Learning Machines:
Final Report - Group Echo

Ilze A. Auzina
Msc Artificial Intelligence
Vrije Universiteit Amsterdam
Amsterdam, the Netherlands
ilze.amanda.auzina@gmail.com

Suzanne Bardelmeijer
Msc Artificial Intelligence
Vrije Universiteit Amsterdam
Amsterdam, the Netherlands
suzanne-bardelmeijer@live.nl

Abstract

This document contains the final report on the course *Learning Machines* given at the Vrije Universiteit Amsterdam in January 2020. Deep Q-Learning learning method was investigated as an optimal solution for three tasks: (1) obstacle avoidance, (2) foraging and (3) predator-prey behaviour. The performance was evaluated both in simulation and in real-life. The final results indicate that the chosen method is an appropriate solution for the tasks at hand, as the goal of each task was achieved. The reality-gap caused minor difficulties, however they were overcome as the project progressed. Therefore, the present papers confirms the existing literature that deep-reinforcement learning can be successfully applied to learning machines both in simulation and real-life.

1 Introduction

Learning machines are agents that learn how to execute a given task by learning through experience. In order to achieve this the agent must be able to (1) receive information from the environment in the form of a sensor, (2) decide to perform an action given that information (controller) and (3) execute the action (actuator), while receiving an evaluation of the performed action in a state through a penalty or a reward. The goal of the current report is to find the best learning method that would allow an agent to learn by itself the optimal action to execute given a certain goal and environmental input. In particular, the learning method will be evaluated on three main tasks: (1) obstacle avoidance, (2) foraging, and (3) predator-prey relationship modeling. As the present environment represents a sequential decision-making problem it can be viewed and modelled conceptually as a Markov decision process (MDP) with a continuous state space. A common solution approach for MDP problems are temporal difference (TD) methods as they do not require a model of the environment, its reward and next-state probability distributions [11], which is desirable for the present problem as no model of the agent-world interaction is required. One of such algorithms is Q-learning, an off-policy TD control algorithm, which combined together with a neural network as a function approximator has achieved a human level performance on many Atari video games with continuous input space [7]. The resulting algorithm is called 'Deep Q-Learning', and it has showed to perform well with high-dimensional observation spaces and discrete, low-level action spaces [7]. Consequently, it was chosen as an appropriate learing method for the present goal.

2 Related work

The interest in TD methods as a possible game solution sparked in 1990s, when reinforcement learning was successfully applied to backgammon [12]. However, application of the same method to other games was less successful, hence, leading to a belief that the method is only applicable to a backgammon setting. This view has changed in recent years with the growing interest in combining deep learning with reinforcement learning. In particular, the utilization of a neural network (NN) allows to learn successful control policies [7], where instead of learning all action values in all

states, a parameterized value function is learned by the network [13]. One of such early examples is neural fitted Q-learning (NFQ), where the basic underlying idea is that instead of updating the neural value function on-line, the update is performed off-line considering an entire set of transition experiences [10]. This implementation resolved the issue that typically several ten thousands of episodes had to be done until an optimal or near optimal policy was found with an on-line update rule. Furthermore, an additional problem that arises from combing model-free reinforcement learning algorithms with neural networks comes from the fact that subsequent states in reinforcement learning tasks are correlated [8]. A way to overcome this issue is by incorporating replay memory and updates are made on a set of tuples (s, a, r, $s^{'}$), called a minibatch, sampled from the replay buffer [7]. Besides the advantage of covering a wide range of the state-action space, minibatch updates have less variance in comparison to single tuple updates [1]. Apart from successful application in the domain of video games, different variants of Deep Q-Learning also have been applied to real-life robots.

First applications of reinforcement learning for robots using NN dates back to the same time period of TD-backgammon success, where it was concluded that artificial agents can acquire complex control policies effectively by reinforcement learning [6]. In more recent years, different variants of Deep Q learning have shown to perform well on various different tasks and complex environments [3]. For example, deep double-Q network, has shown to successfully learn obstacle avoidance based on monocular RGB vision, and generalize well to various new environments [14]. Furthermore, it also has been applied in the domain of cooperative agents, such as robot swarms, with the goal of finding and maintaining a certain distance and locating a target [5] [4]. Nonetheless, not all implementations have been as successful. Special focus has to be paid to the transfer of the performance in simulation to real hardware, where the mismatch between the input stimulus can cause a drop in performance [15].

3 Methodology

3.1 The proposed approach

As mentioned earlier, the proposed approach is Deep Q-Learning algorithm. As the word 'deep' implies, the main difference from a standard Q-learning implementation is that the state-action values are estimated by a neural network. The Deep Q-Learning was implemented with experience replay, to overcome the fact that traditional Q-learning algorithms are inefficient with respect to their usage of information: experiences are obtained by trial-and-error and are utilized only once to adjust the networks, then thrown away [6]. This is wasteful because some experiences may be rare and costly to obtain. Applying experience replay solves this issue, because it allows the agent to re-use its experiences, therefore, even rare events can be sufficiently presented in the memory, thus allowing to learn from them. A pseudo code for Deep Q-Learning is presented in Appendix A.

As the chosen learning method is based on a reinforcement learning framework, certain assumptions about the environment were made. Formally, reinforcement learning algorithms are implemented under the assumption of the Markov Property: 'a state signal that succeeds in retaining all relevant information is said to be Markov' [11]. From this definition, one can clearly conclude that the the real world environment violates this assumption as the environment is not fully observable to the robot and it is dynamically changing. Therefore, for the current model the state represents an approximation of a Markov state: it does not fully satisfy the Markov property, but it is sufficient for predicting future rewards and for selecting actions, allowing a successful implementation of Q-Learning. Furthermore, the utilization of a neural network as a function approximator requires to pay detailed attention to the model parameter settings to avoid overfitting on the training data. In the present use case the input data is very limited, hence, the hidden layers should be made relatively simple with additional regularization methods to reduce overfitting (dropout) [2].

3.2 The controller representation

A feed-forward neural network was used as the controller in order to learn the correct mapping from the input states to state-action values. The input represents a sensor reading (s) from the environment, while the output is a q-value for all the possible actions (a) in the corresponding state mapped via a weight matrix (θ), $Q(s, a; \theta)$ respectively. The weight matrix was updated using gradient descent with the loss function defined as mean-squared-error (equation 1).

$$L_i\left(\theta_i\right) = E\left[\left(y_i - Q\left(s, a; \theta_i\right)\right)^2\right] \tag{1}$$

3.3 The learning algorithm

Q-Learning with epsilon-greedy exploration was used to learn the optimal action policy for any state encountered. This was achieved by iteratively updating the Q table. The update is derived from Bellman equation [9], where depending on the state 'type' a different update rule was used: for non-terminal states equation 2, terminal states equation 3, where, r represents the reward obtained, and γ represents the discount factor, which influences how much the agent 'cares' about future reward.

$$y_i = E\left[r + \gamma * \max_{a'} Q\left(s', a'; \theta_{i-1}\right) | s, a\right] \tag{2}$$

$$y_i = E[r|s, a] \tag{3}$$

3.4 Image pre-processing

For some of tasks the readings from the camera are utilized as input, therefore, image pre-processing was performed to extract only the relevant features from the image. To achieve this openCV package was used, and the following steps were executed: (1) images were smoothed using a Gaussian blur to reduce noise introduced by different light-brightness in the real-life environment, (2) 'blobs' were extracted using the HSV (hue, saturation, lightness) method as it was more accurate than using the RGB color model (different colors can fall in the same RGB range, resulting in a suboptimal color extraction), (3) the middle coordinates (x,y) of the closest 'blob' were extracted and normalized with respect to the image size (the closest 'blob' was identified by comparing the sizes of the different 'blobs' identified in an image). Consequently, applying image pre-processing extracted 3 values of the closest 'blob': its middle coordinates (x,y) and size.

4 Experimental setup

The code was written in python 2.7 and the package 'Tensorflow 2.2.4.' was used for the designs of the neural networks. All simulations were implemented in the software VREP (CoppeliaSim Edu V4.0.0), and the real-life performance was executed by 'Robobo' an educational robot.

4.1 Task 1: Obstacle avoidance

For the first task the agent had to learn how to explore the environment while successfully avoiding any obstacles encountered. Hence, to achieve such a behavior an optimal reward function should promote exploration most of the time, but once an object is detected switch towards actions that would ensure successful avoidance.

For the present task the infrared readings from the robot were used as the input state. It was chosen to only use the front sensor readings (5 values), as the robot's actions were limited to 3 moves: front, left, right. Given these environment settings the corresponding reward function was created (see Table 1 in Appendix B), where (1) high negative reward was given for crashing to avoid riding into obstacles, (2) high positive reward was assigned for action forward to promote exploration, (3) a small negative reward was assigned to right and left movements to prevent rotation, and lastly for all actions executed an 'additional reward' was added, which was estimated by summing the absolute values of the IR reading, when the agent sensed an object, to promote a movement towards an object but not too close. Gamma (γ) was set to 0.9 to make the agent very susceptible to possible future punishments, meaning that it should avoid crashing at all times, as well as ϵ was initialized to 0.9 at the beginning of the learning process to promote exploration of all possible states, and it was decremented overtime by $1/N_{iterations}$.

Given the created reward function, observations could be gathered for the NN, which would take as input the IR reading (5 values), pass them through a hidden layer (128 units, relu activation, dropout 0.2), and map them to the corresponding action q-values (3 values, linear activation). To achieve this, initially 500 observations were gathered with random movements, after which the learning process was initiated: estimating the Q-table via the NN, and updating the q-values using Q-learning. The agent was trained for 9500 iterations, with batches of 64 observations extracted from the buffer, learning rate set to 0.01 and optimizer set to RMSprop, respectively. The training was executed in an environment with 12 obstacles, all rectangular shaped.

4.2 Task 2: Foraging task

In the second task the agent's overall goal it to collect food items. This can be further subdivided in 3 main problems: localize (find object), approach object and consume as much as possible. From an

implementation perspective this means (1) applying image pre-processing to extract only relevant features, (2) designing a reward function that encourages approaching the correct object and (3) take the shortest path to consume as many objects as possible.

All the food items were presented in color green in the environment, and none of the food items were hidden from the agent's visual field. Therefore, the HSV range for image-preprocessing was limited to color's green hue, and it was assumed that the agent is always able to see the food it needs to collect. To encourage the agent to look for food a negative reward was received at every time point if the agent does not detect any food in its visual field. If the agent detects an object, it is encouraged to rotate in its direction by receiving a positive reward that is influenced by a '*center penalty*' score. The '*center penalty*' is defined as the absolute distance from the center of the image, meaning that if the distance is small, the agent is facing the object correctly, thus, action forward is encouraged with additional positive reward, while for all other actions the reward is decreased by the distance value to promote rotation in the food's direction. Finally, the agent receives a high positive reward if it has collected an object. An overview of the rewards associated with task 2 can be found in Table 2 in Appendix B.

Initially the input to the NN was defined as 3 values: x and y coordinates of the 'blob' and its size. Due to the reduction in input space, also the hidden layer size was reduced to 10 from 128, while the remaining network settings were kept the same as before (see task 1). The minor changes made were that the learning time was extended to 19500 iterations and there was only one update rule for Q-learning as there were no terminal states and γ was reduced to 0.6 to prevent a preference to action forward, just because it collects food quite often. Subsequently, based on the results obtained the input state was extended to 6 values combining the values of the present state s_t and the state after action execution s_{t+1}. This allowed to agent to have more fine grained actions, as it provided memory of the object's location, even if it has passed it by accident. The agent was trained in an environment with 12 green objects with random initialization of the object location when all the objects were collected, or it got stuck at a wall.

4.3 Task 3: A predator chasing a prey

The third task involves two agents, a predator and a prey. The goal of the predator is to catch the prey as fast as possible, while for the prey - to escape successfully. Therefore, two controllers are needed. The similar aspect of both is that they share a common problem of successful localization (see where the other agent is), while the individual aspects are, for the predator - approaching an agent, and for the prey - moving away from an agent while performing obstacle avoidance.

The camera of the predator perceives the front, whereas the camera of the prey is rotated 180 degrees to detect the back. Both cameras are tilted at a 70 degree angle as the predator and prey are relatively low in height. Considered that in real life both predator and prey are able to perceive the entire arena with respect to the camera's position, it was chosen not to expand the simulated arena to imitate the real conditions as much as possible.

As the present task for the predator is quite similar to its previous task, firstly, it was investigated whether the previous model could be applied to the current problem. Therefore, as the prey was depicted in red, the color extraction hue was changed to red for the predator and different models were evaluated based on how many time steps the predator needs before catching the prey both in simulation and in real-life with a very simplistic prey (random actions). Based on this evaluation the model, which was trained for 15000 iterations, was chosen as the most optimal for the predator's controller, switching the focus on modeling different options for prey's controller and investigating the selected predator's performance on them.

In total 4 controllers were established for the prey, and they were modelled with increasing action complexity (see Table 4 in Appendix C). The first 3 models were fully hard-coded, hence, only having a controller, as no learning was needed, while for the last model a learning method was combined with hard-coded rules. The hard-coded rules were used for obstacle avoidance, while the actual behavior was implemented by following the same controller and learning algorithm representation as for the predator, but with different reward definitions (see Table 3 in Appendix B). In particular, the same negative reward was assigned for every time point to promote the prey to look for the predator, but then rather than approaching it, a larger positive reward was given for the actions that would reduce the size of the object detected (modelled by the *size penalty*), supporting an action that would avoid the predator. All the remaining settings were kept the same as for task 2, apart from γ which

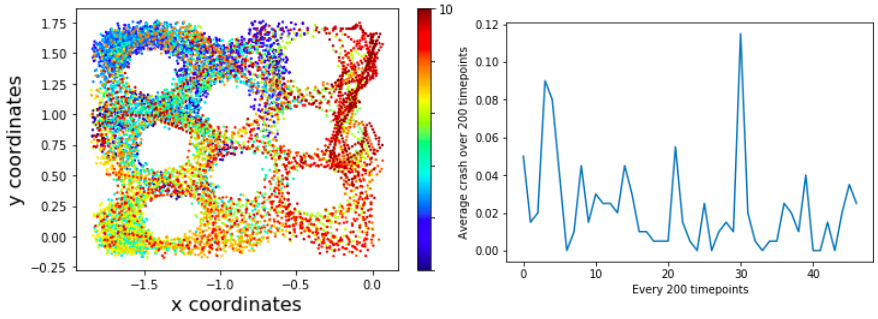
was increased back to 0.9 to make the prey very reactive to avoid being captured, and the training time was reduced to 14500 iterations due to time limitations.

5 Results

For accurate evaluation of the learning method's settings see Appendix D, where the total cumulative reward is used to evaluate the accurate implementation of Q-learning, and the structure of the NN is evaluated by examining the loss overtime.

5.1 Task 1: Obstacle avoidance

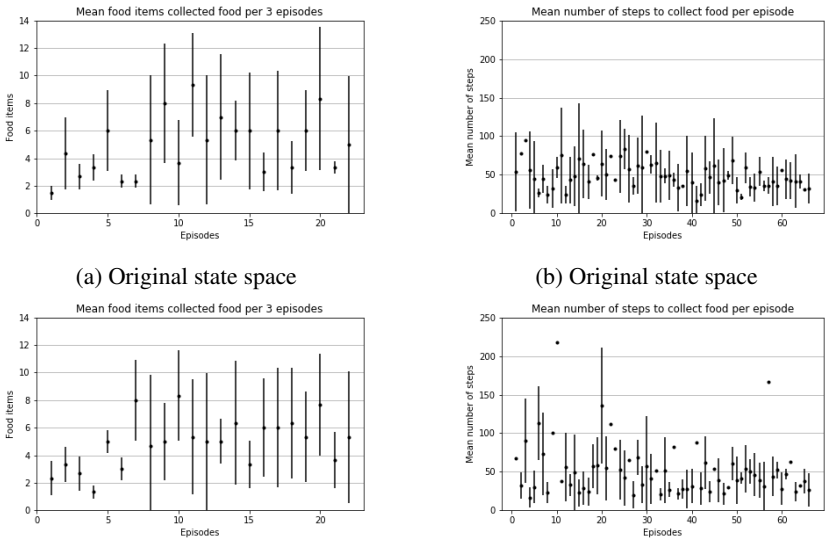
In Figure 1a the location of the agent overtime is plotted, where the blue color represents the start of the episode and the red color the end of it. It can be observed that over time the agent visits every location in the arena. This indicates that the agent is exploring the arena sufficiently. Furthermore, figure 1b shows that the average number of collisions decreases over time. Together the findings from Figure 1a and 1b suggest that the chosen learning method combined with an accurate reward definition is able to learn an obstacle avoidance policy with sufficient exploration. However, the transfer of the high-performance in simulation to real-life was not successful, due to calibration issues of the IR readings, hence, it was not possible to evaluate the learned controller in real-life.



(a) Locations visited

(b) Average crash per 200 iterations

Figure (1) Results Task 1



(a) Original state space

(b) Original state space

(c) Extended state space

(d) Extended state space

Figure (2) Results Task 2

5.2 Task 2: Foraging task

To asses the performance the average number of food items collected and the average number of steps needed to collect food items were used as an evaluation metric. Figures 2a and 2b show the obtained results of a state space definition that only takes the current state into account. From Figure 2b it

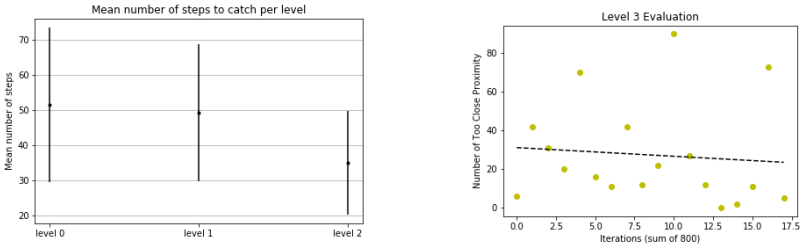
can be observed that over time the mean number of steps the agent needs to collect food declines, however, the mean number of collected food items fluctuates quite heavily. If compared to Figures 2c and 2d it can be seen that extending the state space by incorporating the previous state does lead to better results. Figure 2c shows a clear increase in mean food items collected over time, while at the same time Figure 2d shows a distinct decrease in mean number of steps the agent needs to collect food (the overall fluctuations observed in all plots are due to the random initialization of every episode: just by chance the food items could be position closer or further away from the agent). The observations suggests that increasing the state space helps the agent to find the food more successfully, as well as all objectives were met: (1) localize, (2) approach and (3) collect food. The present model also performed extremely well in real-life which can be attributed to a very successful calibration of the input features.

5.3 Task 3: A predator chasing a prey

The trajectories of the predator and prey during an episode were investigated and plotted in Figure 9a, 9b and 9c (see Appendix E). The trajectories are marked with colored dots, again the blue color indicates the start and the red color the end of the episode. In all Figures it can be observed that the prey's behavior follows the hard-coded rules and, although an episode ends when the prey is captured by the predator, the trajectories do not overlap at the end of the episode. This, however, can be attributed to the size of the agent.

The number of steps (mean average over 10 episodes per level) the predator needs to catch the prey is shown in Figure 8a for the first 3 levels of the prey. Surprisingly, it can be observed that as the behaviour of the prey advances the mean number of steps needed for the predator decreases. An explanation can be found in the "walls" of the arena. At both Level 0 and Level 1 the prey does not take the walls of the arena into account and can therefore get stuck at the wall. Hence, for the predator to catch it, it has to do the entire distance until the wall, while in Level 2 the prey knows how to avoid the wall, but in its attempt to avoid it - it can accidentally walk straight into the "arms" of the predator, resulting in a catch with even fewer steps. These findings suggest that more advanced behaviour does not necessarily perform better at escaping the predator in simulation. Due to limitations of the VREP application, it was not possible to evaluate the two learned models at the same time in simulation. Therefore, another evaluation metric is used to evaluate the performance of the learned prey controller. Figure 3b shows the number of time points (cumulative count over 800 iterations) that the predator was in too close proximity of the prey. Overtime, it can be observed that this number decreases which suggests that the prey successfully learned to avoid the predator.

Lastly, all levels were also evaluated in real-life. The transfer of simulation to real-life was successful, as calibration had been done accurately, however, the conclusions of which is the best controller for the prey differed. In real-life Level 2 controller had the best performance, followed by the learned controller, and then relatively similar Level 1 and Level 0. This is because in real-life successful wall avoidance becomes a crucial point for avoiding the predator, and because the size of the arena is bigger, the chance of falling into predator's arms while avoiding a wall are less likely to occur. Even though, the learned controller had successfully learned to move away from the predator if it sees it, the performance was sub-optimal because once the predator was not in its visual field, it tried to search for it by doing a 360 degree rotation, however, while executing this search, the predator had already caught it. A solution would be changing the camera's position to a horizontal view space (similar to real-life prey animals), or adding an additional positive reward for turning in the direction that is in opposition to prey's location.



(a) Mean number of steps to catch prey per level (b) Cumulative number of 'too close' proximity

Figure (3) Results Task 3

6 Conclusions

Standard machine learning approaches are very powerful yet static models, hence they do not function well in real-life, which is a highly dynamic environment. Therefore, there is a need for methods that are able to learn and adapt overtime. Nonetheless, one can successfully apply a static model in a combination with dynamic programming to bridge the gap between the two, such as Deep Q-learning. The present paper shows that the present learning method is able to solve three different tasks: (1) obstacle avoidance, (2) foraging and (3) predator-prey behaviour. The chosen strategy, Deep Q-Learning, allowed the agent to learn from experience, have a continuous state space as input and to choose the optimal action given a certain goal as defined by the reward function. Furthermore, during the project we experienced how important it is to take the reality gap into account: a perfect model in simulation is useless in practice, unless you are able to relate the simulated parameter readings to real-life. Furthermore, the performance in simulation is not always in line with the one in real-life, as often the environment is not comparable. Nonetheless, despite the short learning time, the present paper confirms the growing trend in the literature that deep-reinforcement learning can be successfully applied to learning machines both in simulation and real-life.

Acknowledgments

We would like to thank the teaching staff for all the suggestions received, as well as for the positive work environment experienced.

References

[1] V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare, and J. Pineau. An introduction to deep reinforcement learning. *CoRR*, abs/1811.12560, 2018.

[2] G. E. Hinton, A. Krizhevsky, I. Sutskever, and N. Srivastva. System and method for addressing overfitting in a neural network, Aug. 2 2016. US Patent 9,406,017.

[3] B.-Q. Huang, G.-Y. Cao, and M. Guo. Reinforcement learning neural network to the problem of autonomous mobile robot obstacle avoidance. In *2005 International Conference on Machine Learning and Cybernetics*, volume 1, pages 85–89. IEEE, 2005.

[4] M. Hüttenrauch, S. Adrian, G. Neumann, et al. Deep reinforcement learning for swarm systems. *Journal of Machine Learning Research*, 20(54):1–31, 2019.

[5] M. Hüttenrauch, A. Šošić, and G. Neumann. Guided deep reinforcement learning for swarm systems. *arXiv preprint arXiv:1709.06011*, 2017.

[6] L.-J. Lin. Reinforcement learning for robots using neural networks. Technical report, Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1993.

[7] 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.

[8] S. S. Mousavi, M. Schukat, and E. Howley. Deep reinforcement learning: an overview. In *Proceedings of SAI Intelligent Systems Conference*, pages 426–440. Springer, 2016.

[9] S. Peng. A generalized dynamic programming principle and hamilton-jacobi-bellman equation. *Stochastics: An International Journal of Probability and Stochastic Processes*, 38(2):119–134, 1992.

[10] M. Riedmiller. Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In *European Conference on Machine Learning*, pages 317–328. Springer, 2005.

[11] R. S. Sutton, A. G. Barto, and R. J. Williams. Reinforcement learning is direct adaptive optimal control. *IEEE Control Systems Magazine*, 12(2):19–22, 1992.

[12] G. Tesauro. Temporal difference learning and td-gammon. 1995.

[13] H. van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. 2016.

[14] L. Xie, S. Wang, A. Markham, and N. Trigoni. Towards monocular vision based obstacle avoidance through deep reinforcement learning. *arXiv preprint arXiv:1706.09829*, 2017.

[15] F. Zhang, J. Leitner, M. Milford, B. Upcroft, and P. Corke. Towards vision-based deep reinforcement learning for robotic motion control. *arXiv preprint arXiv:1511.03791*, 2015.

A
Pseudocode Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-Learning with Experience Replay

Initialize replay memory *D* to capacity *N*
Initialize action-value function *Q* random weights
Initialize state s_t
  For t in iterations **do**:
    If $i <$ gather observations:
      select random action a_t
    Else:
      With probability ϵ select a random action a_t
      otherwise select a_t=$max_a\ Q^*(\phi(s_t), a; \theta)$
    Execute action a_t and observe r_t and s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in *D*
    Sample random minibatch of transitions $(s_t, a_t, r_t, s_{(t+1)})$ from *D*
    Set y_t = $\begin{cases} \text{r}_j \text{ for terminal } s_{t+1} \\ \text{r}_j + \gamma max_{a'} Q(\phi(s_{t+1}), a^{'}; \theta) \text{ for non-terminal } s_{t+1} \end{cases}$
    Perform a gradient descent step on (y_t - $Q(\phi(s_t), a_t; \theta))^2$

B
Reward definitions for Task 1, 2, and 3

Condition	Reward
$a_{forward}$	$+8$ + $r_{additional}$
$a_{right/left}$	-1 + $r_{additional}$
$crash$	-15

Table (1) Rewards Task 1

Condition	Reward
every time point	-1
$x! = 0\ AND\ y! = 0$	+2.5 - 4 * $center\ penalty$
$a_{forward}\ AND$ *center penalty* < 0.1	+4
$collected\ food$	4

Table (2) Rewards Task 2

Condition	Reward
every time point	-1
$x! = 0\ AND\ y! = 0$	+3 -4**size penalty*
caught	-6

Table (3) Rewards Task 3 Prey

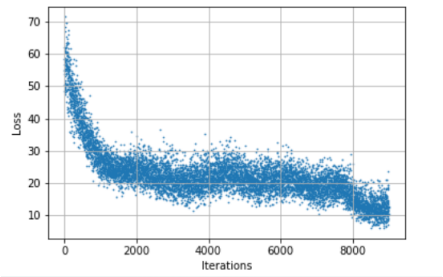
C
Prey behaviour levels

Level	Behavior
0 (random)	*equal probabilities for all actions*
1 (forward focused)	*0.6 probability of* $a_{forward}$*, 0.4 probability for* $a_{right/lext}$
2 (smart)	*First look if at wall, if not, look for predator, if not, random action*
3 (intelligent)	*Learned behaviour*

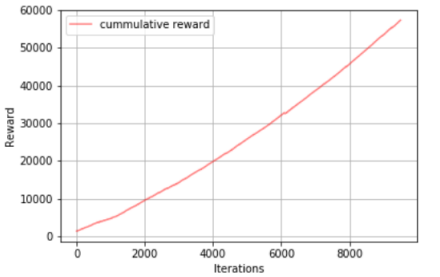
Table (4) Levels associated with prey behaviour

D
Loss and cumulative rewards for Task 1, 2 and 3

D.1 Task 1



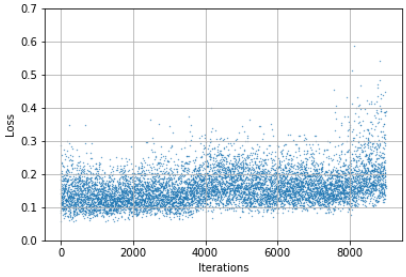
(a) Loss



(b) Cumulative Reward

Figure (4) Evaluation of the Controller and Learning Algorithm

D.2 Task 2



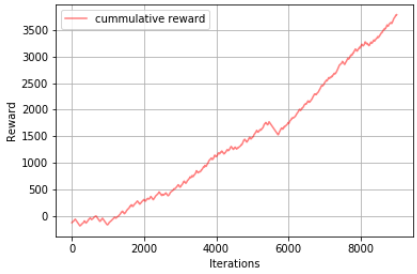
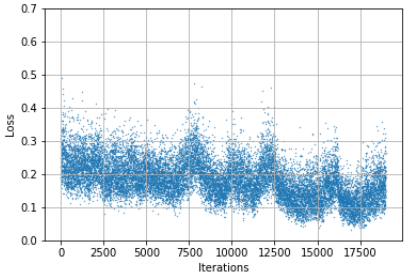


Figure (5) Evaluation of the Controller and Learning Algorithm: 128 Hidden Units



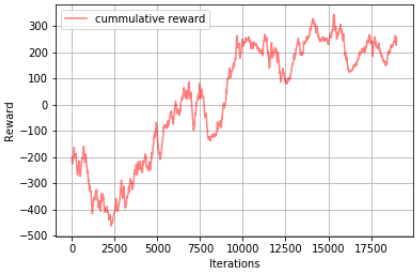
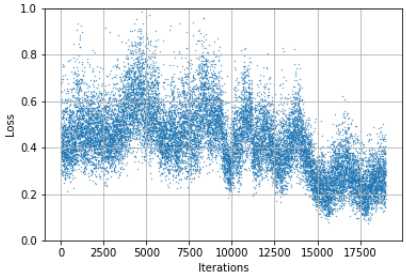
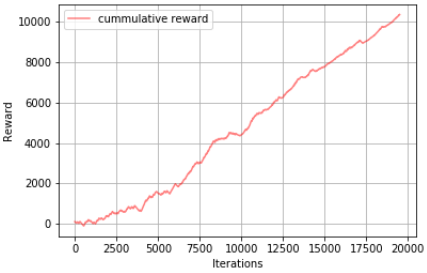


Figure (6) Evaluation of the Controller and Learning Algorithm: 10 Hidden Units



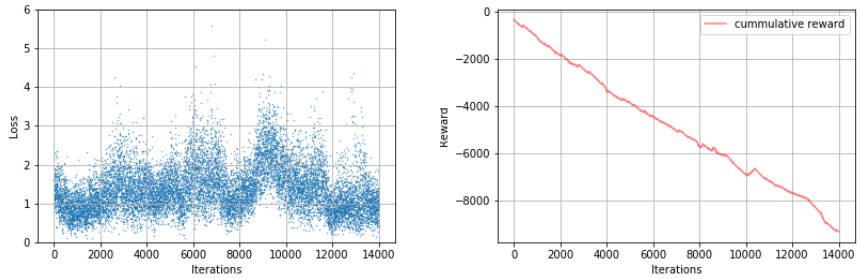
(a) Loss



(b) Cumulative Reward

Figure (7) Evaluation of the Controller and Learning Algorithm: 10 Hidden Units Extended State Space

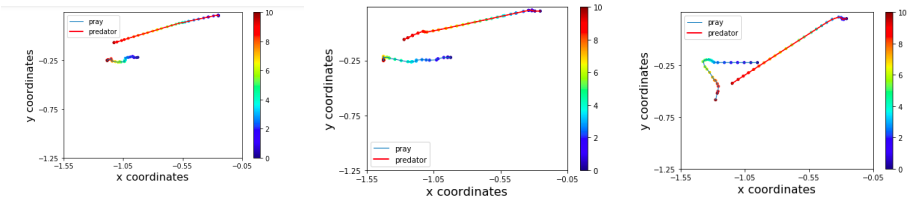
D.3 Task 3



(a) Loss (b) Cumulative Reward

Figure (8) Evaluation of the Controller and Learning Algorithm: Prey

E
Trajectories of predator and prey



(a) Prey Level 0 (b) Prey Level 1 (c) Prey Level 2

Figure (9) Trajectories of predator and prey at different prey behaviour levels