Robotics Project WS16/17 - Scientific Report

Kevin Kepp (366263) Philipp Braunhart (361923)

Abstract— **In this project we applied a Deep Q-Network (DQN) to a partially-observable search problem in a map-like environment. The agent is trained to extract the visual structure of the environment to indicate useful search directions towards locating a given target. A reward signal is only given when the agent successfully locates the target. We introduce a novel way of providing the agent with a memory over its past actions. This action history is provided as an input to the neural network alongside the visual observations. Furthermore, we implement a simple version of prioritized replay, which we call proportional replay, in order to increase the influence of rewarding experiences the agent gathered while interacting with the environment. We show that combining action history and proportional replay the agent is able to overcome the problem of perceptual aliasing. Experiments with training on randomly chosen environments show that the algorithm does not generalize well to more complex problems. Further work is necessary to investigate whether different neural networks architectures or a different learning algorithm are necessary to overcome this limitation.**

I. Introduction

Methods of deep reinforcement learning recently showed above human-level performance in a wide variety of games such as in the Atari domain [1, 2], in Go [3] and Poker [4]. In this project we applied the technique behind one of these successes to a new kind of problem.

A. Problem statement

We imagine a drone in the real world that flies in constant height and perceives the world using a camera pointing downwards resulting in a bird's-eye view. The objective of this drone is to locate a given target in the world. For example, after natural disasters it can be used to quickly locate broken infrastructure such as damaged bridges or train tracks. To reach this objective an agent is trained in a simulated environment to develop a behavior policy to find the given target based on its observations. While training, the agent receives a reward if it locates the target. Furthermore, the learning algorithm should not be specific to the exact environment the agent is navigating or the specific properties of the target. Thus, to develop an effective policy only based on the reward signal the agent has to leverage the underlying structure of its visual input and extract useful search directions.

For this project we consider a simpler version of the drone problem. The map-like environment the agent is navigating is represented by a 70 by 70 pixel gray-scale image as illustrated in figure 1. The agent observes a 7 by 7 pixel section of this environment based on its current position. Every time step it is asked to choose one out of four possible directions - up, down, left or right - and is then moved by one pixel accordingly. However, the learning procedure we apply to solve this problem should in principle be capable of handling the original problem of a drone in the real world.

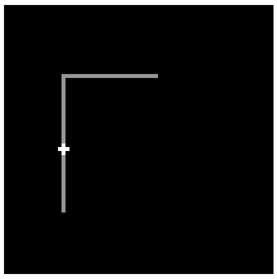


Fig. 1: Gray-scale image representing the simulated environment the agent navigates in. Dimensionality is 70x70 pixels. Background pixels are black, the single one-corner path is gray and the target pixels are white.

Our hypothesis is that the agent should be able to learn to locate the target by following the path that leads to the target, independently of where on the path its navigation is started.

B. Problem characterization and related work

This scenario can be formally described as a finite partially-observable Markov decision process (POMDP) which is a 7-tuple $(S, A, T, R, \Omega, O, \gamma)$. Here, the states S are the positions of the agent in the world, the actions A are the four directions the agent can navigate, the transition function T moves the agent in the world according to the direction it chose and the reward R is 1 if the agent reaches the target and 0 otherwise. The current state is only partially observable to the agent because it only observes the world around it with a limited horizon and also it has no information about its absolute position in the world. The observations O are the 7 by 7 pixel sections of the world that can be observed by the agent and the observation probabilities Ω represent the fact that the agent only observes the section of the world around its current position. The discount factor γ controls the influence of early rewards compared to more distant rewards.

As the transition and reward functions are unknown to the agent, this can be described as a reinforcement learning (RL) problem [5]. Consequently, the agent can be trained to learn a behavior policy that maximizes the cumulative reward from interactions with the environment. For RL problems this is usually done by either directly searching for the best policy in the policy space or estimating the value of state-action pairs and developing an optimal policy based on those, which was recently popularized by being applied to ATARI games.

There are multiple challenges of applying RL to this problem. For one, the POMDP underlying our problem is quite large due to the possibly very complex visual input and large environments. Thus, policy or value functions can not be estimated based on lookup tables anymore. One way of dealing with this is using function approximators instead of lookup tables [6]. Furthermore, we can't provide the agent with hand-engineered visual features because, as we stated earlier, the learning problem should not be restricted to a specific environment or target description. Instead we provide the agent with raw pixel observations such that it can extract the representations that are helpful for the specific target the learning procedure is applied to. This raises the need for non-linear function approximators such as neural networks which have recently shown huge successes in computer vision and other domains [7]. Additionally, as we described before, the problem is only partially observable resulting in the well known problem of perceptual aliasing which is also called the hidden state problem [8]. Here, multiple states can be aliased to the same observation which makes it impossible for a memory-less agent to distinguish between theses states. Finding an optimal policy in a POMDP using a memory-less agent has shown to be intractable [9]. Thus, in this project we will also examine different methods of providing the agent with information about past states.

The goal of this project is to apply a learning algorithm similar to a Deep Q-Network (DQN) [2] to our problem, evaluate its performance and test different variants of the algorithm. DQNs have proven to be a general RL technique that succeed in the Atari domain on very different games without adapting the underlying neural network or its hyper-parameters to the specific task.

Similar to our work, Kornuta and Rocki also used the DQN algorithm to solve a navigational task in a partially observable environment [10]. Although, their environment is more visually informative through a heatmap-like color coding around the target location. Furthermore, it is not obvious what reward information was provided to the agent during the learning procedure.

II. Methods

A. Deep Q-Learning

Q-Learning is a model-free RL technique that uses an action-value function $Q(s,a)$ to estimate how much reward can be collected from the current time step on until the game ends when following a certain policy. Instead of searching for the optimal policy directly we try to find the optimal action-value function $Q^*(s,a)$ which is defined as the expectation of this future reward over all possible policies. Using the Bellman equation we can describe $Q^*(s,a)$ as follows:

$$Q^*(s,a) = \mathbb{E}_{s'}\left[r + \gamma \max_{a'} Q^*(s',a') | s,a\right]$$

The idea behind this is that if the optimal value $Q^*(s',a')$ for the state s' at the next time step would be known for all actions a', then the optimal strategy is to select the action a' that maximizes the expected value of $r + \gamma Q^*(s',a')$ [2]. To estimate the optimal action-value function the DQN algorithm uses a neural network with weights θ as non-linear function approximator $Q(s,a,\theta) \approx Q^*(s,a)$. To train the network experience tuples (s,a,s',r) are drawn from an experience buffer and combined to a mini batch D. For every experience we define an error between the network prediction $Q(s,a,\theta)$ and the target value $y = r + \gamma max_{a'} Q(s',a',\theta^-)$. Here, θ^- are the weights for an earlier iteration which are kept fixed for this update iteration. The loss function is then defined as the mean squared error between the network predictions and the target values for all experiences in D:

$$L(\theta) = \frac{1}{|D|} \sum_{(s,a,s',r)\in D} \left[r + \gamma \max_{a'} Q(s',a',\theta^-) - Q(s,a,\theta)\right]^2$$

Now, in every learning step the gradient of the loss function is computed based on the current weights and the newly sampled mini batch. The gradients are then used to update the weights using the RMSProp algorithm. When predicting $Q(s,a,\theta)$ for the current state s we don't compute separate forward-passes for every possible action a, but the network has four output neurons and thus only needs one forward-pass to predict the Q-values for all actions. Also, when asked to choose an action the agent uses an ε-greedy strategy: With probability $(1-\varepsilon)$ the algorithm chooses the action that maximizes $Q(s,a,\theta)$, else it chooses a random action. Thus, the learning procedure is off-policy for $\varepsilon > 0$, because it optimizes assuming a greedy strategy of always choosing the action that maximizes $Q(s,a,\theta)$ but the agent actually behaves differently.

For our project, we use a slightly adapted version of DQN. For example, the original DQN uses a multi-layered neural network including convolutional layers whereas our network only uses one fully connected hidden layer. This simplification makes the back- and forward propagation through the network much less computationally expensive. Also, since the state space of our training environment, as seen in figure 1, is much simpler than in the Atari domain this model complexity might suffice. The hidden layer consists of 64 neurons and uses the rectifier activation function. The output layer consists of 4 neurons, one for each possible action, and uses a linear activation. The following sections explain further adaptions we have implemented.

B. Action history

As described in section I-B we need to provide the agent with a memory over past states to tackle the hidden state problem and thus make the problem tractable. The original DQN algorithm for playing Atari games uses frame stacking for this. As opposed to the Atari domain, in our example the observations are only influenced by the agent actions and no external factor. Thus, instead of providing the agent with a history of observations we provide it with a history of actions. As we can't provide the neural network with a variable length input we apply a function ϕ to transform the sequence of all past actions to a fixed length.

The input to the ϕ-function is the list of four-dimensional vectors which represent the one-hot encoded actions in the order they were executed. We implemented the following ϕ-functions with different methods of summarizing past actions:

1) Last actions: The last n action vectors get appended to a $nx4$-dimensional matrix.

2) Sum: All action vectors get summed up and each element of the resulting vector is scaled to $[0,1]$.

3) Average: All vectors get averaged in a time-dependent fashion with granularity n. For this, a $nx4$-dimensional matrix H_0 with the first row equal to the first action a_0 and all zeros in the remaining rows is created. We apply the following update rule to obtain H_{i+1} form H_i: Starting from the last row $j = n-1$, every row j in H_{i+1} is the result of averaging the row j in H_i and row $j-1$ in H_{i+1}. Lastly, the first row $j = 0$ is simply assigned the vector corresponding to new action a_{i+1}. See figure 2 for a visualization of this update rule.

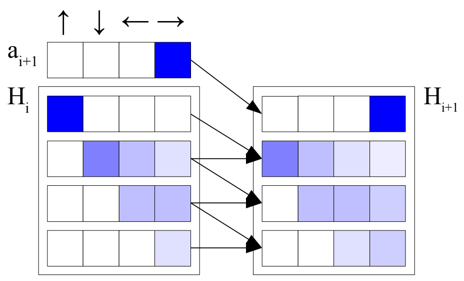


Fig. 2: Update rule for the average action history with granularity $n=4$. The history matrix H_{i+1} is calculated based on the previous history matrix H_i and the last action a_{i+1}. Arrows represent the addition of a scaled source vector to the target vector. In this case H_i represents the history for the action sequence "right, left, down, up" and H_{i+1} for "right, left, down, up, right".

C. Proportional Experience Replay

Instead of using regular experience replay as in the original DQN algorithm, we use a simple form of prioritized replay [11] which we call proportional replay. Here, while gathering experiences from the agent's interactions we distinguish between rewarding and non-rewarding experiences. Rewarding experiences are not only tuples (s,a,s',r) where $r>0$ but all experiences from a training episode in which the agent reached the target and received a reward. Using regular experience replay, the experience buffer mainly consists of non-rewarding experiences because the agent only receives very sparse reward. Thus, the randomly drawn mini batches are less likely to contain rewarding experiences and the agent may take longer to learn an optimal policy. With proportional replay the experience buffer has separate equal-sized sections for rewarding and non-rewarding experiences. An equal number of samples are then drawn randomly from both sections to put together the mini batch. Figure 3 illustrates this process. Both buffer sections get subsequently overwritten with new experiences independently from each other.

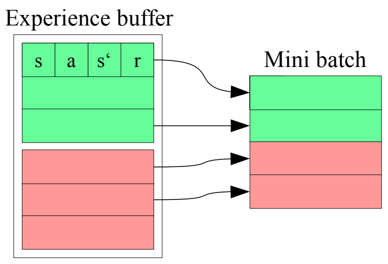


Fig. 3: Training samples are randomly drawn from the two sections of the experience buffer according to the given proportions for the sections. In this example the two buffer sections have the same size. Rewarding experiences are represented in green, non-rewarding ones in red.

D. Regularization

Q-learning with non-linear function approximators is known to have problems with instabilities during the learning procedure and overestimating Q-values [12]. When inspecting the weights of the network we sometimes noticed a connection between continually increasing/decreasing maximum/minimum weight values and overestimation of Q-values. The idea behind introducing regularization is to prevent weight values from getting too extreme which in turn might lead to higher sensibility of Q-values to small input changes and thus possibly to overestimation. For our example, we saw a beneficial effect of using regularization, more specifically weight decay, towards these problems. We implemented regularization by adding the scaled l2-norm of all weights in all layers of the network to our loss function. Empirically, a scaling factor in the order of 10^{-4} yielded the best results.

E. Expansive Sampling

To accelerate training we choose starting positions for the agent that are close to the target for the first training episodes and slowly expanded the sampling space for later episodes. Specifically, we anneal the sampling area towards the whole world over the first 250 episodes. Our hypothesis is that due to this sampling strategy the agent observes the target early on which sparse a longer exploration phase to learn that there exists a target which yield high reward.

III. Experiments

A. Challenges

There are two main challenges to this problem with respect to the experimental procedure. The first one is the high stochasticity of the learning process. We identified five stochastic processes: Choice of a starting position in each episode, initialization of the weights for the neural network, ε-greediness as described earlier, choice of a direction in case the agent is asked to execute a random action, the composition of mini batches from the experience buffer. The high stochasticity leads to a highly non-deterministic learning procedure and thus the learned policy has to be robust against these influences in order to perform well. Also, our experimental procedure has to account for the non-deterministic learning procedure by relying on statistics over repeated executions. The other challenge is the large space of hyperparameters. There are more than 15 hyperparameters to our algorithm. Because of the high computational cost of each learning procedure an exhaustive parameter search is impossible. That's why for our experiments we relied mostly on parameter values used in the original DQN algorithm.

B. Training Procedure

As already described in section I-A, our agent is trained on an environment that is represented by a 70 by 70 pixel image with a one-corner path and a target location on the path. The environment is illustrated in figure 1. We will also refer to this as the static one-corner world.

One training episode consist of sampling a starting position and repeatedly simulating a training step. Within a training step, the agent is asked to choose an a action, i.e. a search direction, which is then simulated by moving the agent accordingly, resulting in a new state. If in the new state the agent is located on the target, i.e. on one of the white pixels, it is receives a reward of $r = 1$ and the episode ends. The training episode also ends if the agent exceeds the maximum step count of 500 or runs out of bounds. In these cases the reward is $r = 0$. The starting positions at the beginning of each episode are samples such that the agent perceives the path in its initial observation but not the target. After each episode the agent history is reset. A complete learning procedure consists of 1000 training episodes. We persist the weights of the neural network model at fixed episode intervals during the learning procedure in order to evaluate agents at different progress levels. The ε factor for the ε-greedy strategy is annealed from 1 to 0.1 over the first 500 episodes and is then fixed at $\varepsilon = 0.1$. Mini batches consist of 32 experiences and the discount factor for future rewards is 0.95. We use a regularization factor of 10^{-4}. To account for the high non-determinism in the learning procedure, as described before, we run 20 learning procedures for every agent configuration resulting in 20 fully trained agent models. In appendix I we provide a complete list of all hyperparameter values used by the algorithm.

C. Testing Procedure

To evaluate the learning procedure we assess the performance of different agent configurations on a dedicated test set. This test set consists of the same environment as it was used for training, see figure 1, and 33 predefined starting positions. For each of these starting positions we run the agent as we do for a training episode but without providing rewards and while fixing $\varepsilon = 0.1$. We then assign a success score of 1 if the agent reaches the target or 0 otherwise. This is analogous to the reward value during the training procedure. We then calculate the average and standard deviation for the success score for all starting positions over the 20 agent models.

D. Randomized Worlds

To assess the generalization behaviour of our algorithm we also trained agents on more difficult environments. For one, we created a set of one-corner worlds by rotating the image used for the previously described scenario, see figure 1. Also, we randomly created more complex environments consisting of a single path with potentially many corners and a single target.

IV. Results

A. Best agent

The agent that performed best on the static one-corner world features a running average action history with granularity of four, proportional replay with 0.5 and regularization with a factor of 10^{-4} achieving an average reward of 0.91 ± 0.12. On this world it did not make a difference for any trained agent whether to use expansive or simple sampling. The performance of the agent quickly converged to the final score within 300 episodes (see figure 4). The steps that it took on average are 45 ± 38, whereas the average distance in steps of the starting positions from the goal in the testset is 30.

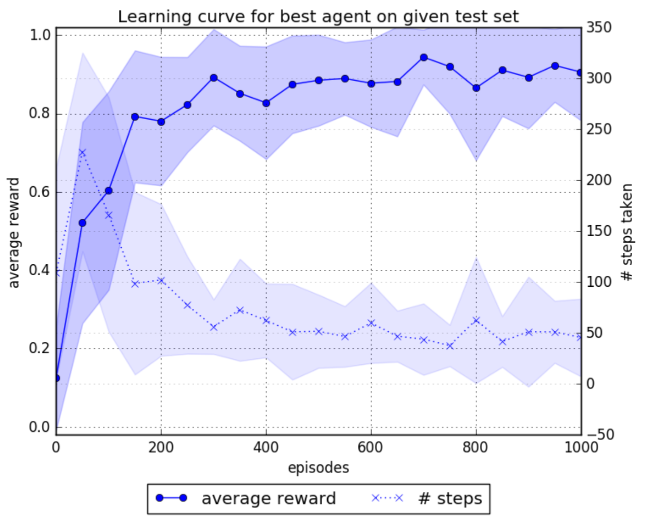
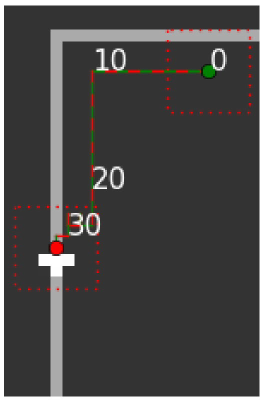


Fig. 4: Performance of best agent with running average action history, proportional replay and regularization. The average reward curve corresponds to the left y-axis and the steps curve corresponds to the right y-axis.

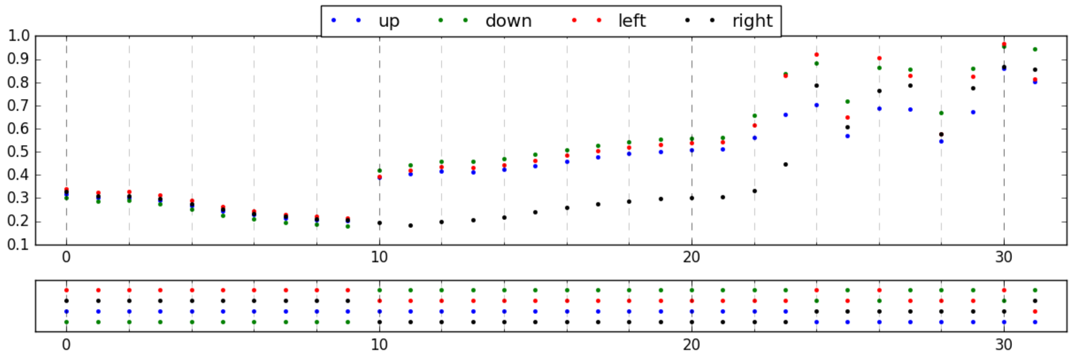
For the best performing agent typical paths went either mainly straight around the corner towards the goal (see figure 5a) or from below straight up, depending on the starting position. The corresponding Q-values for each possible action increased with decreasing distance to the goal (see figure 5b), but since 10% of the actions where chosen randomly not every action corresponds to the highest Q value.

B. Proportional replay and regularization

Proportional replay performs comparable to regular experience replay if no action history is used. However, with action history proportional replay boosts performance drastically (see table I).



(a) Path



(b) Corresponding Q-values to 5a

Fig. 5: Left: Path the best trained agent took starting at the green dot, ending at the red dot. The dotted square corresponds to the view of the agent at these positions. Right: top - Q-values at each step, bottom - sorted Q-values for each action for better overview

Regularization itself boosts the performance of every agent as well. A small 1D grid search was done, to determine the best regularization factor testing 10^{-i} with $i \in (-2,-3,-4,-5,-6)$ were 10^{-4} performed best. Proportional replay was also tested with different portions of $(0.25, 0.5, 0.75)$ giving the same performance, but an increasing amount of overestimated Q-values when using 0.75 ocurs.

agent type	w/o regularization	with regularization
replay + a.h.	0.03 ± 0.10	0.50 ± 0.12
replay	0.46 ± 0.14	0.63 ± 0.16
prop. replay	0.38± 0.16	0.72 ± 0.17
prop. replay + a.h.	0.67 ± 0.15	**0.91** ± **0.12**

TABLE I: Average reward for different agent types with features a.h. meaning average action history of granularity 4, proportional replay and with and without regularization

C. *Action history*

Action history worsens performance with regular replay but greatly enhances performances if proportional replay is used (see table I).

An agent with proportional replay and regularization was also tested with different action histories (see table II), giving the best equal performance for agents that included a history that reflected all past actions (average and rescaled sum). Using the last $N \in (4, 32)$ actions as history decreased the performance of the agent when compared to not using action history at all.

action history type	average reward
no a.h.	0.72 ± 0.17
last 4 actions	0.37 ± 0.09
last 32 actions	0.70 ± 0.26
average (granularity 4)	**0.89** ± **0.12**
average (granularity 32)	**0.89** ± **0.14**
normed sum	**0.90** ± **0.14**

TABLE II: Average reward for agents with different types of action histories using proportional replay and regularization

agent type	reached goal
replay	20%
replay + a.h	60%
prop. replay	35%
prop. replay + a.h.	100%

TABLE III: Agents that have learned to reach the target from starting positions in area 1 **and** 2; 20 agents of each type were trained and tested

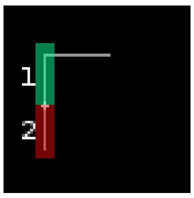


Fig. 6: Two areas of starting positions

Adding action history increases the amount of agents, that are able to learn a strategy to reach the target from below as well as above where the pure visual input is the same (see table III and figure 6). When only having visual input, presented with a vertical line, most agents learn either to go only up or to go only down, but just a few yield a strategy to go down on one side and up on the other side of the vertical line in order to reach the goal.

D. *Randomized worlds*

The best agent and agents that had more hidden layers and neurons were also trained and tested on completely random worlds, with more than one corner. The result was a low mean success-rate of around 20% for all on a test set that included many starting positions, where they had to pass at least one corner to reach the target.

Trained and tested on a bit simpler environment of different worlds containing only one corner, resulted in a mean success rate of around 35% on a test set for one-corner worlds. We also evaluated the performance when introducing more information about the problem by providing small intermediate rewards (a) when the agent stays on the path, (b) the sum of its view times a factor and (c) punishment when running out of bounds or reaching maximum steps. None of these strategies did make a difference regarding the success rate compared to just giving a reward at the target.

Looking at the performance of each individually trained agent on one-corner worlds affirmed, that not one agent was able to converge to a strategy that would yield a success rate higher than 40% on the test set.

V. Discussion

On a static world with a one-corner path we found an agent configuration that learned to use the visual input and memory (that included a history of all past actions of an episode) in order to stay on the path and reach the target. In this reduced context it was therefore able to solve the task.

A. *Interaction of action history and prop. replay*

Combining action history with proportional replay results in a positive effect, whereas just adding an action history without prop. replay yields a negative effect. The action history brings in more structural knowledge, but at the same time increases the complexity of available information, which only could be exploited by the agent by adding proportional replay.

B. *Different action histories*

Using an action history that incorporated the information of all past actions in one episode (average or normed sum) had a greatly beneficial effect, since the agents could overcome the problem of perceptual aliasing [8]. Without it the visual input of a straight vertical line is ambiguous regarding the agents hidden state in the world. A memory in form of an action history changes this.

C. *Regularization and Q-value overestimation*

During training and testing a behaviour of the agent to avoid the goal was recognized. This was connected to an overestimation or even divergence of Q-values, which is a known problem when using Deep Q-Learning [12]. By adding regularization the occurrence of overestimation was greatly reduced and even when occurring, the Q-values converged back to the value of the given reward at the target. We did not see a similarly beneficial effect of Double Q-learning [12], error clipping [2] and batch normalization of network layers [13, 14].

D. *Problems with random worlds*

As soon as randomness in the worlds is introduced no agent of the current setup or even more complex (with more hidden layers and neurons) is able to extract the underlying structure properly in order to learn good performing strategies. So it can be overfitted to one static example but can not generalize to the more complex problem. Even when receiving additional information through intermediate rewards the agents fail to learn a high performing strategy. The reason might come from one of the following sources:

(0) Not enough information is passed from the environment to the agent. This should not be the case, since a human can immediately think of strategies to apply that would work and we also tested to give more information through intermediate rewards.

(1) The architecture of the network (type of network, hidden layers, activation function, ...) might not suit the problem, e.g. not complex enough or another one (convolutional layers) would be better.

(2) The memory in form of action history might be insufficient. So rather incorporating an internal memory might help.

(3) The learning procedure, i.e. DQN, can be rather unstable [12], which we also witnessed with our problem.

(4) Last the large set of hyperparameters might also not be chosen properly to suit with the random worlds, although short tests were conducted or literature references were used to determine the most promising parameter values.

VI. Future Work

In order to tackle the challenge to generalize to more complex problems (like random one-corner worlds), we identified two main areas, which could be enhanced and further investigated.

A. *Methodology*

On the one hand that is to enhance the capabilities of the current software in order to enable the following features:

(1) A way to do automated hyperparameter optimization in order to systematically find more suitable hyperparameters.

(2) Visualize the agent's decision making with Layer-wise Relevance Propagation [15] for a better understanding of the agents behaviour. This would enable a more thorough analysis of what the agent focuses onto most and might help to understand what is currently missing to solve the random world task properly.

(3) Enhancing the testing capabilities by not only testing the agents on one common test set, but also on specified sets that investigate certain behaviours, for example going around a corner, reaching a target from below, etc.

B. *Other techniques*

On the other hand other promising techniques could be implemented. To increase the robustness of learning the first layers could be pretrained as Heess et. al. did [16] or with autoencoders, to first learn features of the input and then only train subsequent layers to solve the regression task. Or more sophisticated approaches to prioritized replay could be used [11] to further leverage rewarding experiences and learn from actions that led to them.

Using recurrent neural networks (e.g. LSTMs) to include another form of memory could help to solve the hidden state problem of this POMDP even better. By adding noise or other features to the world the hidden state problem could also be made less complex.

Also, instead of using DQNs, the authors of the original ATARI paper Minh et al. proposed asynchronous gradient descent for optimization of deep neural network controllers (see their A3C algorithm [17]), which performed better on the ATARI domain with less training effort. It also performed well on the task of navigating a 2D and 3D maze, which is quite similar to the problem described in this report.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[3] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[4] M. Moravčík, M. Schmid, N. Burch, V. Lisỳ, D. Morrill, N. Bard, T. Davis, K. Waugh, M. Johanson, and M. Bowling, "Deepstack: Expert-level artificial intelligence in no-limit poker," *arXiv preprint arXiv:1701.01724*, 2017.

[5] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press Cambridge, 1998, vol. 1, no. 1.

[6] L. Baird *et al.*, "Residual algorithms: Reinforcement learning with function approximation," in *Proceedings of the twelfth international conference on machine learning*, 1995, pp. 30–37.

[7] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[8] S. W. Hasinoff, "Reinforcement learning for problems with hidden state," *University of Toronto, Technical Report*, 2002.

[9] M. L. Littman, "Memoryless policies: Theoretical limitations and practical results," in *From Animals to Animats 3: Proceedings of the Third International Conference on Simulation of Adaptive Behavior*, vol. 3. MIT Press, 1994, p. 238.

[10] T. Kornuta and K. Rocki, "Utilization of deep reinforcement learning for saccadic-based object visual search," *arXiv preprint arXiv:1610.06492*, 2016.

[11] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *arXiv preprint arXiv:1511.05952*, 2015.

[12] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning." in *AAAI*, 2016, pp. 2094–2100.

[13] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[14] 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 preprint arXiv:1509.02971*, 2015.

[15] S. Lapuschkin, A. Binder, G. Montavon, K.-R. Müller, and W. Samek, "The lrp toolbox for artificial neural networks," *Journal of Machine Learning Research*, vol. 17, no. 114, pp. 1–5, 2016. [Online]. Available: http://jmlr.org/papers/v17/15-618.html

[16] N. Heess, G. Wayne, Y. Tassa, T. P. Lillicrap, M. A. Riedmiller, and D. Silver, "Learning and transfer of modulated locomotor controllers," *CoRR*, vol. abs/1610.05182, 2016. [Online]. Available: http://arxiv.org/abs/1610.05182

[17] 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," in *International Conference on Machine Learning*, 2016.

Appendix I
Hyperparameters

Hyperparameter	value
world	
world size	70 x 70 pixel
view size	7 x 7 pixel
training episodes	1000
maximum steps	500
copies run per agent	20
agent	
hidden layer	64 fully c., rectified act.
output layer	4 fully c., linear act.
ε annealing	linear from 1 to 0.1 over 1/2 of episodes
RMSprop	$lr = 0.0001$, $\rho = 0.9$, $\varepsilon = 10^{-6}$
batch size	32
discount rate	0.95
clone interval	250
buffer size	10^5
steps to start learning	1000
learning at every kth step	1
proportional replay portion	0.5
reward at target	1
average action history granularity	4

TABLE IV: Used default hyperparameters