Map-less Navigation in Novel Environments using Deep Reinforcement Learning

A Curiosity-driven approach for Mobile Robotics

Frederik Nørby Rasmussen

Autonomous Systems, 2019-06

Master's Project

Department of Materials and Production
Aalborg University
http://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Map-less Navigation in Novel Environments using Deep Reinforcement Learning

Education:
Autonomous Systems

Project Period:
Spring Semester 2019

Participant(s):
Frederik Nørby Rasmussen

ECTS:
30

Supervisor(s):
Evangelos Boukas

Copies: 1

Page Numbers: 51

Report Hand-in:
June 3, 2019

Abstract:

This project investigates the possibilities of creating an autonomous robot deep reinforcement learning agent for maples navigation in novel environments. To train the agent a simulated environment is created with Unity, that applies the principles of domain randomization. An *Intrinsic Curiosity Module* (ICM) is implemented, to encourage the agent to explore unseen areas of the environment. A *Deep Recurrent Q-network* (DRQN) agent and a DRQN agent with an ICM are trained on a limited version of the environment to explore the system's potential. During the training the agents use the trick of *Learning from Easy Missions* along with the proposed trick of *Naive Visual Hindsight Experience Replay*. The trained agents show signs of improvement; however, the training is still in its early stages.

Institut for Materialer og Produktion
Aalborg Universitet
http://www.aau.dk

AALBORG UNIVERSITET
STUDENTERRAPPORT

Titel:
Map-less Navigation in Novel Environments using Deep Reinforcement Learning

Uddannelse:
Autonome Systemer

Projektperiode:
Forårssemestret 2019

Deltager(e):
Frederik Nørby Rasmussen

ECTS:
30

Vejleder(e):
Evangelos Boukas

Oplagstal: 1

Sidetal: 51

Afleveringsdato:
3. juni 2019

Abstract:

Dette projekt undersøger mulighederne for at udvikle autonome robot agenter der kan navigere i usete miljøer vha. *deep reinforcement learning*. For at træne agenterne udvikles der et simuleret miljø med Unity, som anvender principperne af *domain randomization*. For at anspore agenterne til at undersøge usete områder, implementeres der et *Intrinsic Curiosity Modul* (ICM). En *Deep Recurrent Q-network* (DRQN) agent og en DRQN agent med et ICM trænes i en indskrænket version af miljøet for at teste systemets potentiale. Under træningen udnytter agenterne tricket *Learning from Easy Missions* samt det foreslåede trick *Naive Visual Hindsight Experience Replay*. De trænede agenter viser tegn på forbedring, men træningsfasen er stadig i et tidligt stadie.

Nomenclature

Acronyms

CNN Convolutional Neural Network.

DQN Deep Q-network.

DRQN Deep Recurrent Q-network.

ELU Exponential Linear Unit.

HER Hindsight Experience Replay.

ICM Intrinsic Curiosity Module.

LEM Learning from Easy Missions.

LSTM Long Short-term Memory.

MDP Markov Decision Process.

NN Neural Network.

NVHER Naïve Visual Hindsight Experience Replay.

POMDP Partially Observable Markov Decision Process.

ReLU Rectified Linear Unit.

RL Reinforcement Learning.

RNN Recurrent Neural Network.

VHER Visual Hindsight Experience Replay.

Contents

Preface

This project has been an exciting, although very challenging, experience. There are definitely more interesting directions in which the project could be expanded, than the time constraints allow for now.

First of all, I would like to thank Dr. Tsampikos Kounalakis, for providing me with insights and helpful discussions along the whole semester. Likewise, I would also like to thank my supervisor Dr. Evangelos Boukas, for his guidance during the project. Finally, I would like to thank friends and family for reading through early drafts of the sections in the report, to provide a second opinion on grammar and style.

Aalborg University Copenhagen, June 3, 2019

Frederik Nørby Rasmussen
<afnr17@student.aau.dk>,
<frederiknoerby@outlook.dk>

Chapter 1

Introduction

The potential of robotics is immense, and on a long enough timeline, the tasks that robots or automated processes will not be able to solve, are perhaps mostly limited by imagination. Even in the foreseeable future, the impacts that robotics and automatic processes can have on society are enormous.

Robots are suitable of handling tasks that are exhausting or tedious to humans. Additionally, as technology improves, it can simply be cheaper to employ robots in jobs, previously fulfilled by humans. This development is already apparent in industries such as manufacturing, logistics, service, and agriculture. Even fields, such as law, banking, or medicine, which have classically employed highly trained individuals, have seen areas being overtaken by robotic process automation. However, robots and automation also come with the potential to take on tasks, which are unfeasible in different ways for humans to fulfill. Regarding physical robots, one of the advantages they have over humans, is the ability to be specifically built for a particular task or environment. This includes environments that are dangerous, or simply unreachable for humans, e.g. the depths of the oceans, the Arctic, outer space, waste facilities, etc. This could mean, that robots are to play a significant role in helping solve big global issues like climate change. Robots are and will also be used more in households. Robots that can accomplish household chores or assist people with disabilities, comes with the promise of freeing families and individuals to pursue activities they themselves find meaningful.

One of the challenges to creating intelligent autonomous robots, is enabling them with the ability to foresee the long-term consequences of actions. The research area of *Reinforcement Learning*, discussed more in later sections, is one of the most promising approaches to this problem. This project will focus on the task of navigation for mobile robots in unknown areas. This first chapter of the project, will serve as an introduction to the current work, as well as some of the challenges concerning the topic of the project.

1.1 Background and Motivation

Within the field of machine learning, Reinforcement Learning (RL) proposes a framework to solve tasks that involve sequential decision making. This sequential decision making is refered to as a Markov Decision Process (MDP) (Bellman, 1957a). The core concept of RL is to have an artificial agent interact with its environment and receive cumulative rewards. The interactions are discretized into timesteps, where at each timestep the environment presents itself in a given state that the agent can then take into consideration and base an action on. By performing this action, the environment will transition into the state at the next timestep and the agent will receive a corresponding reward to taking the given action at the given state. Using the experience from interactions, the agent should learn to avoid behavior that yields low cumulative rewards and instead strive to reinforce the behavior that leads to the highest cumulative rewards. Figure 1.1 shows an illustration of the agent-environment interaction in a MDP.

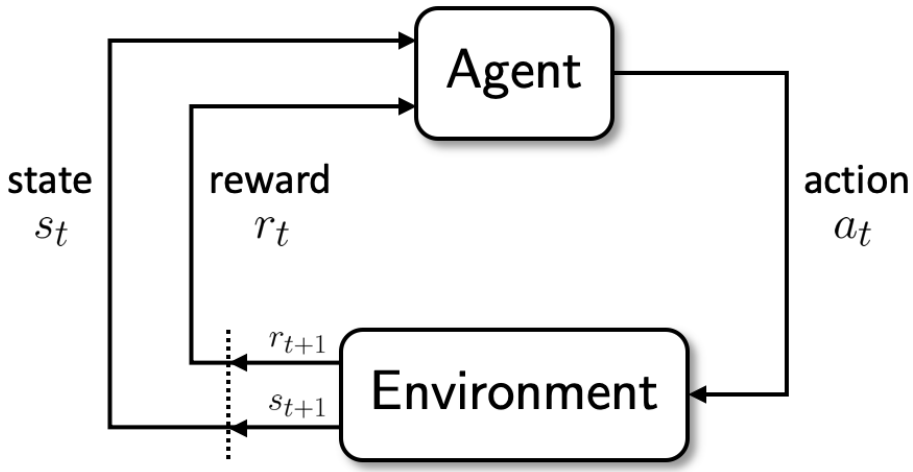


Figure 1.1: Illustration of the agent-environment interaction (based on figure from Sutton and Barto, 2018, p. 28). At each timestep t, the agent receives a state s_t and a reward r_t from the environment. The agent then takes an action a_t, which yields a new state s_{t+1} and reward r_{t+1} from the environment at the next timestep $t + 1$.

In recent years, the popularity of RL has increased massively, due to several successes using RL to solve demanding sequential decision-making tasks. E.g., playing the game of Backgammon at a level close to equaling the world's best human players (Tesauro, 1995), inverted flight of an autonomous helicopter (Ng et al., 2004), a robotic arm playing table tennis (Kober, Oztop, and Peters, 2011), playing *Atari* videogames at human level or above (Mnih et al., 2015), beating the world champion in the game of *Go* (Silver et al., 2017), defeating top professional players one versus one in the videogame of *StarCraft II* (Vinyals et al., 2019). Many of these

successes can also attribute part of the success to the use of deep learning techniques. Deep learning provides a way for the system to build complex concepts out of simpler concepts, by utilizing Neural Networks (NNs) for nonlinear function approximation (Goodfellow, Bengio, and Courville, 2016, p. 5). The combination of RL and deep learning has been given the term deep RL. Of the aforementioned recent successes using RL, many have been successes where the agent learns to master a discipline within a virtual environment or simulation. However, when the agent is a robot acting in the physical world, some additional challenges to implementing RL arise. The four major challenges, as branded by Kober, Bagnell, and Peters (2013), are: **the curse of dimensionality**, **the curse of real-world samples**, **the curse of under-modeling and model uncertainty**, **and the curse of goal specification**.

The curse of dimensionality (Bellman, 1957b, p. ix) refers to the enormous amount of possible state-action pairs, that occur if the state or action-spaces are high-dimensional or continuous. The curse of real-world samples refers to the difficulty of gathering data in the physical world. Because exploration is an important component of the learning process, implementing RL on robots in the real world can be a safety hazard and may potentially damage expensive equipment. Furthermore, gathering real-world data can also require complex setups of sensors and software to fully capture the relevant states. Due to these issues, when RL is deployed to the real world, the training of the agent is often done in simulation (Peng et al., 2017; Pinto et al., 2017; Tobin et al., 2017), or by using an initial policy (e.g., human controlled behavior) to improve upon (Levine et al., 2015; Smart and Kaelbling, 2002). The complications of using simulation to gather data, is what the curse of under-modeling and model uncertainty refers to. Creating a model that accurately depicts the robot and its environment well enough to generate useful data, is a challenging task. Moving from simulation to the physical world is often referred to as *bridging the reality gap*. The final curse, the one of goal specification, refers to the challenges of defining a good reward function that can help shape the robot towards the desired behavior.

1.2 Thesis Objective

The goal of this project is to create an autonomous two-wheeled robot RL agent, which can use its onboard camera to navigate novel environments with a map-less approach. Meaning that no map will be used or created by the agent, it will only have the possibility to learn from its own experiences. The sole input to the agent will be a stream of images, captured by the robot's onboard camera. Furthermore, the training of the model will be based only on generated data. This means that a simulated environment must be created, that can mimic the experiences of the robot when operating in the real world.

1.2.1 Research Questions

As a result of the completion of the thesis objective above, the following research questions may be answered:

1. **How can a sufficient simulated environment for training the model be created?** The created environment should not suffer from the curse of under-modeling and model uncertainty thereby allowing for bridging of the reality gap following training.

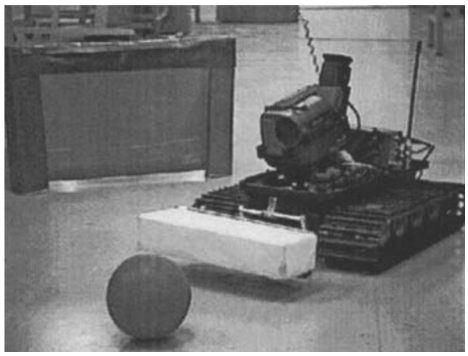
2. **What architecture is needed for the model to succeed?** The model should be able to take video frames as input and learn which actions to take based on this.

3. **How can a reward function be created to help shape the desired behavior?** To avoid the curse of goal specification, the rewards received by the agent, should encourage the desired behavior but should not be over-engineered.

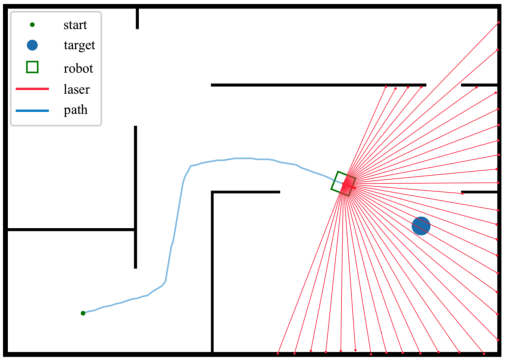
1.3 Related Work

In (Asada et al., 1996) the authors attempt to teach a mobile robot to score goals by driving a ball into a goal. They do this by using RL on observations, made by the robot's onboard camera. However, the camera frames are modified to fit one of 319 possible states, making the state-space much smaller, than using the actual camera frames as input. To deal with sparse rewards, the authors introduce Learning from Easy Missions (LEM) to the early stages of training. LEM refers to initializing episodes of training with the agent in a very favorable position to achieve the goal. Hereby maximizing the possibility that the agent, by random, choses the action(s) which allows it to get the reward for scoring. They also use simulation during training and end up with the physical robot scoring a goal in 60% of the test trials. Figure 1.2a shows a picture of the used robot together with the ball and goal.

As stated in section 1.1, an alternative approach to using simulation during training, is that of using an initial policy from the agent to build on. This approach was used by Smart and Kaelbling (2002) to train RL agents for mobile robots. In their paper, the authors divide the learning of the agent into two phases. In the first phase the agent is merely watching (i.e. gathering state-action samples), while having no control over the robot's actions. The robot is instead controlled by a supplied control policy, which can be either control code or a human teleoperating the robot. Once the agent has gathered enough samples to control the robot efficiently, the agent is given control of the robot to continue learning. The authors test their system with two simple tasks of corridor following and obstacle avoidance. Furthermore, the work dissociates it from this work, by not using an onboard

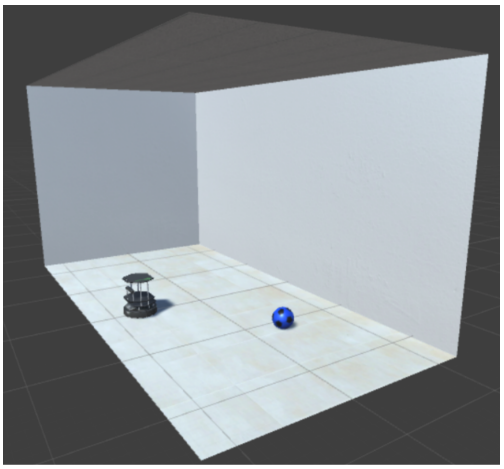


(a) A picture of the robot and ball in front of the goal from Asada et al. (1996).



(b) An example of the simple environment used for testing and training by Zhelo et al. (2018).

(c) Picture of the physical robot hand (top), and an example of a simulated hand (bottom) used by OpenAI et al. (2018)



(d) The simple rectangular room in which the simulated mobile robot and ball are spawned used by Lillelund (2018).

Figure 1.2: Examples of related work.

camera for observations. Another example of learning from an existing policy is the work by Henry et al. (2010). Here the authors use example traces of humans moving through crowded environments, to learn policies that can enable mobile robots to have human-like behavior, when performing the same task. However, the approach is only tested in a crowd motion simulator.

Examples where the training phase is done exclusively in simulation before being applied to a physical robot, is done by OpenAI et al. (2018) and Sadeghi and Levine (2016). In the work by OpenAI et al. (2018), the authors train a vision-based RL agent that can perform in-hand object reorientation through control of a humanoid robot hand. The agent is trained in a simulated environment using the MuJoCo[1] physics engine, while rendering training images with the Unity[2] game engine. To bridge the reality gap, the authors use the term domain randomization, which refers to randomizing different aspects of the simulated environment to enable the agent to generalize to the real world. After training, the agent is transfered to a physical robot hand, in which it successfully performs the task without any further training. In figure 1.2c, the physical robotic hand as well as one of the simulated hands used during training are pictured. In the work by Sadeghi and Levine (2016), an RL agent is trained for collision avoidance for drone flight. The training is likewise done exclusively through simulation of imagery with domain randomization, here generated using the 3D modeling suite Blender[3]. The learned model is applied to a real drone and fares better than a baseline model in a series of tests.

In the Master's thesis by Lillelund (2018), it was proposed that modern game engines could be used as realistic simulators for robotics. To prove their claim, The authors also used the game engine Unity to simulate a mobile robot, running a deep RL model in a simple environment. The simple environment consisted of a small rectangular room, in which only the robot and a ball would spawn (see figure 1.2d). The goal of the robot was then to use its camera feed to drive towards and hit the ball. However, the authors were not successful in having a physical mobile robot drive towards a similar looking ball, after transferring the model to it. This was concluded to be caused by the simulated environment not looking the same as the physical one.

A final related work is that of Zhelo et al. (2018). In this paper the authors use an RL agent, with an Intrinsic Curiosity Module (ICM), to tackle the problem of map-less navigation for mobile robots. Both training and testing is in this work exclusively done in a simple simulated environment. The simulated environment consists of a small maze-like map, in which the robot has to use laser sensors to navigate to a target position (see figure 1.2b).

[1] MuJoCo website: http://www.mujoco.org
[2] Unity website: https://unity3d.com
[3] Blender website: https://www.blender.org

CHAPTER 2

Theoretical Background

This chapter will deal with the underlying theory of the concepts, used in the rest of the project. The chapter will be comprised of two main sections. Section 2.1: *Deep Learning*, and section 2.2: *Reinforcement Learning*.

As stated in section 1.1, Reinforcement Learning (RL) is a subfield of machine learning, the same is true for deep learning. Machine learning refers to the field of study, that enables computers to learn without being explicitly programmed (Samuel, 1959). In traditional programming it is the job of the technician to provide the data and logic to yield the answers. In machine learning, the technician instead provides data with expected answers and then has the program learn a model that fits the examples.

2.1 Deep Learning

The term *deep learning* regards a system, that allow computers to learn from examples and gain an understanding of these, in terms of a hierarchy of concepts. In the hierarchy of concepts, the computer can learn complex representations, by defining them through their relation to simpler concepts. How these concepts relate to each other, can be drawn as a deep graph with many layers, hence the name deep learning (Goodfellow, Bengio, and Courville, 2016, ch. 1).

The feedforward Neural Network (NN) is the quintessential deep learning model. The goal of a feedforward NN, is to define a mapping $y = f(x;\theta)$, by learning the parameters, θ, which provide the best function approximation (Goodfellow, Bengio, and Courville, 2016, ch. 6).

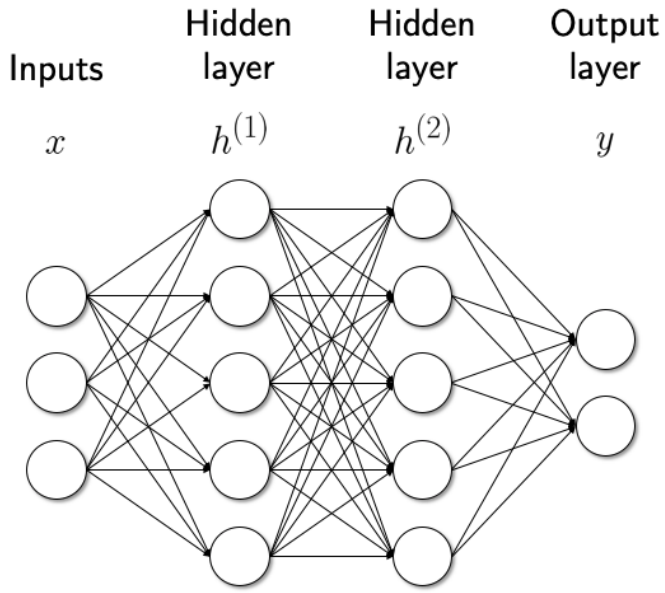


Figure 2.1: Example of a fully-connected feedforward Neural Network with two hidden layers.

2.1.1 Fully-connected Layers

The perhaps most common layer used in a feedforward NN, is the fully-connected layer. Figure 2.1 shows an example of a feedforward NN, with two hidden fully connected layers. The input layer represents the input values x, which is a column vector of size $n_x \times 1$ ($n_x \in \mathbb{N}$). The input values are then transformed by the first hidden layer. This is done through a matrix multiplication by $W^{(1)}$ of size $n_{h^{(1)}} \times n_x$ ($n_{h^{(1)}} \in \mathbb{N}$), plus a bias term $b^{(1)}$ of size $n_{h^{(1)}}$, followed by a transformation by the non-linear activation function $g^{(1)}$:

$$h^{(1)} = g^{(1)}(W^{(1)}x + b^{(1)}) \tag{2.1}$$

The output of the layer, $h^{(1)}$, will hereby be of size $n_{h^{(1)}} \times 1$. This output will then serve as the input to the next layer and follow the same transformation process:

$$h^{(2)} = g^{(2)}(W^{(2)}h^{(1)} + b^{(2)}) \tag{2.2}$$

This process will repeat itself, for as many hidden layers the network has, until the output layer, which outputs the values, y, in a similar fashion:

$$y = g^{(L)}(W^{(L)}h^{(L-1)} + b^{(L)}) \tag{2.3}$$

Where L is the number of layers the model has.

All of the layers are then trained to minimize the empirical error $I_S[f]$ on sample data. Usually, the parameters of a NN are updated based on gradient descent

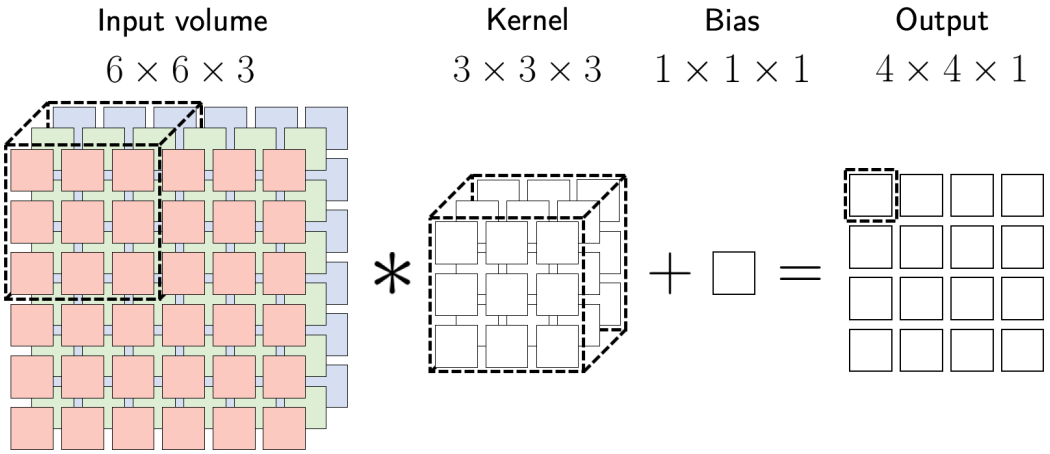


Figure 2.2: Example of a convolutional layer with an input of a $6 \times 6 \times 3$ RGB-image with a $3 \times 3 \times 3$ kernel with no padding and a stride of 1, resulting in a $4 \times 4 \times 1$ output volume. In the image, the enclosed area of the input volume is what will be convolved into the enclosed field of the output volume. This is done by multiplying the enclosed fields with the corresponding field in same position in the kernel. The results from all the multiplications are then added together into a single value, which then has the bias term added to it. The resulting value is then the value of the field in the output volume. The number of channels of the output volume is equal to the number of kernels in the layer, which, in the depicted example, is one

through the backpropagation algorithm (E. Rumelhart, E. Hinton, and J. Williams, 1986). In the simplest scenario, the algorithm changes its parameters θ to fit the desired function at each iteration (François-Lavet et al., 2018):

$$\theta \leftarrow \theta - \alpha \nabla_\theta I_S[f] \tag{2.4}$$

Where α represents the learning rate, i.e. the step size.

2.1.2 Convolutional Layers

A Convolutional Neural Network (CNN) (Lecun and Bengio, 1995) is a specialized feedforward network, which is particularly suited for processing images, or other data that has a grid-like arrangement (Goodfellow, Bengio, and Courville, 2016, ch. 9). When images are the input to a CNN, the first layers will learn to detect simple patterns, e.g. edges or textures, while following layers can learn more complex patterns, e.g. parts of objects (Erhan et al., 2009; Olah, Mordvintsev, and Schubert, 2017). CNN layers consist of kernels (or filters) and bias terms, which are both learnable. See figure 2.2 for an example of a convolutional layer. The input volume to a convolutional layer will have the dimension of $n_H^{(l-1)} \times n_W^{(l-1)} \times n_C^{(l-1)}$, respectively representing the height, width, and number of channels of the input. The kernels are then of size $k_H^{(l)} \times k_W^{(l)} \times n_C^{(l-1)}$. Notice that the number of channels

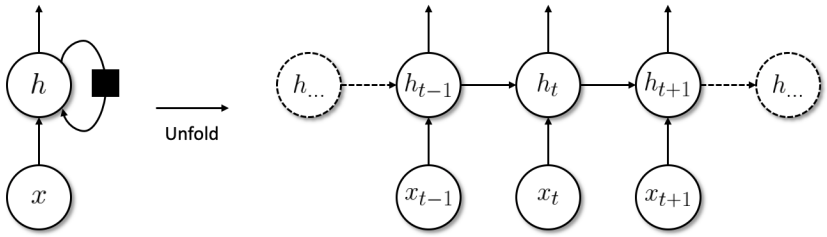


Figure 2.3: Illustrated part of a Recurrent Neural Network (based on figure from Goodfellow, Bengio, and Courville, 2016, ch. 10). In the left part of the illustration, the input x is passed into the recurrent layer. The output of the recurrent layer h, is both passed forward to the next layer, as well as being passed forward through time. Here the black box represents a delay of a single time step. In the right part, the same network is unfolded, and the nodes are associated with a particular time step.

match the input. The number of bias terms of the layer will be equal to the number of kernels used in the layer $n_C^{(l)}$. Having multiple kernels in the same layer means, that some of the kernels can learn to detect edges while others detect textures etc. The output of the layer will then be a volume of dimensions $n_H^{(l)} \times n_W^{(l)} \times n_C^{(l)}$. The height and width of the output layer is calculated by:

$$n_H^{(l)} = \frac{n_H^{(l-1)} - k_H^{(l)} + 2p}{s} + 1 \tag{2.5}$$

$$n_W^{(l)} = \frac{n_W^{(l-1)} - k_W^{(l)} + 2p}{s} + 1 \tag{2.6}$$

Where p represents the amount of *zero padding* and s represents the *stride* (Karpathy, 2015). Zero padding is a trick used to control the dimensions of the output volume. It works by adding rows and columns of zeroes around the input volume. This is most commonly done to retain the same height and width of the input in the output volume, without affecting the output values. Stride refers to the number of fields the kernel should skip, once it is done with a convolution. When the stride is one, the kernel moves one field at a time.

2.1.3 Recurrent Layers

A Recurrent Neural Network (RNN) is a network specialized for processing a sequence of data (Goodfellow, Bengio, and Courville, 2016, ch. 10). Recurrent layers not only take the current input at time step t into account, but also its own output from prior time steps. See figure 2.3. This project will utilize a special type of recurrent layer, called Long Short-term Memory (LSTM) (Hochreiter and Schmidhuber, 1997). The LSTM layer consists of a number of LSTM cells, depicted in figure 2.4. LSTM cells are connected recurrently, and on top of passing their output forward through time, they also pass a term called the *cell memory* forward through time.

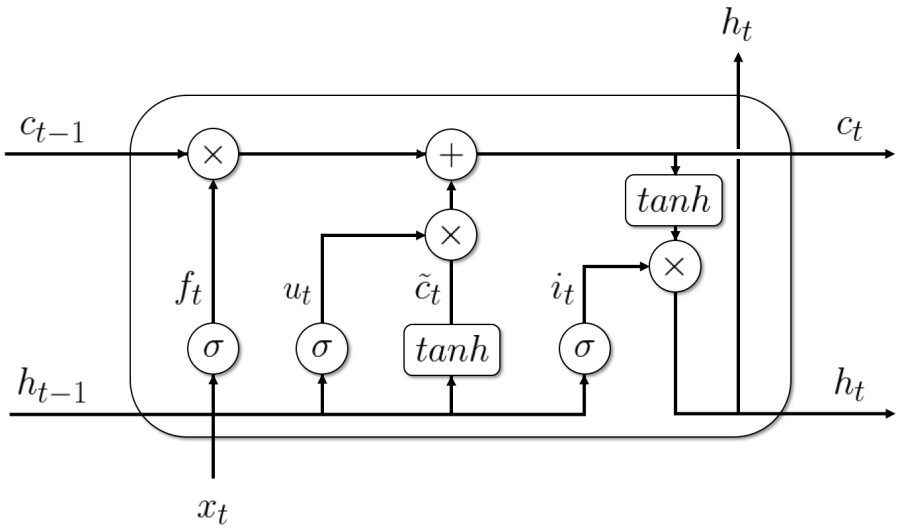


Figure 2.4: Illustration of a Long Short-term Memory cell (based on figure from Olah, 2015). LSTM cells are connected recurrently to each other, and pass forward through time the cell state c_t, and the layer output h_t. The output is also passed to the next layer of the network. In the illustration, the σ's represent the sigmoid function, while $tanh$'s represent the hyperbolic tangent function.

The cell memory stores and passes on information, that can be important at later time steps. Through a series of *gates*, the cell can remove and add information to the cell memory. The input to the gates, is the input at the current time step x_t, concatenated with the cell output in the prior time step h_{t-1}. The concatenated terms are multiplied by weight terms, then has bias terms added to them, before being put through the sigmoid function, which maps the function input in the range from 0 to 1. The gates are given by equations (2.7, 2.8, 2.11). Because of the output range from 0 to 1, the sigmoid function is in the LSTM cell, used to control how much of a term to keep or discard by element-wise multiplications.

$$f_t = \sigma(W_f[h_{t-1},\ x_t] + b_f) \tag{2.7}$$

$$u_t = \sigma(W_u[h_{t-1},\ x_t] + b_u) \tag{2.8}$$

$$\tilde{c}_t = tanh(W_c[h_{t-1},\ x_t] + b_c) \tag{2.9}$$

$$c_t = u_t * \tilde{c}_t + f_t * c_{t-1} \tag{2.10}$$

$$i_t = \sigma(W_i[h_{t-1},\ x_t] + b_i) \tag{2.11}$$

$$h_t = i_t * tanh(c_t) \tag{2.12}$$

The first of the gates is called the *forget* gate, f_t. The forget gate is multiplied element-wise, by the prior cell memory c_{t-1}, hereby determining what of the prior cell memory to keep, or forget. The next gate is the *update* gate, u_t. The update gate determines how much of the candidate cell memory $\tilde{c}_t$, is added to the cell memory. The candidate cell memory in (2.9) has, like the gates, learnable weight and bias terms, but uses the hyperbolic tangent function as a non-linear activation. The cell memory in (2.10), is then comprised of the remaining prior cell memory and candidate cell memory. The final gate, the *output* gate i_t, determines what of the new cell memory to output, after it has been through an activation function, typically the hyperbolic tangent, determining the output h_t in (2.12).

2.2 Reinforcement Learning

In the machine learning subfield of RL, it is the program itself that gathers the data and answers, to learn the rules by a series of interactions. Described in other words, with the commonly used RL terminology, the agent gathers experiences by taking actions, which yield rewards, and uses these experiences to form a policy for the most rewarding behavior in its environment. One of the factors that separates RL from the other machine learning subfields is consequently, that instead of trying to learn a hidden structure of the data, RL is simply trying to maximize a reward signal (Sutton and Barto, 2018, p. 2).

Because of the agent's desire to maximize the reward signal, the trade-off between exploration and exploitation arises exclusively in RL. On one hand, the agent has to take actions that exploit what it has already experienced to achieve reward. On the other hand, the agent has to take actions to explore better action selections for the future. Both exploration and exploitation must be pursued for the agent, to learn a good policy (Sutton and Barto, 2018, p. 2). The policy is what determines the agent's behavior. In other words, a policy maps the states perceived, by the agent, to actions.

As stated in section 1.1, the sequential decision making task, that a RL agent tries to solve, is called a Markov Decision Process (MDP) (Bellman, 1957a). In MDPs, actions not only impact immediate rewards, but also later states, and thereby also future rewards (Sutton and Barto, 2018, p. 47). Recall the illustration of the agent-environment interaction in figure 1.1 on page 2. The agent interacts with its environment over a series of discrete time steps. At every time step t, the agent is presented with a state $s_t \in \mathcal{S}$ of the environment. On the basis of the state, the agent uses its policy $\pi \in \Pi$, which is a stochastic rule, to select an action $a_t \in \mathcal{A}$. As a consequence of the action, the agent receives a reward r_{t+1} given by a reward function $R(s_t, a_t, s_{t+1}) \in \mathcal{R} \subset \mathbb{R}$, which it can use to evaluate the action,

and as a further consequence, the environment has transitioned to the next state s_{t+1}. The goal of the agent is then to find a policy, that maximizes the expected sum of rewards, i.e. the return. The expected return for starting in a state s, and subsequently following a policy π, is given by the value function $V^{\pi}(s)$ as:

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s, \pi\right] \tag{2.13}$$

Where γ represents a parameter called the discount rate. The discount rate's value has to be in the range: $0 \leq \gamma \leq 1$. The discount rate determines how much the agent will value future rewards in the present state. If $\gamma = 0$, the agent will only value immediate rewards. As γ approaches 1, the agent will become more farsighted, as future rewards will be valued higher. If $\gamma = 1$, future rewards will be valued as if they were received immediately (Sutton and Barto, 2018, pp. 54-55). This definition of expected return, allows for a definition of the optimal expected return $V^{*}(s)$ as:

$$V^{*}(s) = \max_{\pi \in \Pi} V^{\pi}(s) \tag{2.14}$$

Another commonly used value function in RL, is the action-value function, also called the Q-value function. The Q-value function is the expected return starting from state s, taking action a, and then following the policy π:

$$Q^{\pi}(s, a) = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| s_t = s, a_t = a, \pi\right] \tag{2.15}$$

Similarly, the optimal Q-value function $Q^{*}(s, a)$, can then be defined as:

$$Q^{*}(s, a) = \max_{\pi \in \Pi} Q^{\pi}(s, a) \tag{2.16}$$

To update the estimated Q-value for a given state-action pair, the Q-learning algorithm (Watkins, 1989) can be used. The update is defined by:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\Big[r_{t+1} + \gamma \max_{a} Q(s_{t+1}, a) - Q(s_t, a_t)\Big] \tag{2.17}$$

2.2.1 Partially Observable Problems

While MDPs provide a good formalism, in almost all real-world problems, the full state is not recognizable, and the agent only receives an observation $o_t \in \mathcal{O}$ at each time step. This is known as a Partially Observable Markov Decision Process (POMDP) (Sondik, 1978). See illustration in figure 2.5. The given scenario of this project is also a POMDP, as the agent only gets camera images as observations, and thus does not necessarily know where the target is placed, or the path to get there.

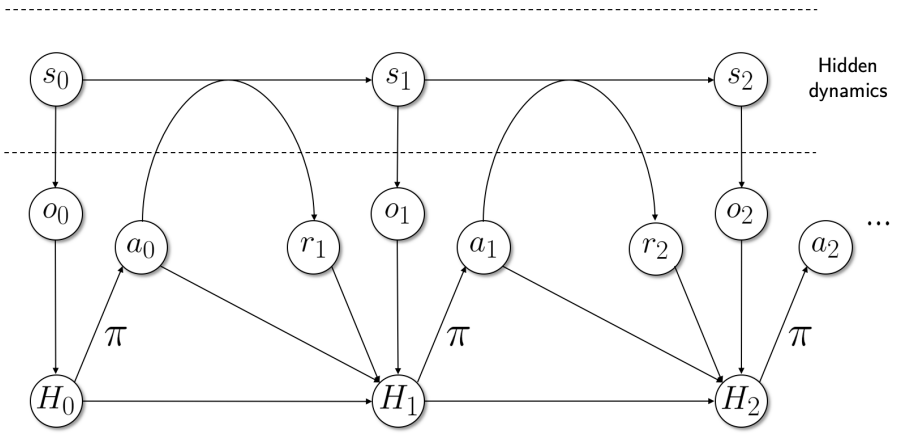


Figure 2.5: Illustration of a sequence of interactions in a Partially Observable Markov Decision Process (POMDP) (based on figure from François-Lavet et al., 2018). The actual dynamics of the POMDP are hidden, meaning the agent only gets observations of the environment, not the full state.

An approach to constructing policy mappings in a POMDP, is to simply use the last observation as a state, as if it were in a MDP. However, this will typically lead to policies that do not capture the system dynamics, as a single observation often does not contain enough information (François-Lavet et al., 2018). Alternatively, the agent can use a history, H_t, of previous observations to base its actions on. One approach is to have the history include all the events, up until time step t. However, increasing the input also increases the cost of computation and risk of overfitting (François-Lavet et al., 2018). An additional approach, is the use of function approximation. As stated in *Reinforcement Learning: An Introduction* by Sutton and Barto (2018, p. 197):

> "Perhaps surprisingly, extending reinforcement learning to function approximation also makes it applicable to partially observable problems, in which the full state is not available to the agent."

By utilizing function approximation, the value functions (2.13, 2.15) could as an example be computed by NNs, with θ representing the NN parameters.

2.2.2 Deep Q-networks

The Deep Q-network (DQN) algorithm (Mnih et al., 2015) was one of the first major success of applied deep RL. When published, the DQN algorithm surpassed all previous algorithms and achieved performance comparable to professional human game-testers, in a set of 49 *Atari 2600* games. The DQN algorithm utilizes a CNN for function approximation of the Q-value function, to learn rewarding behavior policies for each of the games. I.e., the parameters learned for the network were

Algorithm 1: Deep Q-learning with experience replay (Mnih et al., 2015)

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with random weights $\theta^- = \theta$
for *episode* $= 1, \ldots, M$ **do**
  Initialize episode
  for $t = 1, \ldots, T$ **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \arg\max_a Q(s_t, a; \theta)$
    Execute action a_t and observe reward r_{t+1} and next state s_{t+1}
    Store transition tuple $(s_t, a_t, r_{t+1}, s_{t+1})$ in D
    Sample random minibatch of transitions $\left(s_j, a_j, r_{j+1}, s_{j+1}\right)$ from D
    $$y_j = \begin{cases} r_{j+1} & \text{if episode ends at step } j+1 \\ r_{j+1} + \gamma \max_{a'} \hat{Q}(s_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$$
    Perform a gradient descent step on $\left(y_j - Q(s_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$

different from game to game, but the input, network architecture, and hyper parameters were the same across the collection of games. To accommodate partial observability in the games, the four most recent frames where stacked and used as input to the network. The output of the network was then made to represent the estimated Q-values, $Q(s, a; \theta)$, for each possible action in the state, where the state is represented by the input. The reward signal used for DQN, was based on the scores used in the individual games, where an increase in score would signal a reward +1, a decrease in score would signal -1, and an unchanged score would signal 0. This ensured that the reward signal was standardized across the suite of games. To accommodate the exploration versus exploitation dilemma, DQN takes advantage of a method termed *ε-greedy exploration* with linear decay. The ε-greedy exploration method simply means, that the agent will select the greedy action, i.e. the action with the highest estimated Q-value, with probability $(1 - \varepsilon)$, and select a random action with probability ε. The linear decay refers to the ε-greedy exploration value being $\varepsilon = 1$ at the beginning of training, and then decaying to a minimum of $\varepsilon = 0.1$ after one million steps. Meaning, that the agent will behave totally random in the beginning of training and then rely more and more on the learned Q-values, while still behaving randomly with a probability of 0.1 in the end.

When DQN selects an action, the action is executed to yield the reward r_{t+1}

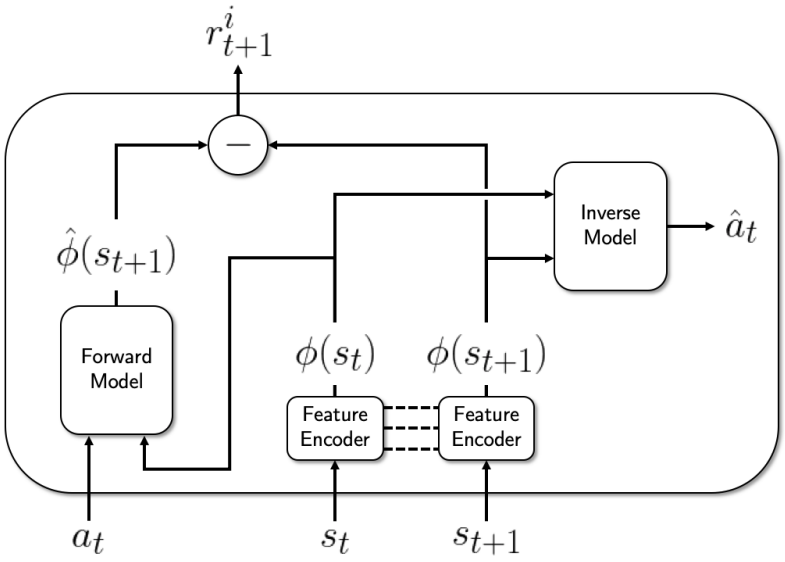


Figure 2.6: Illustration of an Intrinsic Curiosity Module (based on figure from Pathak et al., 2017). The inverse model to the right, takes the two encoded states $\phi(s_t)$ and $\phi(s_{t+1})$ as input and estimates the used action $\hat{a}_t$ to transition between the states. The forward model to the left, takes the action a_t and the encoded state $\phi(s_t)$ as input and estimates the encoding for the next state $\hat{\phi}(s_{t+1})$. The difference between the estimated encoding for the state $\hat{\phi}(s_{t+1})$ and the actual encoding for the state $\phi(s_{t+1})$ is used as intrinsic reward r^i_{t+1}.

and the next image frame, which is then stacked to form the next state s_{t+1}. The transition is then stored in the DQN's memory, D, as the tuble: $(s_t, a_t, r_{t+1}, s_{t+1})$, and s_{t+1} becomes the next input to the network. To update the network, DQN uses a method called *experience replay*. Experience replay means, that at each iteration a random minibatch of transitions $(s, a, r', s') \sim U(D)$ is drawn uniformly from the DQN's memory. The update at iteration i is then uses the following loss function:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r',s') \sim U(D)} \left[\left(r' + \gamma \max_{a'} \hat{Q}(s', a'; \theta_i^-) - Q(s, a; \theta_i) \right)^2 \right] \tag{2.18}$$

Where $\hat{Q}$ is a target network, that uses duplicate parameters, θ^-, of the main network's parameters, θ. The duplicates remain fixed for a certain number of updates, C, at which they are again overwritten by new duplicates of θ. The full DQN algorithm is presented in algorithm 1.

2.2.3 Intrinsic Curiosity Modules

In many scenarios, rewards are extremely sparse, and as mentioned in section 1.1, defining a good reward function that can help shape the agent towards desired behaviour can be challenging. One approach is to use an Intrinsic Curiosity Module (ICM) (Pathak et al., 2017), which supplements the standard reward signal by

adding an extra intrinsic reward signal. The intrinsic reward signal is supposed to help the agent seek novel situations by giving a positive signal when it encounters unexpected scenarios. In other words, the intrinsic reward signal is the prediction error of the agent's knowledge about its environment. To make predictions, the agent will utilize two NNs. An illustration of an ICM is presented in figure 2.6. The first, known as the inverse dynamics model, consists of two submodules. The first submodule takes a state s_t as input, and encodes it into a feature vector $\phi(s_t)$. The second submodule takes two subsequent feature encodings $\phi(s_t)$, $\phi(s_{t+1})$ and outputs an estimated action $\hat{a}_t$ that the agent used to move from s_t to s_{t+1}. Training the network then amounts to learning the function g_I:

$$\hat{a}_t = g_I\Big(s_t, s_{t+1}; \theta_I\Big) \tag{2.19}$$

Where θ_I are the network parameters. To update the network parameters, the error between the predicted and actual actions must be minimized.

Furthermore, an additional NN, known as the forward dynamics model, is trained. The additional network takes an action a_t, and a feature vector $\phi(s_t)$ as inputs, and use these to predict an estimated feature encoding of the subsequent state $\hat{\phi}(s_{t+1})$. Training the network then amounts to learning the function g_F:

$$\hat{\phi}(s_{t+1}) = g_F\Big(\phi(s_t), a_t; \theta_F\Big) \tag{2.20}$$

Where θ_F represents the network parameters. To update the parameters, the following loss function must be minimized:

$$L_F\Big(\hat{\phi}(s_{t+1}), \phi(s_{t+1})\Big) = \frac{1}{2}||\hat{\phi}(s_{t+1}) - \phi(s_{t+1})||_2^2 \tag{2.21}$$

The intrinsic reward signal r^i, is then computed as:

$$r^i_{t+1} = \frac{\eta}{2}||\hat{\phi}(s_{t+1}) - \phi(s_{t+1})||_2^2 \tag{2.22}$$

Where η represents a scaling factor $\eta > 0$.

CHAPTER 3

Methodology

As stated in chapter 1, the goal of this report is to create an autonomous two-wheeled robot RL agent, which can use its onboard camera to navigate novel environments with a map-less approach. The robot used in this report, is a Turtlebot2[1] with an onboard Asus Xtion Pro Live camera. This chapter will consider the development and implementation of the created system. The first section of the chapter, will regard the simulated environment, created for training the RL agent. The following section will then involve the development and architecture of the RL model.

3.1 The Simulated Environment

As stated in section 1.1, because exploration is an important component of the RL process, implementing RL on physical robots can be extremely expensive or even cause safety hazards. To circumvent this, this project will use the approach in which the RL agent gathers experience and learns its model based on simulated data. In the scenario of this project, the robot uses its onboard camera to gather observations of the environment's state. It is therefore the image data from the camera, which will have to be simulated. Even though the Turtlebot has an onboard computational device, the RL model will run from an external computer. This will allow for more computation, as well as making the process of testing and debugging more transparent and easier. For this to work, communication between the robot and the external computer must be established. To implement the communication, the middleware ROS (Robot Operating System), in the Kinetic Kame release[2] (Open Source Robotics Foundation, 2016), will be used. ROS is a collection of tools, libraries, and conventions, and it is commonly used by both academia

[1] Turtlebot2 information at: https://www.turtlebot.com/turtlebot2/

[2] Available at: http://wiki.ros.org/kinetic

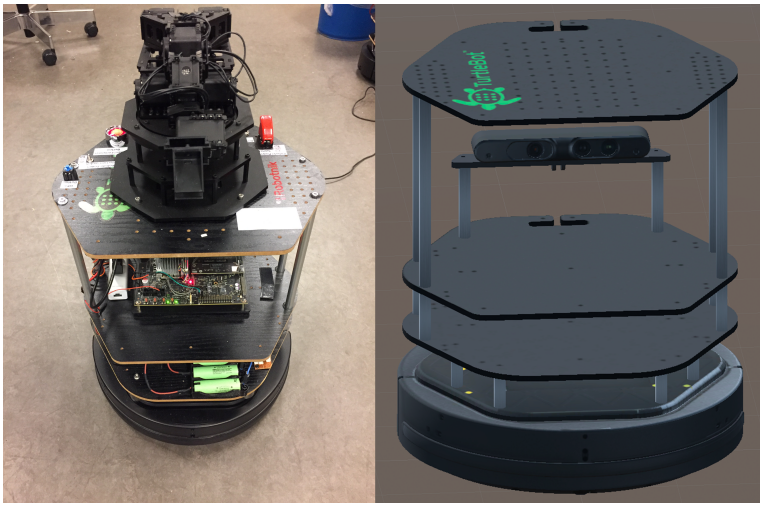


Figure 3.1: Comparison between the physical robot (left) and the simulated version (right).

and industry for research and commercial use. In this project, ROS takes care of distributing information between entities of the system, such as providing the RL model with the image frames, captured by the robot's onboard camera.

As the ambition is to learn a model from simulated data, which can also work in a real environment, the simulated data will have to resemble real data closely enough for the agent to generalize. In other words, the images generated through simulation should, to the agent, not appear very different, from images captured by the physical camera. The simulated data therefore requires software that has the capability of rendering images. Within the ROS framework the 3D dynamics simulation software Gazebo[3] is included. Yet, Gazebo does not directly allow for photo-realistic rendering. As the robot will rely solely on its camera, as a perception to its surroundings, the camera feed generated in simulation should have the possibility of emulating images captured in real life as close as possible. An area that has seen rapid development in terms of photo-realistic rendering in recent years is video games. To benefit from this, the game-engine Unity[4] (Unity Technologies, 2018) will be used to create the simulated environment. With help from the open source library ROS# (Bischoff, 2018), communication between ROS and Unity is available. In figure 3.1 a side-by-side comparison between the physical robot and its simulated counterpart is shown.

The environment in which the physical robot will operate in, is an indoor office-like environment at Aalborg University Copenhagen, the simulated environment will therefore have to reflect this. One approach to creating the simulated environment in which the agent has to train, could therefore be to make an exact replica of

[3]Gazebo website: http://gazebosim.org
[4]Available at: https://unity3d.com/unity

Figure 3.2: Examples of environments with domain randomization.

the physical environment with the same sizes, colors, textures, furniture, lighting conditions, etc. However, getting all these elements correct in simulation would be very time-consuming, and furthermore, the agent would only learn how to behave in these specific conditions. Another approach is that of domain randomization (Sadeghi and Levine, 2016; Tobin et al., 2017), which has proven to be a promising technique, when transferring from images created in simulation to real images, or in other words, bridging the so-called reality gap.

For this project, the goal of the agent will be to locate a target object located somewhere in the environment. The training will be conducted in an episodic manner, where the episode will end if one of the following conditions are true:

1. The robot has moved into the position of the target.
2. The robot has crashed into one of the walls or obstacles.
3. The robot has used more than a prespecified number of steps without solving the task.

3.1.1 Domain Randomization

The concept behind domain randomization (Tobin et al., 2017), is to randomize different aspects of the environment, which the agent encounters during training.

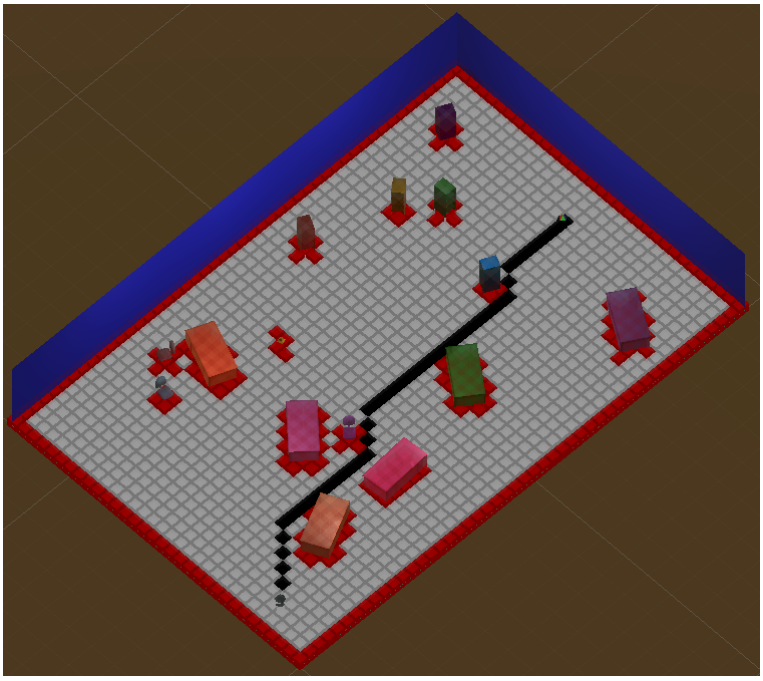


Figure 3.3: Examples of the A* algorithm finding the shortest path between the robot and the target object. The path between the agent and the target object is marked with black tiles. Red tiles are blocked.

By randomizing these different aspects of the environment, the hope is that the agent will have experienced enough different versions of the environment, that it will only give importance to the aspects that actually matter for it to achieve its goal. Then, when the agent is transferred to the real world, the changes should similarly appear to be yet another simulated variation, allowing for the agent to generalize. Examples of this approach proving successful:

1. Tobin et al. (2017) used domain randomization to train an object detector in simulation, that was then applied to have a physical robot arm pick up the detected objects.

2. Sadeghi and Levine (2016) used domain randomization to train an agent for collision avoidance in drone flight.

3. Peng et al. (2017) used domain randomization with the focus of randomizing the dynamics, to have a robotic arm push an object into goal positions.

The randomization of the environment will take place before the beginning of each episode, meaning that the randomized aspects will remain unchanged for the duration of the episode. To elaborate on the point made above, when the agent relies on the real-world images, they should simply appear to reflect yet

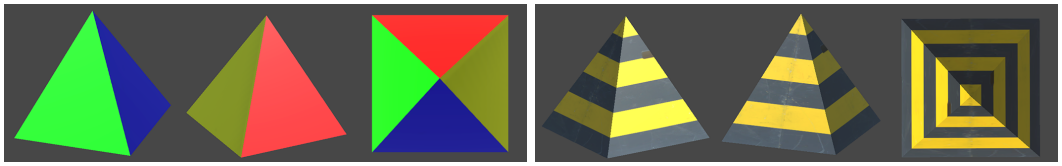
another randomized episode of training. The following aspects of the domain are randomized at each episode of training:

- The size of the generated map.
- Number of rooms in the generated map.
- Sizes of the generated rooms.
- Lighting direction and color.
- Furniture types, placements, and colors.
- Placement and size of doorways.
- Textures and colors of walls, floors, and ceilings.
- Starting location of the robot.
- Location of the target object.

Examples of generated environments can be seen in figure 3.2.

While randomization of the dynamics used by the simulator has been shown to also be useful (Peng et al., 2017), it is excluded for now, to limit the complexity of learning a model. However, it could be included as further development, as it could help create a more robust model.

To generate the rooms in the environment, a procedure for generating unconstrained rectangles within a rectangle from Valenzuela and Wang (2001) is followed. The generation is nonetheless not completely unconstrained, as the rooms have to be big enough to fit the robot. After the rooms have been created doorways are added. Since the rooms generated are all rectangle, the doorways have a chance of covering the whole wall between two rooms. By allowing for this, there is a possibility of creating rooms that have more than four corners. Once the walls and doorways have been added to the environment, the robot, target object, and furniture is added. To guarantee that the environment created for each episode is solvable, i.e. the robot has an unblocked pathway from its spawning point to the goal object, the *A* algorithm* (Hart, Nilsson, and Raphael, 1968) is implemented. The A* algorithm finds the shortest path around the obstacles from the robot to the target object avoiding obstacles, as shown in figure 3.3. If the algorithm fails to find a path, a new environment is generated. The target object chosen for this project is a small pyramid roughly the same height as the robot. As shown in figure 3.4a, the pyramid has four sides each colored in a solid color. The attributes of the pyramid are chosen to make it easy to recreate it from cardboard in the real-world. To encourage the RL agent towards only searching for this specific target, and not just any pyramidic shaped object, an extra pyramid of the same dimensions is added to the simulated environment. The other pyramid is also randomly



(a) The target pyramid seen from sides and above.

(b) The obstacle pyramid seen from sides and above.

Figure 3.4: The simulated pyramids.

Algorithm 2: Episodic generation of environment

```
while Solvable is false do
    Randomly initialize map height H
    Randomly initialize map width W
    Create empty list of rooms L
    Add room of size H × W to L
    Randomly initialize desired number of rooms n
    while L length < n do
        Choose a room R randomly from L
        Choose vertical or horizontal slizing direction randomly
        Choose a random position to cut R in the chosen direction
        Perform the cut, generating two sub-rooms R_1 and R_2
        Replace R with R_1 and R_2 in L
    Randomly place doorways between rooms
    Randomly place robot
    Randomly place target
    Randomly place obstacles
    Use A* to find path between robot and target
    if Path found then
        Environment = true
```

placed at the beginning of each episode and will be considered an obstacle on the same terms as the furniture. The extra pyramid is only distinguishable from the target pyramid by its textures, as displayed in figure 3.4b. In algorithm 2, there is a walkthrough of the procedure of generating the environments.

3.2 The Reinforcement Learning Implementation

This section will regard the choices made for the implementation of the RL agent. The model learned from the interactions between the RL agent and the environment, will be what determines the agent's policy and decides the robot's behavior.

The model will have to map the agent's sensory input, i.e. the visual input from the robot's onboard camera, to the motor output of the robot's two wheels.

3.2.1 ROS Messaging

As stated earlier, ROS will handle the communication between entities of the system. In the given scenario, ROS will therefore need to send images captured by the robot to the computer running the RL model and send messages for driving the wheels from the computer back to the robot. Fortunately, ROS is built to handle this and has several predefined message types available for communication. The communication in ROS, revolves around the concept of publishers and subscribers. To send and receive messages in ROS, first a *ROS Master* must be set up. The ROS Master provides information to the rest of the nodes in the system, about which information is currently available. To send information, a topic and a message type must be defined first. Then, publishers can publish messages to the topic and subscribers can subscribe to the topic to receive the published messages. The message type used to send the images, is in ROS called *compressed image* and are contained in the sensor messages category. The image data is sent as an array of 8-bit unsigned integers. 8-bit unsigned integers can represent values ranging from 0 to 255, which incidentally matches the red, green, and blue pixel values in an RGB image. The message type used for sending movement commands to the robot, are called *twist* messages and are contained in the geometry messages category. The twist messages consist of two three dimensional vectors representing velocity, with one representing linear velocity and the other representing angular velocity. As the Turtlebot only has two wheels that can rotate around one axis, only the linear velocity along the x-axis and the angular velocity along the z-axis are relevant, the rest of the velocity values will be zero. The twist messages will be published to the *cmd_vel* (command velocity) topic, which a hardware driver on the robot subscribes to. The hardware driver will then translate the velocities into a speed for each of the robot's two wheels respectively. In addition to the twist and compressed image messages, this project will also use messages of type string from the category of standard messages. The string message type simply represents a sequence of characters, as it conventionally does in programming languages. In this project, the string messages are used to send information about the reward the agent receives and if an episode is done by fulfilling one of the conditions listed in section 3.1.1.

3.2.2 Defining the Reinforcement Learning Problem

As stated in section 3.2.1, only a value for linear velocity along the x-axis and a value for angular velocity along the z-axis are needed to drive the robot. The output of the RL model should therefore determine these values. In other words, the

Table 3.1: Twist message values for the discretised action space

Linear velocity along x-axis	Angular velocity along z-axis	Outcome
0.0	0.0	Robot is stopped
1.0	0.0	Robot drives straight forward
-1.0	0.0	Robot drives straight backward
0.0	1.0	Robot rotates counterclockwise
0.0	-1.0	Robot rotates clockwise
1.0	1.0	Robot drives forward to the left
1.0	-1.0	Robot drives forward to the right
-1.0	1.0	Robot drives backward to the right
-1.0	-1.0	Robot drives backward to the left

two values together represent the action chosen by the agent at a given timestep, and it is the job of the model to determine this action. One approach to determine the action could simply be to have the model directly output two values, which represent the linear and angular velocities respectively. Another approach, could be to make the model estimate the value of actions, as given by the Q-value function in (2.15). The first approach is known as a *policy gradient method*, because the model directly maps the state to the estimated optimal action, thereby acting as a policy. The second approach is known as an *action-value method*, since the model maps the state to estimated values for performing the possible actions (Sutton and Barto, 2018, p. 321). However, the action-value methods immediately come with a caveat. Because the action space is continuous, it is not possible for the model to output an estimated value for each possible action. A solution to this, could be to discretize the action space. However, by doing so, the term coined the *curse of dimensionality* by Bellman (1957b) arises. For example, if the linear velocity along the x-axis were discretized from -1 to 1 in steps of 0.1 and the angular velocity around the z-axis were discretized from -1 to 1 in steps of 0.1, then the total number of actions would be 400. Policy gradient methods are capable of handling continuous action spaces (Sutton and Barto, 2018, p. 335), but come with their own disadvantages. As stated by OpenAI et al. (2017):

> "... getting good results via policy gradient methods is challenging because they are sensitive to the choice of stepsize — too small, and progress is hopelessly slow; too large and the signal is overwhelmed by the noise, or one might see catastrophic drops in performance. They also often have very poor sample efficiency, taking millions (or billions) of timesteps to learn simple tasks."

Given the time constraints of this project, the action-value method Deep Q-network (DQN), presented in section 2.2.2, is used. However, one simple change will be made. Instead of resetting the target Q-network's parameters, θ^-, to the Q-network's parameters, θ, every C steps, the θ^- parameters will instead be updated

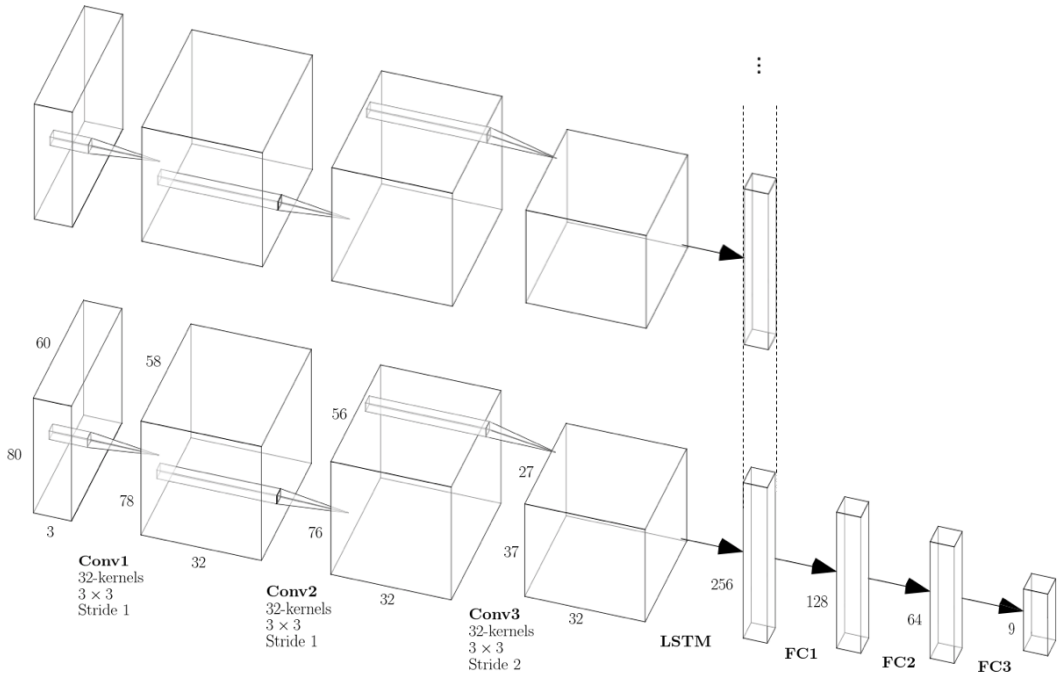


Figure 3.5: Illustration of the used Deep Recurrent Q-network model architecture. The Long Short-term Memory layer takes the output from the convolutional layers and is unrolled over the last 16 timesteps.

a small step towards θ at each timestep, as given by formula 3.1. This was shown by Lillicrap et al. (2016) to improve the stability of learning.

$$\theta^- \leftarrow \tau\theta + (1-\tau)\theta^- \tag{3.1}$$

Where $\tau \ll 1$.

The DQN algorithm is highly sample efficient, because it can use the same sample to update its parameters, θ, multiple times through experience replay. However, if more time were available, a policy gradient method might fit the scenario better. Using the DQN algorithm comes with the consequence of discretizing the action space. To avoid having a very large action space, which would require an abundance of exploration from the agent, the possible actions are limited to nine options presented in table 3.1.

3.2.3 The Model Architecture

Now both the input and the output of the model is defined. The input is images and the output is a column vector with estimated Q-values for each possible action. The type of agent used will be the DQN presented in 2.2.2. However, there

is a missing piece to the puzzle. As it is, the DQN maps observations to a Q-value for each possible action. However, as observations simply are images, the environment is only partially observable and there is nothing in the observations to indicate time. Therefore, the agent will have nothing stored from timestep to timestep, which suggests what has happened so far during the episode. In other words, the observations are disassociated, and the agent merely classifies an observation to one of the nine actions. This will prove a problem, as the environment and target object placement are randomly generated. The agent therefore needs to take into consideration, which areas it has already searched for the target object. This is where recurrent layers, as presented in section 2.1.3, prove useful. Adding recurrence to the DQN where shown by Hausknecht and Stone (2015) to help better estimate the underlying state, s, of an environment by an observation, o. The recurrent version of the DQN is simply known as a Deep Recurrent Q-network (DRQN) (Hausknecht and Stone, 2015). With the introduction of the recurrent layer, all the building blocks for the model architecture are now present. The model, presented in figure 3.5, will take RGB images of size 60×80 as input. Even though the onboard camera is capable of taking pictures of size 480×640, the images are drastically downsized in a measure to reduce training time, given the time constraints of this project. In figure 3.6 a comparison between a 480×640 image and the downsized 60×80 version is shown, which shows that it is still possible to make out the features of the image, although some of the finer details are lost. The input image is processed by three convolutional layers, as done by Mnih et al. (2015); Hausknecht and Stone (2015), before being fed to a Long Short-term Memory (LSTM) layer, presented in 2.1.3. Then, three fully-connected layers are used, presented in section 2.1.1, to finally give the output of nine Q-values. All layers except the LSTM and final fully-connected layers use the Rectified Linear Unit (ReLU) (Nair and Hinton, 2010) function as non-linear activations. The LSTM layer uses the hyperbolic tangent as a non-linearity whereas the final fully-connected layer simply uses the identity function. The LSTM layer will use the last 16 timesteps to determine the output, making the network a recurrent network of type *many-to-one*, because it uses multiple input observations to determine one action at every timestep.

3.2.4 Dealing with Sparse Rewards

As stated in section 1.1, the curse of goal specification refers to the challenges of defining a good reward function, that can help shape the agent towards the desired behavior. As the reward function must be manually programmed, the problem arises when designing and implementing the reward function develops into a task, that approaches the complexity of manually designing the agent's policy, which is what RL promises to avoid. This project will therefore keep the reward function simple and instead test other strategies to help shape the desired behavior.

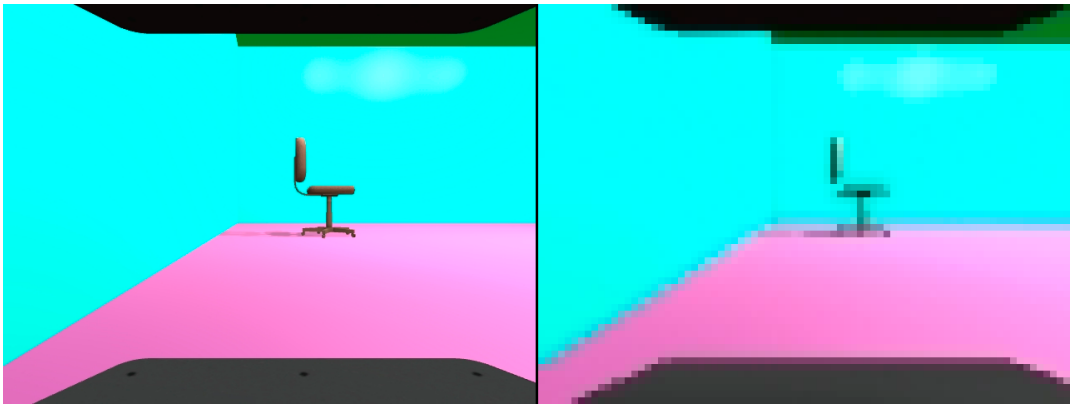


Figure 3.6: Comparison between the fullsize 480×640 image (left) and the downsized 60×80 image (right).

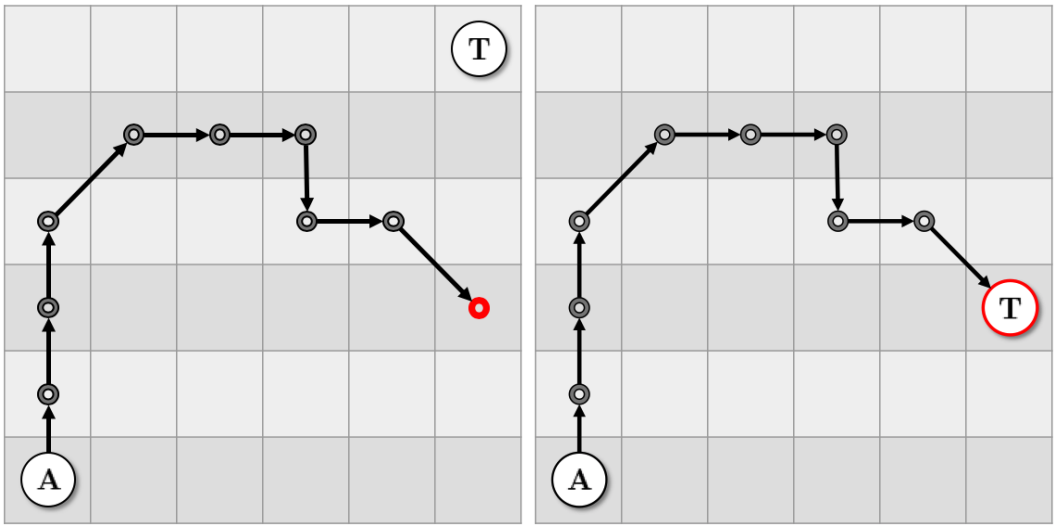
In the scenario given by this project, the agent will only receive positive reinforcement, if it drives into the target object. This means that the rewards in the given scenario are extremely sparse. Furthermore, the agent will receive a penalty, i.e. a negative reward, if the agent crashes into the walls or obstacles in the environment. Additionally, one extra feature is added to the reward function, at every timestep the agent will receive a small *step penalty*. The step penalty is put in place to encourage the agent to solve the task as fast as possible, i.e. if the agent solves the task in two different episodes, then the episode where the agent used the fewest steps will have a greater cumulative reward.

As the task at hand, is for the agent to locate the target object, the behavior which should be encouraged, is one where the agent actively tries to discover unseen areas of the environment, until it locates the target. To encourage this, the Intrinsic Curiosity Module (ICM), presented in section 2.2.3, is implemented. The ICM grants the agent with intrinsic rewards, that increase the more unforeseen the consequences of the agent's actions are. When the agent encounters unseen areas of the environment, it is much harder to predict what the next observation will be, hence the agent will receive a bigger intrinsic reward. The agent will also still receive the normal rewards given by the environment, to distinguish, these will be referred to as extrinsic rewards. An illustration of the ICM is found in figure 2.6 on page 16. To map the observations into feature vectors $\phi(o_t)$, the ICM uses a series of four convolutional layers, each with 32 kernels of kernel size 3×3 and a stride of 2. Exponential Linear Unit (ELU) (Clevert, Unterthiner, and Hochreiter, 2016) non-linearity is used after the convolutional layers as done by Pathak et al. (2017). The dimensionality of $\phi(o_t)$ will be 256. For the inverse model part of the ICM, the two feature vectors $\phi(o_t)$ and $\phi(o_{t+1})$ are concatenated and fed as input to a fully-connected layer with 256 units and the ReLU non-linearity. This is followed by another fully-connected layer with nine units, that uses the Softmax as activation

function. The output can thus be interpreted as probabilities for which of the nine actions the model estimates the agent has taken. For the forward model part of the ICM, the feature vector $\phi(o_t)$, and the actual action taken at the timestep, a_t, are concatenated and fed through two fully-connected layers with 256 units. As the forward model part of the ICM tries to estimate the feature vector $\phi(o_{t+1})$, the output of the fully-connected layers has to have the same dimensionality as the feature vector.

A problem that may occur in scenarios with sparse rewards, is that the agent only rarely, or even never, encounters positive rewards to learn from. To combat this problem, two additional tricks will be used during training. The first trick is known as Learning from Easy Missions (LEM) (Asada et al., 1996), as mentioned in section 1.3. The concept behind LEM, is to start the training phase with episodes where the agent is in a favorable position to solve its task. This maximizes the possibility that the agent, by random, will solve its task and receive the positive reward, even though it has not yet learned anything about the environment. In this project, the target will be placed in the same room as the agent at the beginning of the episode, when LEM is active. Thus, making it more likely that the agent will randomly drive into the target.

The second trick that will be implemented to combat sparse rewards is Hindsight Experience Replay (HER) (Andrychowicz et al., 2017). HER makes an important assumption of the state, which is that the task or goal of the episode is a part of it. The concept behind HER is to store alternative *memories* in the agent's replay buffer. In addition to the actual transitions encountered during an episode, HER also stores transitions where the goal of the episode is changed to be the outcome of the episode. E.g. if the goal of the agent was to end up in position x but it ended up in position y, then in the alternative transitions, the goal of the episode would be changed to ending up in position y, thus creating a memory of a successful episode. As it is, HER is not applicable to the scenario of this project. In this project, the goal is not part of the observations the agent receives. Furthermore, it would require a very complex implementation to change the transitions when creating altered memories, i.e., since the observations are images, the target object would have to be edited into the agent's end position for all observations of the episode. With Visual Hindsight Experience Replay (VHER) Sahni et al. (2019) created a model, that takes the approach of editing the goal into images from failed trajectories. This is done by training a generative adversarial network on observations of the goal, where its relative location to the agent is explicitly known. However, this project will propose a much simpler version of VHER, which will be refered to as Naïve Visual Hindsight Experience Replay (NVHER). Instead of editing encountered observations, the NVHER approach is simply to replay an unsuccessful episode with some minor alterations. The agent and environment will be initialized in the same way as the unsuccessful episode, except for the target



(a) *Episode* i: the agent performs a sequence of nine actions but fails to locate the target within the step limit. Consequently, the agent does not encounter any positive rewards during the episode.

(b) *Episode* $i+1$: the environment is initialized with everything as in the beginning of episode i except the target which is placed in the agent's end position from episode i. The agent is made to redo the same sequence of actions as it did in episode i, thus ending up in the position of the target, thereby encountering a positive reward.

Figure 3.7: Illustration of Naïve Visual Hindsight Experience Replay. The agent is represented by **A** and the target object is represented by **T**. The arrows represent the actions taken by the agent at each timestep, while the dark gray circles mark the position of the agent after taking an action. The red circle is the position of the agent at the end of the episode.

object which will be placed at the agent's end position from the previous episode. Furthermore, instead of following a policy, the agent will be made to repeat the exact same sequence of actions, as it did in the unsuccessful episode. By doing so, the agent will end up in the same position as the target object and all the observation images taken during the episode, will have the target object in the correct positions. See figure 3.7 for an illustration of NVHER.

3.3 The Training Set Up

To test the potential of the developed system, two agent types are trained on a limited version of the environment. The first agent type will be a DRQN agent, whereas the second will be a curious DRQN agent, i.e. a DRQN agent with an ICM. Both agents will be trained with both LEM and NVHER implemented. The

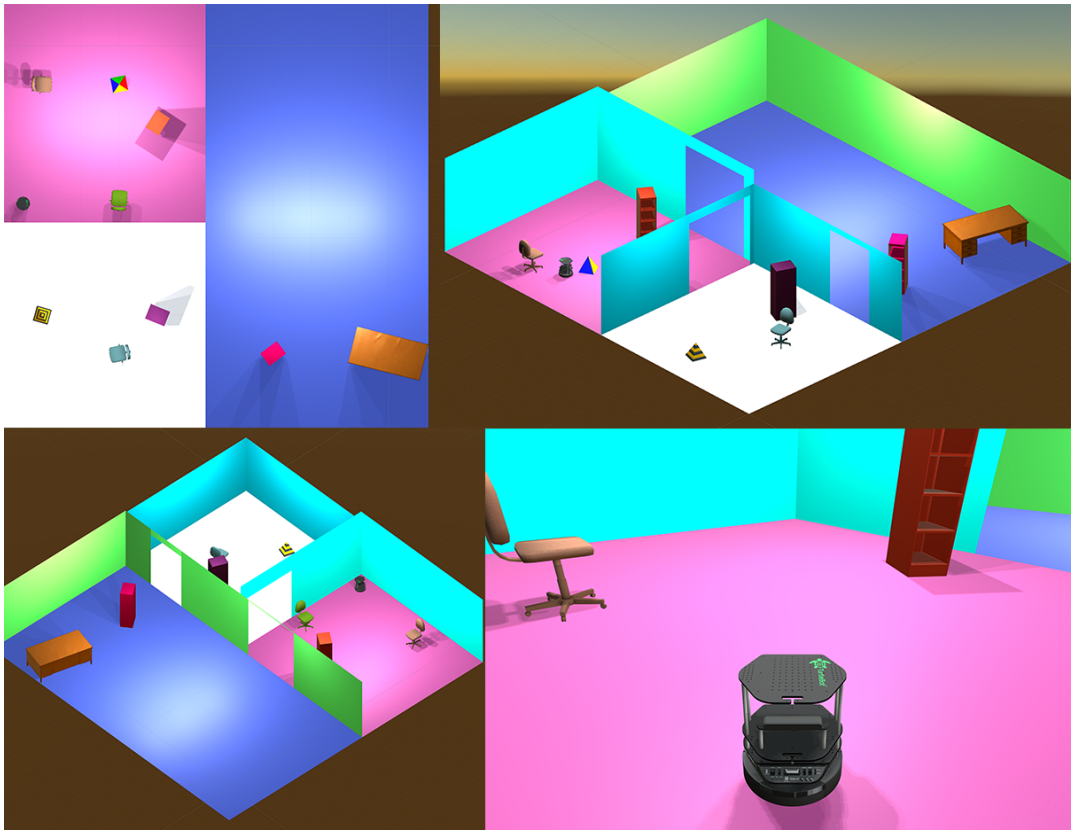


Figure 3.8: Different views of the limited test environment.

limited version of the environment consists of three connected rooms, with obstacles scattered across them. In the limited environment, the only aspect changing from episode to episode is the location of the target object, i.e. colors, lighting conditions, obstacle placements, agent starting location, map size, number of rooms, doorways, and textures do not change between episodes. See figure 3.8 for a picture of the environment. The environment is limited in these ways, to test the potential of the developed system as the time constraints of the project, will not allow for training agents in the full version of the environment. However, the limited environment used is chosen because it represents the challenges of the full environment well, i.e., as the environment consists of three rooms and the target object is randomly placed, the agent still has to explore unseen areas until it locates the target on an episodic basis. However, agents trained in the limited environment will have no grounds for deployment in a real-world environment, as the domain randomization is non-present, and the agents therefore will be specialized to the features of the limited environment.

Even though the reward function is very simple, some choices still have to be

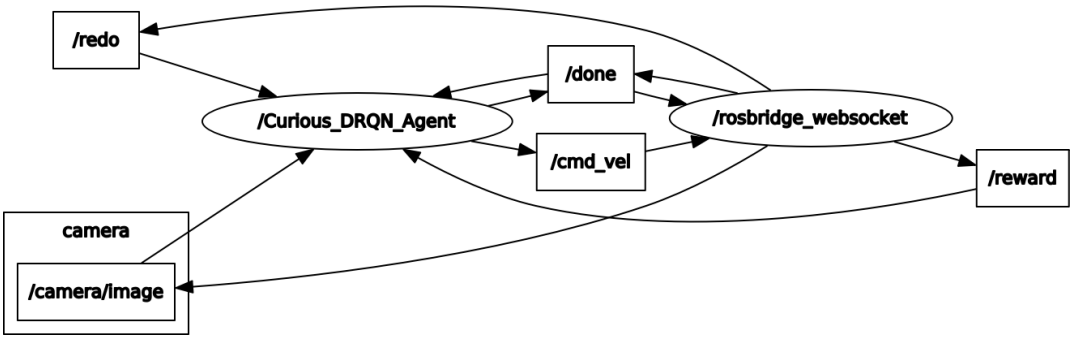


Figure 3.9: Graph showing the relationships between the ROS nodes during the training of the curious Deep Recurrent Q-network agent. The rosbridge websocket node handles the communication to and from the Unity application.

made. First of all, the maximum allowed number of steps the agent can per episode will be defined, as it is important for the rewards given by the environment. For the limited test environment, the maximum number of allowed steps will be 150. The number is chosen because it provides the agent with enough steps to search the entire map for the target object, while still requiring it to be efficient. Finding the right step penalty, i.e. the negative extrinsic reward the agent receives at each step, was due to some experimentation. Because the curious DRQN agent receives an intrinsic reward from its ICM at each timestep, the step penalty should have a greater absolute value, unless the agent encounters something truly unexpected. This is because it would be undesirable if the agent learns, that it is better to stay *alive* for as long as possible than it is to quickly locate the target. After some initial tuning of the ICM, the intrinsic reward seems to settle around 0.25 for predictable outcomes. Therefore, the step penalty is set to -1. This way, the agent will still yield negative rewards at each timestep, unless it encounters an unforeseen consequence of its actions. By determining the step penalty, the remaining rewards can be determined. As the episode ends if the agent hits a wall or obstacle, the penalty for hitting a wall or obstacle is set to -150. This way, it will always be more desirable for the agent to stay *alive*, than to hit a wall or obstacle. If the penalty were smaller, the agent might learn a *suicide* behavior, because it deems it better to drive into something and end the episode early, than to stay alive and yields the negative rewards of this. Similarly, the positive reward for hitting the target object is set to 150, causing all episodes where the agent is successful in having a return greater or equal to zero.

As stated in section 3.1, the simulated environment is made with Unity and everything handling events related to the environment is scripted with *C#*, as this is the language Unity natively uses. All communication between entities of the system is, as stated in section 3.2.1, handled by ROS, and all the programming relating to the RL agents is made with *Python 2.7*, as this is supported natively

by ROS. Furthermore, Python has access to the libraries *Keras*[5] (Chollet, 2015) and *Tensorflow*[6] (Abadi et al., 2015), which provide deep learning functionalities. In figure 3.9 there is an overview of the nodes in the system publishing and subscribing to information, as described in section 3.2.1. However, an additional topic is added to help with the handling of NVHER. The Unity system publishes an order to redo an episode to the *redo* topic, if the agent ended the previous episode in a position where the target object can be placed. The agent subscribes to the topic and will redo its previous sequence of actions in the following episode, if an order is received.

As seen in figure 3.9, both the agent and the Unity system publish and subscribe to the *done* topic. This is because the agent system keeps track on the number of steps used in the episode, while the Unity system tracks collisions between the agent and its surroundings. Therefore, both systems can publish an end to an episode and both systems need to know when a new episode is to begin.

Both the DRQN and the curious DRQN will use ε-greedy exploration with linear decay. Starting with an initial value of $\varepsilon = 1$ that decays to $\varepsilon = 0.1$ after 76750 steps, where it remains fixed. The reasoning behind not having ε decay over a longer period of a million steps as done by Mnih et al. (2015), is that the number of steps during training of the limited environment will not come close to a million. Therefore, the agent should also try to exploit its understanding of the environment and not just explore.

An overview of the code developed for the project is found in appendix A.

[5]Keras website: https://keras.io

[6]Tensorflow website: https://www.tensorflow.org

CHAPTER 4

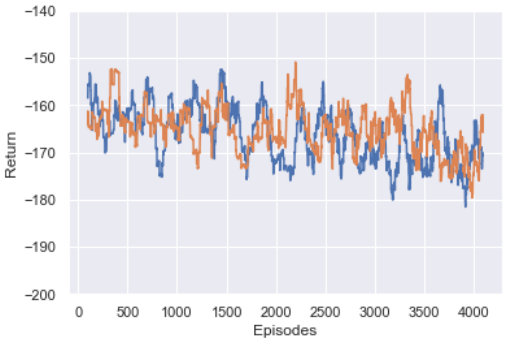
Results and Discussion

The Deep Recurrent Q-network (DRQN) agent and the curious DRQN agent have, before the project hand in deadline, trained for a combined total of 140 hours and 13 minutes in the limited version of the environment. The results presented in this chapter, will be the outcome of this training period, which will serve as a basis for the discussion.

4.1 Training Results

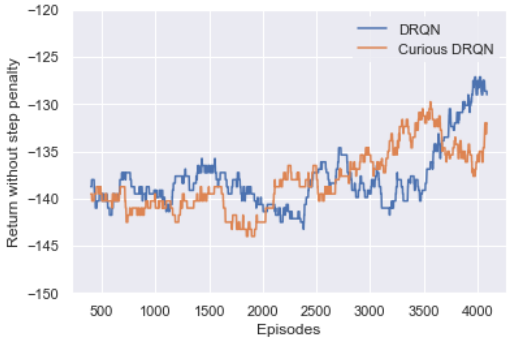
During the training period, both of the agent types have managed to complete slightly more than 4000 episodes each. The results of these training episodes can be seen in figure 4.1. From figure 4.1a, it is evident that neither of the agents have learned a satisfying policy from their training. In fact, the figure shows that both agents have had a tendency to get a smaller return from each episode, as training has progressed. However, as seen in figure 4.1b, as training has progressed, both of the agents have seen a tendency to stay *alive* for more steps during episodes. The downside of staying *alive* for longer, is that the return of the episode will be much lower if the agent collides with the walls or an obstacle later in the episode. This is due to the step penalty. However, if only rewards received from colliding with either walls, obstacles, or the goal target are considered, the agents show a tendency of improvement, as shown in figure 4.1c. This means, that, as training progressed, the agents had more episodes in which they either located the target, or did not collide with anything for the full 150 steps of the episode.

Even though the training of the agents is still in the early stages, e.g. Pathak et al. (2017) train a curiosity driven agent in a 3D environment for millions of steps, there have been signs of early policies for the agents. For instance, both the agents seemed to adopt an early policy of not moving, possibly because experience replay would have shown that standing still, at worst, can yield a negative reward of the step penalty. For the curious agent, this policy seemed to evolve into spinning in



(a) The relationship between the number and training episodes and the return, i.e. the accumulated reward during an episode, displayed as a moving average over the last 100 episodes.

(b) The relationship between the number and training episodes and the steps taken by the agent per episode, displayed as a moving average over the last 100 episodes.



(c) The relationship between the number and training episodes and the return without the step penalty, displayed as a moving average over the last 400 episodes.

Figure 4.1: Graphical illustrations of the training results. The blue line represents the Deep Recurrent Q-network agent, while the orange line represents the curious Deep Recurrent Q-network agent.

place. The reason for this could be, that the outcome of spinning is much harder for the Intrinsic Curiosity Module (ICM) to predict than the outcome of standing entirely still, therefore yielding a higher intrinsic reward. However, as the robot's initial position at the beginning of the episodes is close to a corner, and the robot's behavior is largely influenced by ε-greedy exploration, especially in early episodes, these policies often lead to failure, as the randomness of ε-greedy exploration can easily move the robot in a way, which leads directly into one of the walls.

4.2 Naïve Visual Hindsight Experience Replay

During the training period, the Naïve Visual Hindsight Experience Replay (NVHER) trick, has only been utilized 23 and 15 times for the DRQN and the curious DRQN agents respectively. The low amount of utilizations is due to the fact that NVHER only will be set in motion, if the agent ended the previous episode without a collision, either with the target, walls, or obstacles, and ended the episode in a position with enough space around it for the target to be safely placed there the following episode. Furthermore, it has been observed that NVHER does not always result in a successful episode when applied in the current system. This is caused by the timing of sending messages within the system not being 100 % consistent from episode to episode. In the way the system currently works, when the agent system receives an observation image from the Unity simulation, the agent stops subscribing to the flow of observation images, while it computes the action to take next and does a run-through of experience replay. Then, the action is published, and the system re-subscribes to receive the next observation image. A contributing factor to NVHER not always working could therefore be, that, during the replay episodes of NVHER, there is no need to compute the action to take, making the system able to run slightly faster than when it has to compute the action.

4.3 Dynamics Randomization

As the system is in its current form, the physical dynamics in the simulation remain fixed from episode to episode. Therefore, dynamics randomization could be added to the space of domain randomizations, as future development. The addition of randomized dynamics to the simulations should make the trained agents more adaptable to different real-world environments. E.g., the most optimal policy for driving on a smooth surface is not necessarily the same when driving on a carpet. The use of dynamics randomization to train policies capable of adapting to unfamiliar dynamics, were proven possible by Peng et al. (2017). In their experiments, a RL agent is tasked with controlling a robotic arm, to push an object into a target position. The agent is trained only through simulations, in which the physical properties of the robotic arm and the object is randomized. The agent is then transferred to a physical setup, and proceeds to solve the task with real objects.

4.4 Increase to Actions and Observations

As discussed in section 3.2.2, action-value methods like DRQN are not particularly suited for scenarios with continuing action spaces. This was for this project solved by heavily limiting and discretizing the action space. However, the limited action space comes with the cost of greatly reducing the amount of possible

policies for the agent. This means that very favorably possible policies for the given environment, might be lost in the current setup. Therefore, future developments on the system should see the introduction of a state-of-the-art policy gradient method, capable of handling a continuous action space, like Proximal Policy Optimization (PPO) (Schulman et al., 2017). Similarly, the input images used for the current model are heavily downscaled when comparing to the full capabilities of the robot's camera. Future developments could therefore also see an increase to these input images, allowing the agent more details per observation received.

4.5 Future Development

Many of the improvements to the system discussed above, would require more training episodes, for the agents to improve from them. However, a very limiting aspect of the current system is that the simulations run in real-time. The training is therefore slow, and the most important future development of the system is to improve the speed of training. One approach to solve this, would be to implement a way for the system to simulate physics faster than real-time. Another approach could be to have several agents training in parallel. This could be done by generating more environments at the same time in Unity. Each environment would have its own agent training in it, and each of the agents would have their own streams of observations and actions. Implementing separate information streams is straightforward with the ROS messaging system, which also allows for having more computers connected to the system through a shared network.

Interesting additional future work could be adding humans to the simulations. These could also be randomized in colors, sizes, textures, and placement, as well as having their joints in randomized positions. Furthermore, the simulated humans could be made to move during simulations. Hereby, the agents would have to plan routes around the environment taking future positions of humans into account. As the system is set up at the moment, the agent is only trained to locate the specific goal object from training. However, it would be much preferable if there was a possibility to specify which object the agent should locate from time to time. This could be done by adding the desired object as input to observations and then have the different objects present during training, or a pre-trained object detector could be integrated into the system.

Chapter 5

Conclusion

This project has investigated the possibilities of creating an autonomous robot Reinforcement Learning (RL) agent for map-less navigation in novel environments, with the constraint of using only its onboard camera as an exteroceptive sensor to observe the surrounding environment. Furthermore, the training of the agent was required to be solely based on simulated data. The reasoning behind only using simulated data, were based on the risks associated with gathering explorative RL data from physical robots, such as being costly and a potential safety hazard.

The first problem to be addressed, was how the simulated data for training the agent should be generated. Because the agent only was to sense its surrounding environment through a camera, the simulated data were to be images. The game-engine Unity were therefore chosen to simulate the data, for its capabilities with photo-realistic rendering. A model of the environment was to be made in Unity, were the agent, on an episodic basis, could attempt to locate a target object without colliding with walls or obstacles.

One of the difficulties arising when applying RL to robotics, as according to Kober, Bagnell, and Peters (2013), is the curse of under-modeling and model uncertainty, which refers to the challenges of generating data that sufficiently depicts the real-world counterpart. To battle this curse and allow for future bridging of the reality gap, the concept of domain randomization was built into the simulation system.

To control the agent the middleware ROS was implemented. ROS allowed for communication between entities of the system, i.e. sending messages between Unity and the RL system during training or sending messages between the robot and the computer running the RL system for the real-world test. The messages used to control the agent consisted of a linear and an angular velocity, which was then mapped to the robot's wheels.

The choice was then to either use a policy gradient method and have the agent

directly output the velocities, or to use an action-value method an discretize the action space, and have the agent estimate the values of taking the different actions. Given the better sample efficiency and considering the time constraints, the action-value method DQN were chosen and the action space was discretized to only nine possible actions. Similarly, the input pictures were also downscaled, in a measure to reduce training time. Given the partial observability of the environment a Long Short-term Memory (LSTM) layer was added to the method, effectively making the model a Deep Recurrent Q-network (DRQN).

The reward function of the environment was kept simple, with only positive and negative rewards for colliding with the target object or obstacles respectively, and a small penalty for each step taken, to encourage speed of solving the task. Because of the sparse rewards of the environment, other strategies were pursued to encourage the agent into learning a satisfying behavior. To encourage the agent to explore unseen areas an Intrinsic Curiosity Module (ICM) were implemented, which provides the agent with an intrinsic reward signal corresponding to how unpredictable the outcomes of the agent's actions is. Two additional tricks were implemented to battle sparse rewards. The first was the concept of Learning from Easy Missions (LEM), which, for this project, involved placing the target object in the same room as the agent, to improve the chance of the agent colliding with it during training. The second trick saw the introduction of the concept labeled Naïve Visual Hindsight Experience Replay (NVHER), in which training episodes were controlled to generate a successful outcome. This was done by placing the target object in the position which the agent ended the previous episode in. The agent was then made to repeat the same sequence of actions as it did in the prior episode, hereby making the agent collide with the target object. However, subsequent testing showed that NVHER were unreliable for the given system.

To test the potential of the developed system, a test with a limited version of the environment were sat in motion. In the limited version of the environment, only the starting position of the target object would change from episode to episode. The limited version should make it faster for the agent to learn a satisfying policy but come at the cost of not being applicable to test on the physical robot.

A DRQN agent and a DRQN with an ICM agent was trained for slightly more than 4000 episodes each in the limited environment. The results showed no significantly difference between the agents, with neither of them having developed satisfying policies after the training period. Still, with training of just 4000 episodes it is still in its early stages, and subtle signs of improvement were seen. However, with the time constraints of the project no further training were pursued. Therefore, there were not made any tests on the physical robot, as no agent with a policy to justify this were trained before project hand in.

The biggest issue with the simulation system in the current version, is that it only allows for training the agent in real time. The most important future development of the system is therefore to implement a way to gather training data faster.

Bibliography

Abadi, Martín et al. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. URL: https://www.tensorflow.org/.

Andrychowicz, Marcin, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba (2017). "Hindsight Experience Replay". In: *CoRR* abs/1707.01495. arXiv: 1707.01495. URL: http://arxiv.org/abs/1707.01495.

Asada, Minoru, Shoichi Noda, Sukoya Tawaratsumida, and Koh Hosoda (1996). "Purposive Behavior Acquisition for a Real Robot by Vision-Based Reinforcement Learning". In: *Machine Learning* 23.2, pp. 279–303. ISSN: 1573-0565. DOI: 10.1023/A:1018237008823. URL: https://doi.org/10.1023/A:1018237008823.

Bellman, Richard (1957a). "A Markovian Decision Process". In: *Indiana University Mathematics Journal* 6, p. 15. DOI: 10.1512/iumj.1957.6.56038.

— (1957b). *Dynamic Programming*. 1st ed. Princeton, NJ, USA: Princeton University Press.

Bischoff, Martin (2018). *Announcing ROS#*. URL: https://rosindustrial.org/news/2018/1/8/announcing-ros.

Chollet, François et al. (2015). *Keras*. https://keras.io.

Clevert, Djork-Arné, Thomas Unterthiner, and Sepp Hochreiter (2016). "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)". In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. URL: http://arxiv.org/abs/1511.07289.

E. Rumelhart, David, Geoffrey E. Hinton, and Ronald J. Williams (1986). "Learning Representations by Back Propagating Errors". In: *Nature* 323, pp. 533–536. DOI: 10.1038/323533a0.

Erhan, Dumitru, Yoshua Bengio, Aaron Courville, and Pascal Vincent (2009). *Visualizing Higher-Layer Features of a Deep Network*. Tech. rep. 1341. Also presented at the ICML 2009 Workshop on Learning Feature Hierarchies, Montréal, Canada. University of Montreal.

François-Lavet, Vincent, Peter Henderson, Riashat Islam, Marc G. Bellemare, and Joelle Pineau (2018). "An Introduction to Deep Reinforcement Learning". In: *CoRR* abs/1811.12560. arXiv: 1811.12560. URL: http://arxiv.org/abs/1811.12560.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016). *Deep Learning*. http://www.deeplearningbook.org. MIT Press.

Hart, P. E., N. J. Nilsson, and B. Raphael (1968). "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2, pp. 100–107. ISSN: 0536-1567. DOI: 10.1109/TSSC.1968.300136.

Hausknecht, Matthew J. and Peter Stone (2015). "Deep Recurrent Q-Learning for Partially Observable MDPs". In: *CoRR* abs/1507.06527. arXiv: 1507.06527. URL: http://arxiv.org/abs/1507.06527.

Henry, Peter, Christian Vollmer, Brian Ferris, and Dieter Fox (2010). "Learning to navigate through crowded environments". In: *2010 IEEE International Conference on Robotics and Automation*, pp. 981–986.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). "Long Short-term Memory". In: *Neural computation* 9, pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

Karpathy, Andrej (2015). *Convolutional Neural Networks: Architectures, Convolution / Pooling Layers*. URL: http://cs231n.github.io/convolutional-networks/.

Kingma, Diederik P. and Jimmy Ba (2015). "Adam: A Method for Stochastic Optimization". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. URL: http://arxiv.org/abs/1412.6980.

Kober, Jens, J. Andrew Bagnell, and Jan Peters (2013). "Reinforcement Learning in Robotics: A Survey". In: *The International Journal of Robotics Research* 32, pp. 1238–1274. DOI: 10.1177/0278364913495721.

Kober, Jens, Erhan Oztop, and Jan Peters (2011). "Reinforcement Learning to Adjust Robot Movements to New Situations". In: *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Three*. IJCAI'11. Barcelona, Catalonia, Spain: AAAI Press, pp. 2650–2655. ISBN: 978-1-57735-515-1. DOI: 10.5591/978-1-57735-516-8/IJCAI11-441. URL: http://dx.doi.org/10.5591/978-1-57735-516-8/IJCAI11-441.

Lecun, Yann and Yoshua Bengio (1995). "Convolutional networks for images, speech, and time-series". English (US). In: *The handbook of brain theory and neural networks*. Ed. by M.A. Arbib. MIT Press.

Levine, Sergey, Chelsea Finn, Trevor Darrell, and Pieter Abbeel (2015). "End-to-End Training of Deep Visuomotor Policies". In: *CoRR* abs/1504.00702. arXiv: 1504.00702. URL: http://arxiv.org/abs/1504.00702.

Lillelund, Christoffer Bredo (2018). "Transferring Deep Reinforcement Learning from a Game Engine Simulation for Robots". English. MA thesis. Copenhagen: Aalborg University Copenhagen - AAU.

Lillicrap, Timothy P., Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra (2016). "Continuous control with deep reinforcement learning". In: URL: http://arxiv.org/abs/1509.02971.

Mnih, Volodymyr et al. (2015). "Human-level control through deep reinforcement learning". In: *Nature* 518.7540, pp. 529–533. ISSN: 00280836. URL: http://dx.doi.org/10.1038/nature14236.

Nair, Vinod and Geoffrey E. Hinton (2010). "Rectified Linear Units Improve Restricted Boltzmann Machines". In: *Proceedings of the 27th International Conference on International Conference on Machine Learning*. ICML'10. Haifa, Israel: Omnipress, pp. 807–814. ISBN: 978-1-60558-907-7. URL: http://dl.acm.org/citation.cfm?id=3104322.3104425.

Ng, A.Y., A Coates, M Diel, Varun Ganapathi, J Schulte, B Tse, E Berger, and E Liang (2004). "Inverted autonomous helicopter flight via reinforcement learning". In: *Proceedings of the International Symposium on Experimental Robotics*.

Olah, Chris (2015). *Understanding LSTM Networks*. URL: http://colah.github.io/posts/2015-08-Understanding-LSTMs/.

Olah, Chris, Alexander Mordvintsev, and Ludwig Schubert (2017). "Feature Visualization". In: *Distill*. https://distill.pub/2017/feature-visualization. DOI: 10.23915/distill.00007.

Open Source Robotics Foundation (2016). *ROS*. Version Kinetic Kame. URL: http://wiki.ros.org/kinetic.

OpenAI, John Schulman, Oleg Klimov, Filip Wolski, Prafulla Dhariwal, and Alec Radford (2017). *Proximal Policy Optimization*. URL: https://openai.com/blog/openai-baselines-ppo/.

OpenAI et al. (2018). "Learning Dexterous In-Hand Manipulation". In: *CoRR*. URL: http://arxiv.org/abs/1808.00177.

Pathak, Deepak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell (2017). "Curiosity-driven Exploration by Self-supervised Prediction". In: *ICML*.

Peng, Xue Bin, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel (2017). "Sim-to-Real Transfer of Robotic Control with Dynamics Randomization". In: *CoRR* abs/1710.06537. arXiv: 1710.06537. URL: http://arxiv.org/abs/1710.06537.

Pinto, Lerrel, Marcin Andrychowicz, Peter Welinder, Wojciech Zaremba, and Pieter Abbeel (2017). "Asymmetric Actor Critic for Image-Based Robot Learning". In: *CoRR* abs/1710.06542. arXiv: 1710.06542. URL: http://arxiv.org/abs/1710.06542.

Sadeghi, Fereshteh and Sergey Levine (2016). "(CAD)^{2}RL: Real Single-Image Flight without a Single Real Image". In: *CoRR* abs/1611.04201. arXiv: 1611.04201. URL: http://arxiv.org/abs/1611.04201.

Sahni, Himanshu, Toby Buckley, Pieter Abbeel, and Ilya Kuzovkin (2019). "Visual Hindsight Experience Replay". In: *CoRR* abs/1901.11529.

Samuel, Arthur L. (1959). "Some Studies in Machine Learning Using the Game of Checkers". In: *IBM Journal of Research and Development* 3, pp. 210–229.

Schulman, John, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov (2017). "Proximal Policy Optimization Algorithms". In: *CoRR* abs/1707.06347. arXiv: 1707.06347. URL: http://arxiv.org/abs/1707.06347.

Silver, David et al. (2017). "Mastering the game of Go without human knowledge". In: *Nature* 550, pp. 354–. URL: http://dx.doi.org/10.1038/nature24270.

Smart, William D. and Leslie Pack Kaelbling (2002). "Effective Reinforcement Learning for Mobile Robots". In:

Sondik, Edward (1978). "The Optimal Control of Partially Observable Markov Process over the Infinite Horizon: Discounted Costs". In: *Operations Research* 26, pp. 282–304. DOI: 10.1287/opre.26.2.282.

Sutton, Richard S. and Andrew G. Barto (2018). *Reinforcement Learning: An Introduction*. Second. The MIT Press. URL: http://incompleteideas.net/book/the-book-2nd.html.

Tesauro, Gerald (1995). "Temporal Difference Learning and TD-Gammon". In: *Commun. ACM* 38.3, pp. 58–68. ISSN: 0001-0782. DOI: 10.1145/203330.203343. URL: http://doi.acm.org/10.1145/203330.203343.

Tobin, Joshua, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel (2017). "Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World". In: *CoRR* abs/1703.06907. arXiv: 1703.06907. URL: http://arxiv.org/abs/1703.06907.

Unity Technologies (2018). *Unity*. Version 2018.3. URL: https://unity.com.

Valenzuela, Christine L. and Pearl Y. Wang (2001). "Data Set Generation for Rectangular Placement Problems". In: *European Journal for Operational Research* 2001, pp. 16–20.

Vinyals, Oriol et al. (2019). *AlphaStar: Mastering the Real-Time Strategy Game StarCraft II*. https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/.

Watkins, Christopher John Cornish Hellaby (1989). "Learning from Delayed Rewards". PhD thesis. Cambridge, UK: King's College. URL: http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf.

Zhelo, Oleksii, Jingwei Zhang, Lei Tai, Ming Liu, and Wolfram Burgard (2018). "Curiosity-driven Exploration for Mapless Navigation with Deep Reinforcement Learning". In: *CoRR* abs/1804.00456. arXiv: 1804.00456. URL: http://arxiv.org/abs/1804.00456.