mapless navigation. *CoRR*, abs/1703.00420, 2017. 3

[17] H. van Hasselt, A. Guez, and D. Silver. Deep Reinforcement Learning with Double Q-learning. *AAAI Publications, Thirtieth AAAI Conference on Artificial Intelligence*, 2016. 2

[18] Z. Wang, T. Schaul, M. Hessel, H. V. Hasselt, M. Lanctot, N. D. Freitas, and G. Deepmind. Dueling Network Architectures for Deep Reinforcement Learning. *Journal of Machine Learning Research*, 48(9), 2016. 2

[19] C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3-4):279–292, 1992. 2

[20] S. Wen, X. Chen, C. Ma, H. K. Lam, and S. Hua. The Q-learning obstacle avoidance algorithm based on EKF-SLAM for NAO autonomous walking under unknown environments. *Robotics and Autonomous Systems*, 72:29–36, 2015. 2