Transferring Deep Reinforcement Learning from a Game Engine Simulation for Robots

Christoffer Bredo Lillelund
Msc in Medialogy
Aalborg University CPH
Clille13@student.aau.dk

May 2018

Abstract

Simulations are used to gather data for machine learning algorithms at low cost. However, many robot simulators can not render realistic graphics. Realistic image data is essential for several robots using reinforcement learning algorithms in fields such as self-driving cars, agricultural weed detection and grasping objects. In this report, we propose the use of modern game engines with highly realistic rendering for simulating robots and training them with deep reinforcement learning using image data. We successfully simulated and trained a Turtlebot2 robot in Unity3D with Deep Q-learning to find and drive into a blue ball. The resulting reinforcement learning model was used to control a real Turtlebot2 robot. The simulated and real robot are interchangeable by design, making it easy to use the reinforcement algorithm to control either. The real robot was controllable by the Q-learning algorithm, but not able to perform the task. The use of modern game engines for simulation of robots for reinforcement learning is shown to be promising. However, for future work, testing of more realistic simulation environments are needed to assess the usability of game engines for realistically simulating robots.

1 Introduction

From the industrial era to modern society, the use of robots has increased human capabilities and efficiency exponentially. Robots are capable of performing simple and repetitive tasks effectively, accurately and with high frequency. However, historically they have not been able to act in dynamic and complex environments. A robot arm in a factory can, for example, not operate on a new task without reprogramming. Also, programming a robot to act in a dynamic and unpredictable environment is almost impossible due to the amount of possible scenarios it may encounter. However, with modern machine learning algorithms, the capabilities of robots to perform complicated tasks that

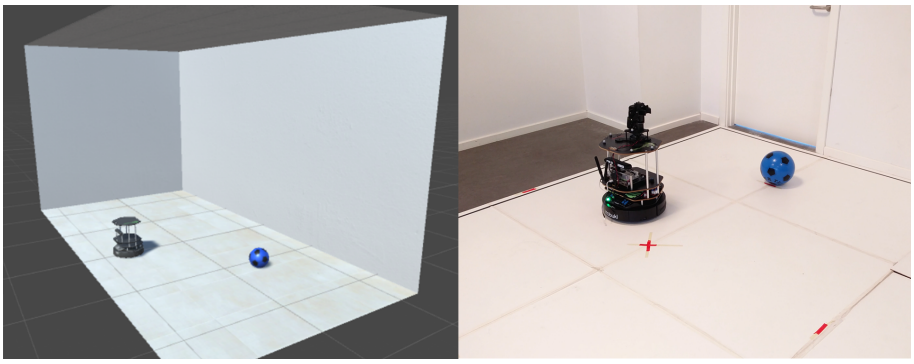

Figure 1: A Turtlebot2 robot was simulated in Unity3D environment (left) and trained to find and drive into a blue ball by using Q-learning. The real Turtlebot2 (right) was used to verify the ability to control the robot just as the simulated counterpart. The training model from the simulated Turtlebot2 was used to control the real robot, but it was unable to find and drive into the blue ball in the real environment. Both robots are controlled using the same ROS communication system and Reinforcement Learning model.

requires problem solving is indeed feasible. State of the art examples of this is map-less navigation [1], socially aware motion planning [2], grasping objects [3], and self-driving cars [4].

A method of teaching robots to perform tasks in dynamic and complex environments is Reinforcement Learning (RL). RL teaches an agent to perform a task by learning from its past experiences. RL has its roots in theories of neuroscience about human and animal's ability to predict future events, by changing their expectations through rewards and punishments [5]. In RL, an agent uses some knowledge of the *state* of its environment to predict the best *action* to take [6]. A *state* for a robot could for example be the sensor inputs for a self-driving car to determine what is around it, and an *action* could be to accelerate, brake or turn.

The RL agent is taught how to perform its task by receiving rewards for its actions and calculates how to gain the most rewards; even rewards in the distant future. In contrast to supervised machine learning techniques, the agent does not need labelled data. It only needs to model a *policy* from past experiences to estimate the correct actions to take to optimize rewards. A *policy* is what the RL algorithm uses to determine which action to take for certain states. The ability to teach an AI to perform complex tasks by just letting them act in the environment is very important for the machine learning community, as it could decrease the need for human programming of complex problem solving, and increase learning efficiency of AI applications.

Examples of experiments using simulations for training robots to perform complex tasks are grasping objects [7] and navigating dynamic environments [2].

Google Deepmind [8] made an agent able to perform complex tasks at human-level only by the means of raw sensor data in 2015. The Deep-Q network used was able to play 49 different Atari games comparable to professional human players, with only the information of the 84X84X4 pixel values of the game screen.

Researchers had strayed from using trail-and-error reinforcement learning for robots, as low amounts of training steps can cause it to be unreliable. However, Pinto & Gupta [3] succeeded in training a robot to grasp and pick up different objects, with only images of the objects as input. The amount of experience needed resulted in 700 hours of trail-and-error by the robot. The learning comes at the cost of great many resources, such as electricity usage, potential damage to the robot and its surroundings, as well as wear on the robots parts. Meanwhile, the potentially expensive robot is occupied doing learning tasks, unable to perform any other productive tasks.

The need to run the robot for many hours was overcome by Google, by using many robots simultaneously [9]. Between 6 and 14 robotic arms was simultaneously used through the process, each collecting grasping attempts and collectively learning from each other how to grasp and pick up objects in front of them. The risk of damage caused by the robots are still present, but the robots can be replaced if the budget allows. however, the budget does not allow for a high number of robots and potential damage on these for many researchers. This multi-robot setup is therefore not feasible as a general solution.

To solve this issue of cost and risk of damage, previous research papers has performed training of robots in virtual environments [1] [2] [7] [10]. By simulating faster than real world physics allow, and keeping the actions within a simulation, the resource cost is greatly reduced. No real robot are used, there is no risk of damaging any equipment or personnel, and with increased simulation speed the robot learns faster.

Image sensor data is important for robots to understand its surroundings, and thereby make correct choices in complex and dynamic environments. As an example, self-driving cars needs image data to understand its surroundings sufficiently to take proper decisions to drive safely [4] [10]. Without cameras, the self-driving car will not have sufficient data to take the correct decisions.

Other robotics applications use raw image data to complete complex tasks. Examples of this are weed detection in agricultural robots [11], and medical robots for precise and minimally invasive surgery [12].

Each application need precise, realistic image data to perform correct detection and recognition of its environment and thereby taking the correct actions. Failure to recognize objects can cause healthy crops to be sprayed with pesticides, incorrect surgery to be performed, or the action could cause a car accident. However, if the training of the robots are done properly, the benefits are great. Autonomous precision farming, precise help with surgery, and safe self-driving cars. Other examples of needing image data for robots are multi-floor navigation through an elevator [13], fastening bolts in aircraft production using a humanoid robot [14], and hand-eye coordination for grasping [9].

Gazebo [15] and V-rep [16] are state of the art robot simulators, designed

to enable fast and precise simulations of robot controls. However, Gazebo and V-rep do not have suitable computer graphics capabilities to replicate a real world scenario, such as a dynamic crowd of pedestrians, a busy city road, or fields of grass. Therefore they would not be suitable to provide image data to train a robot to perform in the real world.

Researchers has successfully been using games and game engines to collect usable realistic image data sets for machine learning algorithms [10] [17]. We propose the use of a game engine with realistic rendering capabilities to simulate and train robots with raw image data using RL.

In this paper, we use a real-time connection between Robot Operating System (ROS) and the game engine Unity3D [18] to create a realistic simulation of a robot, its controls, and environment. A Turtlebot2 robot is simulated in the Unity3D game engine and provide RGB camera sensor data for a reinforcement learning algorithm. The idea is to create a safe environment for the robot to learn by RL, without the use of a real robot. The resulting RL model will then be used to control a real Turtlebot2 robot to verify if the learning of simulated training of the robot can be transferred to its real counterpart. The focus of the paper is to show the plug-and-play capabilities of Unity3D and other game engines to train on simulated robots and control real robots with the resulting learning.

We list the main contributions of this paper: (1) Proposing modern game engines as realistic robot simulators and showing the plug-and-play capabilities for changing between simulated and real robot. (2) Showing that a reinforcement learning model trained in a game engine environment are easily transferable to a real robot.

2 Background

2.1 Reinforcement Learning

Data points for machine learning is generally assumed by the algorithm to be independent of each other, meaning that the result of one data point does not effect the result of the algorithm for the next data point.

However, in reinforcement learning the state it enters is dependent on the previous states and actions it performed. For example, if a self-driving car drives forward, the next state would be a result of the last state and the action of driving forward. As a result, the action taken by the agent will affect the state that the agent finds itself in. Reinforcement learning algorithms therefore have to take future states into consideration and which states its actions will lead to.

Bellman's equation calculates maximum achievable reward from the current state by all possible actions and future states [19]. The Q-value is the maximum estimated reward available from the current state for all possible future stats and actions. The calculation of the Bellman's equation can therefore estimate the potential future reward (Q-value) of taking a certain action in the current

state and landing in the corresponding next state.

$$Q(s, a) = r * \gamma maxQ(s', a') \tag{1}$$

$Q(s, a)$ is the Q-value for the current state and action, r is the reward for entering the current state, $Q(s', a')$ is the Q-value of the next state and actions, and γ is a discount factor that determines how important future potential rewards are for the agent compared to immediate rewards.

In 2013, Mnih et al. introduced Q-learning [19]. Q-learning is a RL algorithm based on Bellman's equation. To perform RL the Q-learning equation (2) updates the Q-value of certain step and action by adding the new estimated Q-value multiplied with a learning rate factor α. The Q-value is therefore changed slightly (depending on the α value) instead of completely, every time the state is met. The algorithm learns to be more precise in its estimation of Q-value the more it encounters the state.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_{t+1} + \gamma maxQ(s_{t+1}, a) - Q(s_t, a_t)) \tag{2}$$

Using the Q-values calculated, the Q-learning algorithm will train a neural network fitting the weights to choose the action that will provide the most reward given the current state. This training is performed over many iterations of entering states and taking actions.

There are two kinds of spaces when describing an environment, called *discrete* and *continuous* space. A discrete space is when the space is divided into a finite amount of possible states. For example, when playing chess, the amount of states available are finite. The pieces has a finite amount of fields to be placed on, and there are a finite amount of ways all the pieces can be arranged on the board. A discrete space are generally not applicable in the real world. For example, chess pieces can in the real world be placed anywhere on the board, not confined to a field, as their movement are continuous when moving from one place to another. Continuous space is when the space or environment cannot be divided into individual fields. The state of a real environment is continuous, meaning that to move to a new position, you need to move to the half-way point to that position, and before that move to the half-way point of the half-way point etc.. The space therefore contains an infinite amount of possible states.

One of the main difficulties when training reinforcement learning for the real world, is that the states and the actions of the robot, such as sensor data and acceleration, does not follow a discrete space. The need for an algorithm with capabilities of predicting actions for states of continuous space is therefore needed. The Q-learning algorithm from Mnin et al [19] can be used for continuous spaces, but can be computationally expensive, thus affecting the training process.

Lillicrap et al. [20] introduced a deep RL algorithm based on the Q-learning algorithm from Mnih et al., which they call Deep Deterministic Gradient Policy (DDPG). The DDPG algorithm is an actor-critic model with capabilities of estimating actions in continuous space. An actor critic model uses two separate networks. The actor network predicts the action to take given the state, while

the critic "evaluates" the decision of the actor, giving a score representing how good the action was.

However, as this report aims toward showing the plug-and-play capabilities of game engine simulations of robots, the robot will have a simple task to perform. Due to the simplicity of the task, the Q-learning algorithm from Mnih et al. [8] will be used. For future studies with more complex robot tasks, a more complex learning algorithm such as the DDPG actor-critic model by Lillicrap et al. [20] is expected to be compulsory.

2.2 Robot simulations

Chen et al. [2] trained their navigating robot to move through crowds while being socially aware, meaning it would follow common norms of passing and overtaking. The agent was trained in a simulation consisting of four agents, learning from each other. Even though the real robot used for validation in the real world used cameras in addition to other sensors, the cameras were only used to detect and, with help from Intel Realsenses, determine the distance to pedestrians. The agent therefore was not trained on camera sensor data, but only the position and distance of by-passers. If the robot was trained using realistic rendering in the simulation, this position and distance translation from real images would not be necessary.

Chen et al. [2] trained the robot to be socially aware with 4 webcam cameras, 3 Intel Realsense cameras, and a Lidar Pointcloud laser scanner. The benefit of training with raw camera data would be to significantly decrease the cost of sensors needed for the robots.

The movement of real world objects are continuous. A robot simulation therefore needs to simulate the movement of a robot in a continuous space, which follows the physics of the real world. If a reinforcement learning algorithm was taught in a simulation that deviates from the real world, the model will not be transferable to a real world environment. Furthermore, when using a camera as input for the robot, the graphical rendering of the simulation must correspond to real camera inputs.

There are robot simulation applications available, which are useful for simulating robot controls. Gazebo is an open source robot simulator with the purpose of simulating realistic controls for robots. Gazebo offers quick and easy imports of URDF files and control-ability of robot joints [15]. Gazebo does not contain a realistic rendering engine (see Figure 2). A rendering library, which implements a render engine capable of ray-tracing [21], is available. The library provides capability of realistic reflection renders, but does not enable the user to create realistic renders that resembles the real world.

Another robot simulation application is V-rep [16]. This simulation platform is developed to support fast algorithm development, automation simulations, and prototyping robotics in a safe environment. The platform is great for simulating big autonomous robot systems, such as factory floors, as well as smaller cases, such as the map-less navigation performed by Tai, et al. [1]. However, the application suffers from the same issue as Gazebo, and does not have realistic

Figure 2: Example of the graphics capabilities of the Gazebo robot simulator.

rendering (see Figure 3). Tai, et al. [1] used a point laser scanner as input for the robot, thereby negating the need for realistic graphics in the simulation.

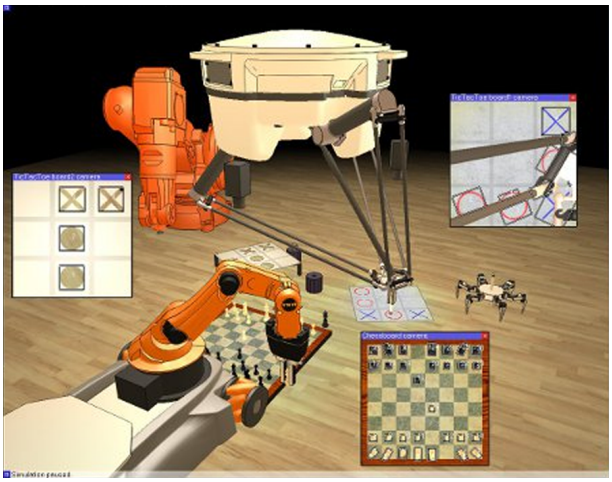

Figure 3: Example of the graphics capabilities of the V-rep robot simulator.

The lack of realistic rendering in robot simulation is apparent. Both the render-engines of Gazebo and V-rep are lacking in realism, which could prove a problem for learning robots relying on camera data.

The idea of performing training on robots in a high-fidelity simulated environment are being investigated by the technology company Nvidia. In 2017 Nvidia announced Isaac, an SDK with capabilities of simulating robots and performing machine learning from the simulated robots' sensor data [22]. The goal

of Isaac is to create an inexpensive and safe environment for robots to act and learn. Nvidia Isaac strives for realistic rendering and physics to simulate the world as realistically as possible. They stretch that the render capabilities of such simulation is very important for robots using camera data.

However, at the time of writing, the Isaac SDK has yet to be released to the general public and the SDK has not been tested by us. The dedication from Nvidia to create Isaac highlights the usefulness and need of such simulator with highly realistic visuals.

Opposed to Gazebo and V-rep, Unity3D [18] is a game engine with the purpose and capability of creating realistic environments and render them in real-time (See Figure 4). It also contains hrealistic physics that can be altered to the need of the simulations. The game engine can therefore not only simulate this world, but other worlds such as foreign planets, that are difficult to get robots to. Unity3D is a promising platform to use for simulating a robot and its environment realistically.

Figure 4: Example of the graphics capabilities of the Unity3D game engine. This image is from a demo reel of a real-time rendering cinematic called ADAM.

2.3 Realism in Simulations

As mentioned before, projects with robots trained in a virtual environment has already been made [1] [2]. However, as far as we know, at the time of writing, no research project has used raw camera data from a simulated robot for reinforcement learning, and successfully transferred the learning to a real robot.

Game engines has already been used to collect data for deep learning [10] [17] [23] [24]. The use of game engines is due to the collection of data only costing computation power, reduced cost of human labelling, and the realistic

rendering capabilities of the engines.

The idea of collecting data from realistic renders is not new. In 2010 and 2014, authors Mariín et al. [23] and Xu et al. [24] respectively, captured images of pedestrians using the game Half-Life 2. The images had the capability of training a pedestrian recognition classifier with approximately the same accuracy as classifiers from real data sets.

In between these studies, other research papers has found that the virtual data sets and real data sets were too different from each other, which decreases the accuracy of object detection [25] [26]. In the paper from Xu et al. [24], they counter the difference in data sets by training on virtual data as well as additional but relatively few real world images. They also use enhanced textures in Half-Life 2 to increase the quality of the graphics.

Johnson-Roberson et al. [10] successfully used the game GTA V to collect images from a simulated car, and trained a deep learning network for object recognition with the same accuracy as a deep learning network trained with KITTI and Cityscapes data sets. Self-driving cars need to be able to recognize objects around them and thereby take the correct action accordingly. Therefore, camera sensor data is needed as an input. As mentioned by Johnson-Roberson et al. [10], vision-only driving is very sought after in the autonomous driving community, due to the low sensor costs. Even though the collected images from GTA V was able to train an object recognition model as accurate as real data, the authors mention that they needed many more training images to achieve the same accuracy. The reduced cost of acquiring the images should, however, more than equalize the effort.

These research papers show the versatility of game engines to collect useful data for real world applications, such as object recognition. The early experiments [23] [24] needed to use real data as well to receive a proper accuracy of the algorithm. These experiments were performed with outdated graphics compared to what is achievable nowadays. Johnson-Roberson et al. did not need real world data, as the game they used had much more realistic graphics. A hypothesis for decreasing the difference in real world data and virtual data is to decrease the difference between the virtual and the real world, thereby eliminating the need for real world data to fine tune algorithms.

Computer graphics have become capable of creating highly realistic renderings, even in real time. This should solve the issue of data set differences and increase the accuracy in the real world for systems trained in a virtual environment.

Another important factor for transferring reinforcement learning models to the real world, is the variability of the conditions of the simulated environment. Robots of the real world may need to perform in multiple environments and with often changing lighting conditions. To train a robot for a single condition limits its general usefulness as the image data is very dependent on the lighting conditions of the environment.

To train a machine learning model of any kind, it is important to have varying data for the learning algorithm. Tremblay et al. [17] uses the Unreal game engine to gather data for supervised machine learning. The authors underline

the importance of changing parameters within the simulation, to create a wide variety of realistic images. They use *domain randomization*, which essentially means randomizing lighting, pose, and object textures in a scene. If the environment and its lighting conditions are changing, the model is trained to learn the task in many conditions, thereby generalizing the performance to many environments. This is especially important when the robot only input is from raw pixel data. As shown be Tremblay et al. [17], modern game engines has the capability of domain randomization and requires little to implement.

The Unity3D game engine has the capability of realistic rendering, physics, and domain randomization. The game engine is chosen for this project to simulate a robot and train it using reinforcement learning.

Due to time constraints, the training of the reinforcement learning model in this paper will have static lighting conditions and the simulated environment will be not have the focus of being realistically similar to the real world environment. As mentioned before, the experiment of the paper focuses on the plug-and-play capabilities of the game engine. For improvement of robot performance in real environments, a realistically rendered environment with randomizing lighting conditions is believed to be essential.

3 Implementation

In this section, the implementation of the robot simulation and RL algorithm is described. The robot chosen it the Turtlebot2. We will establish a communication between ROS with Unity3D to create a similar message system as a real Turtlebot2 uses when controlled by ROS. The simulation of the robot will be run on Unity3D with realistic physics, such as collision, gravity and velocity control. The RL algorithm will be performed by a python script, which also handles ROS messages that are responsible for controlling the robot.

3.1 ROS

ROS is a common middleware for robotics handling communication between robot hardware and instructions given by software. It is used to send and receive messages to and from robot sensors and actuators thus allowing remote controls for a robot. In this paper, we use images from a camera sensor mounted on the robot as the only input for the RL algorithm. Therefore, in our case we will need to receive an image message from the simulated and the real robot.

To receive external messages, ROS needs to subscribe to a topic that publishes the needed data. A topic is a stream of messages. The topic can receive information by getting data published to it, and it can send information by being subscribed to. A publisher pushes data to the stream, while a subscriber receives the data from the stream.

Connecting to the simulation is done by using the ROSbridge package. This enables the ROS messages to be send across networks to robots or other applications.

A *sensor_msgs/Image* message topic is needed, with a publisher to the topic from the simulation, and a subscriber to the topic in the RL algorithm. A *sensor_msgs/Image* message contains a byte array of pixel data and description of the format. For this project, the *sensor_msgs/Image* message format is PNG. For ROS to receive images from the mounted camera, the robot publishes images to the *sensor_msgs/Image* topic.

To control the velocity of the robot, a *geometry_msgs/Twist* message is needed. A *geometry_msgs/Twist* message contains 6 values: the linear and angular x, y, and z velocity. For the Turtlebot2, only the linear velocity on the x-axis and the angular velocity on the z axis is needed to drive forwards, backwards, and turn. The other values are set to 0.

Additionally, the RL algorithm needs rewards published from the simulation. Rewards are sent from the simulation through ROS the RL algorithm by a *std_msgs/String message*. The RL algorithm translates the String messages to floating point values. A publisher is created in Unity3D, and a subscriber in the python script, capable of sending rewards to teach the model of its actions.

3.2 Unity3D robot simulation

Unity3D was chosen as the game engine to simulate the robot. Unity3D contains a realistic physics system with controllable gravity, mass of object, and great collision detection. All of which are important to simulate real world interactions between dynamic objects. As have been seen in Figure 4, the rendering capabilities of the game engine is also highly realistic, which is essential for replicating complex real world scenarios.

Unity3D does not provide native support for ROS compatibility. The ROS# package for Unity3D provides the ability to create message classes similar to the messages used by the ROS [27]. It can send and receive messages to and from ROS by providing the ability to create publishers and subscribers. It also provides the ability to import URDF files. URDF files are standard ROS XML representations of the robots model. It has detailed information about the robots kinematics, dynamics, and sensors. This makes it easy to import a robot model and place its 3D model in the Unity3D environment.

The first robot implemented in Unity3D through the ROS# plugin was a Baxter robot (see Figure 5). Baxter is a two-armed industrial robot with end effectors capable of grasping objects. The URDF file of the robot was imported into Unity3D, and collisions with other objects was achieved. To apply physics to an object in Unity3D a rigidbody component is used, which controls physics interaction of the object. As all parts had a rigidbody attached, they would all interact with each other. However, due to the collision boxes of each part of the robot overlapping, they would constantly collide with each other. This would cause the robot to glitch. A work-around for this was tried. The arms were set as kinematic rigidbodies, which means they would not be affected by physics, but they would still affect other non-kinematic objects. That introduced new problems, where the rotation of the arms could no longer be controlled by adding torque or velocity to the rotation, as physics did not affect them. Therefore the

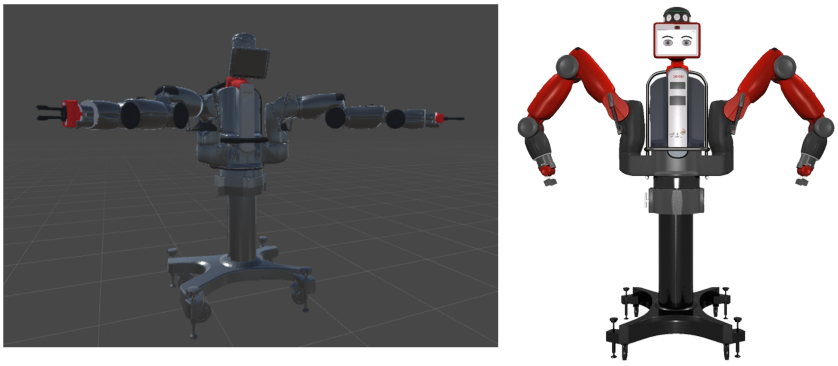

Figure 5: A side by side comparison of the simulated and a real Baxter robot. The first robot simulated in Unity3D for this project was the Baxter robot.

robot needed to be controlled by a step-wise rotation change of the joints, which does not realistically simulate the real robot. Also, as all links were kinematic, they were not able to collide with each other, making the robot able to phase through itself. These issues are believed to be solvable within Unity3D, but due to time constraint, another robot with no movable joints was chosen for the purpose of this project.

3.3 Simulating Turtlebot2

The Turtlebot2 is a small robot with four wheels, one front, one back, one left, and one right (see Figure 6). The Turtlebot2 steers and drives by controlling the torque of its left and right wheel. It can move forwards, backwards, and turn left and right, giving it 2 degrees of freedom for control. Additionally, the Turtlebot2 has a mounted Primesense camera. As the Turtlebot2 contains no individually moving parts except for its wheels, the difficulties of the Baxter was not present. The Turtlebot2 robot was therefore chosen for this project.

The Turtlebot2 was successfully imported into Unity3D (see Figure 7) and was realistically controllable by making only a few adjustments. The adjustments were as follows.

The imported 3D model from the URDF file had many individual components, each with a rigidbody and each was held together by joints to adjacent parts. As the parts of the Turtlebot2 are all static, needing no individual movement, the rigidbody components were removed and each part was converted to static objects connected to a parent object, being the robots base. One rigidbody was connected to the parent class of the Turtlebot2 to enable physics for

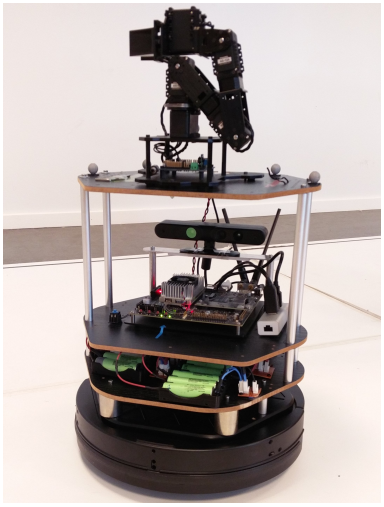

Figure 6: The Turtlebot2 robot.

the robot. This caused all parts of the robot to move uniformly with the base, just as the real robot, and keep all physics interactions.

Additionally, wheel colliders were added at the locations of the wheels at the bottom of the robot to enable driving. Linear and Angular velocity of the simulated robot was controlled by adjusting the torque applied to the left and right wheel.

A camera object was added to the robot, placed at the Primesense camera models location, with similar settings to the real Turtlebot2. These wheel colliders and the camera made the robot able to control and provide images in the same manner as its real counterpart.

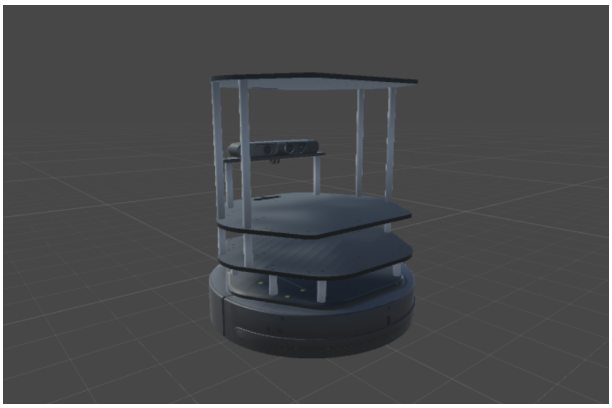

Figure 7: The simulated Turtlebot2 robot in Unity3D.

To control the simulated Turtlebot2, an ROS connection is used. The ROS communication is implemented as a replication of the communication with a real Turtlebot2. By having the same communication system for the real and simulated robots, the robots are seamlessly interchangeable.

In Unity3D, the simulated robot is controlled by receiving a *geometry_msgs/Twist* message from ROS. The message is read by creating a subscriber to the *geometry_msgs/Twist* topic using the ROS# plugin. The linear x-axis and angular z-axis velocity from the *geometry_msgs/Twist* message are translated to torque, which is applied to the left and right wheel of the robot, enabling it to drive and turn. The camera object in the Unity3D simulation renders an RGB image at size 80x60x3. The low resolution is due to computation power needed to perform convolutional RL. The camera object renders to a render texture, which is published to a *sensor_msgs/Image* topic using the ROS# plugin. The simulated robot therefore controls and sends data to ROS the same way as a real Turtlebot2 does. Due to this, the two robot counterparts are interchangeable, and the same RL model can be applied to both of them. The resulting architecture of the message system can be seen in Figure 8.

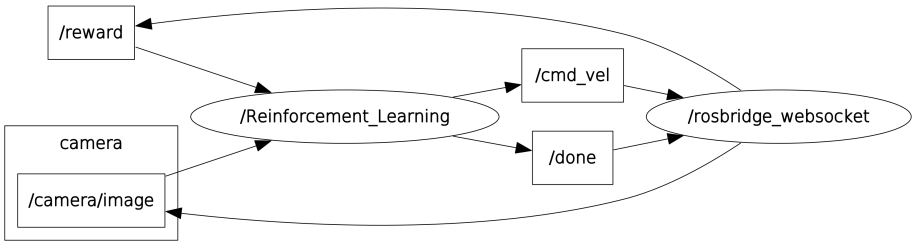

Figure 8: The message system used to perform RL on the simulated robot in Unity3D. */rosbridge_websocket* is the connection to the robot or the simulation (Unity3D). As can be seen, the RL script publishes to the */cmd_vel* and */done* topics. The messages published are *geometry_msgs/Twist* and *std_msgs/String* messages respectively. The robot then subscribes to these topics, using them to control the robot and knowing when to reset the environment. The robot publishes to the */camera/image* and */reward* topics. The messages are *sensor_msgs/Image* and *std_msgs/String* messages respectively. The RL learning script subscribes to these topics to get information about the environment and using that information for RL.

3.4 Reinforcement Learning Algorithm

To create the RL model for this project, the Keras neural network library based on Tensorflow is used. A Python script with the library is able to create RL models, fit model with new weights, predict the estimated best action from the state, and more. ROS runs publishers and subscribers by python scripts. It is therefore possible to perform reinforcement learning and send and receive

Table 1: Table of the nine possible actions to provice for the Turtlebot2 by the RL algorithm.

Action	Forward	Back	Stand Still	Turn Left	Turn Right	Back Left	Back Right	Forward Left	Forward Right
Lin. X vel	1	-1	0	0	0	-1	-1	1	1
Ang. Z vel	0	0	0	-1	1	-1	1	-1	1

messages to and from topics in the same script.

For this project, a single python script sets up a subscriber for the *sensor_msgs/Image* and *std_msgs/String Reward* topics, a publisher for the *geometry_msgs/Twist* and *std_msgs/String Done* topics, and contains the RL algorithm. For each time an image is fed to the subscriber from the *sensor_msgs/Image* topic, it runs a step of RL, predicting the best action to take and publishes that *geometry_msgs/Twist* message to the *geometry_msgs/Twist* topic.

For the RL algorithm, we will use a Deep Q-learning algorithm [8]. The neural network used for the Q-learning algorithm consists of three convolutional layers, each with 32 filters and no pooling, followed by two fully-connected layers, each with 200 neurons. The final output layer contains nine neurons. These represent the available actions covering its 2 degrees of freedom controls (see Table 1). The neural network has a learning rate of 10^{-3} and a discount factor $\gamma = 0.99$. From the input image, the neural network will chose its estimated best action (see Figure 9).

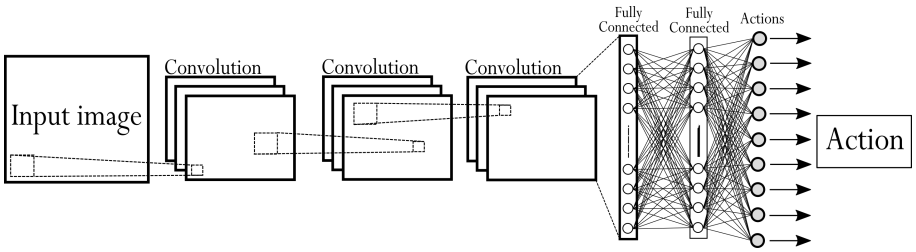

Figure 9: The Neural Network used in the Q-learning algorithm. The network is sequential and deep. It consists of three convolutional networks followed by two fully connected layers. Nine different actions are available for the robot, therefore the last layer consists of 9 neurons.

The neural network is updated through the Q-learning algorithm with rewards send from the Unity simulation. When the scene has to be reset, another string message is published from the RL script and subscribed to in Unity3D, which then proceeds to reset the robot environment when receiving such a message.

The Q-learning algorithm proposed by Mnih et al. [8] trains its RL model by using *experience replay*. Each state, action, reward and next state of a step of RL is stored in a memory matrix. These are called experiences. Experience replay is to train the RL model with multiple past experiences, which are not necessarily the most recent. It is a way for the RL model to learn from the same experience multiple times. As the Q-values in the algorithm changes slightly for each Q-value estimation, training on the same experiences multiple times are beneficial for learning.

For each step of RL for this implementation, a batch of 16 random samples from the memory is used to train the model. Taking random samples instead of the 16 most recent experiences reduces the risk of overfitting to recent actions or causing oscillation. As the system trains the model by 16 experiences each time one new experience is encountered, each experience is training the model multiple times, resulting in well-rounded learning. The max memory size is 10^6.

The robot will act in the environment sending images and rewards to the RL algorithm, which then proceeds to predict the best action, sending that to the robot. Upon receiving the next image, the RL algorithm will perform experience replay, calculate the Q-value achieved in the state and send the chosen action to the robot. It fits the model using experience replay, recalculating the weights of the neurons in the network.

For this RL algorithm, two models are used. The first model is used to calculate the Q-value of the current state and is constantly fitted with the experience replay from every stop. The second model called the Target Model, is used to estimate future Q-value of the next states. The fully calculated Q-value therefore uses both models for the calculation. The Target model is only fitted at the end of an episode. It is set to the same weights as the first model, thereby gaining the learning from the experiences. This is done to not influence the future predictions in an episode while acting in that episode. A pseudo code explanation of the RL algorithm can be seen in Algorithm 1.

A RN agent has two possibilities when choosing actions. It can exploit or explore. Exploiting is when the agent takes the estimated best action. By doing this the agent will perform its best in the environment according to its learning. However, having the agent only exploit can cause the agent to perform the same actions, thinking it is the best achievable strategy, even though another unknown strategy may be superior. It has just not tried the other possibilities.

That is where exploration comes in. Exploration is when the agent performs an action that is not necessarily the best estimated action. This causes the agent to explore the environment, possibly finding actions that are better to take than those previously thought by the model.

A way of controlling the exploitation and exploration of the RL agent is to use an ϵ -greedy policy. An ϵ -greedy policy uses an ϵ value to determine the probability of taking a random action instead of the best estimated one. For example if the ϵ value is 1 then 100% of the actions taken by the agent will be random, while if the ϵ value is 0 then 0% of the action taken is random.

During training, the ϵ value will decay by a predefined factor. By starting with a high ϵ value of 1 and decaying it over time, the RL agent explores in

```
Initialize Model and set parameters;
Initialize Target Model and set parameters;
while Learning is incomplete do
    while Episode is not done do
        if State is received then
            Save experience to memory;
            Perform experience replay ;
            Fit Model with experience from random memory batch;
            Estimate best action and send to Robot;
        end
        if Episode is Done then
            Fit Target model;
            Reset environment;
            Set new Episode to not Done;
        end
    end
end
```

Algorithm 1: Pseudo code explanation of the Reinforcement Learning algorithm

the early stages of learning, where the model has yet to be properly fit. With decreasing ϵ value, the RL agent starts to increasingly exploit as the model is being fit to perform well in the environment. The ϵ -greedy policy therefore ensures much exploration when the model is not ready to exploit yet, and makes it exploit more and more as it learns.

It is common to keep the ϵ at a minimum value, even late in the learning stages. This is to keep exploring the environment. If the agent stops exploring, it will perform the same actions, which it estimates are the best. This can cause it to hit what is called a local maximum in the achievable rewards. It has explored a strategy which is not the best, but it believes it is the best. By keeping a low ϵ value, these local maximums can be avoided, as the agent is forced to break its strategy. It can then explore new actions, which may prove to provide better rewards, finding a new reward maximum. An ϵ -greedy policy is used for this project with an epsilon decay of 0.99999 for each RL step.

4 Experimental Design

In this section, we will explain the experimental setup used to test the capabilities of the Unity3D game engine for training a RL model for the Turtlebot2, and using the resulting RL model to control the real counterpart of the Turtlebot2. The experiment is to test the capability of transferring the learning from the simulated environment to a real environment, and test the plug-and-play capabilities of changing from a simulated robot to a real robot.

4.1 Task of the Robot

To show the RL capabilities of the Unity3D simulation of the Turtlebot2 robot, we created an environment for the robot to perform a simple task. The environment consists of a small 2.5x5.0m room with 4 walls and a blue ball that is spawned at a random location in the room. The task for the Turtlebot2 is to drive into the ball using the shortest amount of time possible.

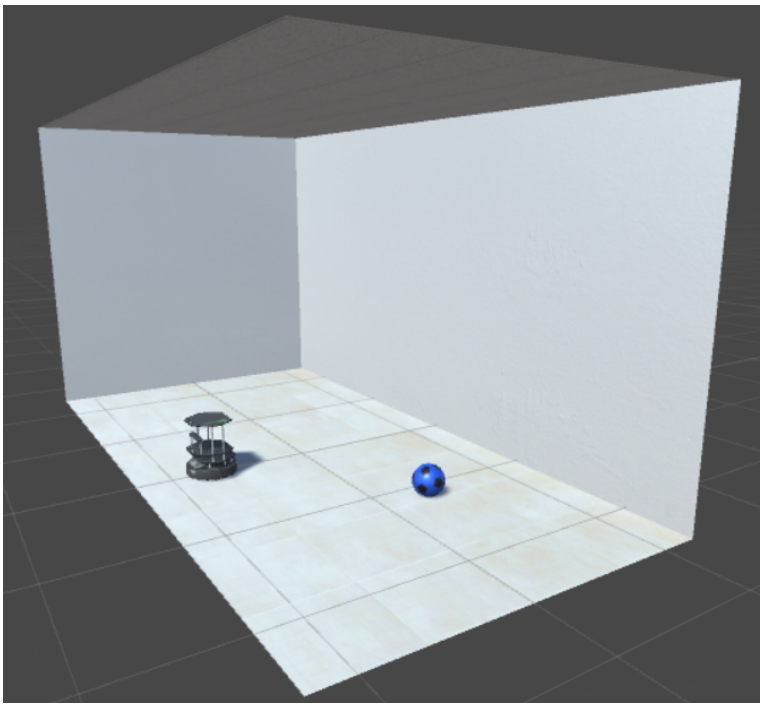

Figure 10: The Unity3D environment for the simulated robot. The task for the robot is to drive into the blue ball.

The task is designed to be simple, but still difficult for the robot to perform. The only knowledge of the environment the robot has is a 80x60 RGB image of its front view (see Figure 11). The algorithm has to learn to navigate the environment with only this knowledge of its state space. No pre-trained convolutional neural networks are implemented in this RL model. It therefore learns to control its movement towards the blue ball only from raw pixel data.

4.2 Rewards

Reward shaping is when extra intermediate rewards are given to the learning agent before the task is completed [28]. The intermediate rewards are shaped by the programmer to guide the robot towards its end-goal. Sparse rewards means that no rewards was given to the RL agent [29], unless the task was completed,

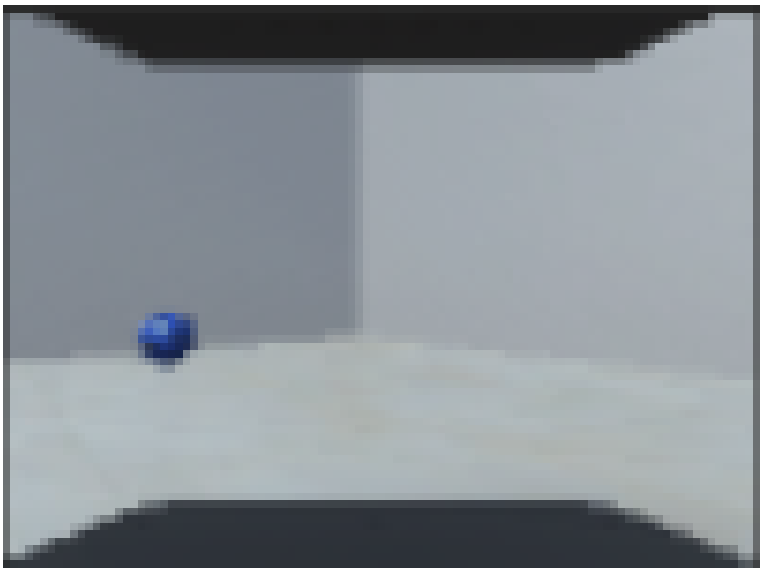

Figure 11: Example of an 80X60 RGB image sent from the simulated robot. This is the only input for the RL algorithm. Such an image corresponds to a state in the RL model.

e.g. the robot hit the ball. This can cause a slow start in learning, as the robot will have to perform uniformly random actions until it has accidentally completed the task and can learn from that.

A possibility to ease learning with sparse rewards is to use demonstrations. Večerík et al. [7] was able to train a robot arm for object insertion tasks, by providing it with human controlled demonstrations. The resulting model outperformed models using shaped rewards. However, providing demonstrations to a RL algorithm arguably defeats the purpose of self-learning.

In this project, the goal of the robot is to drive into the ball in the environment. Three different reward systems were tried. One with sparse rewards (SR), and two with reward shaping (RS1 and RS2). SR only provided the robot with a reward when it hit the ball. For RS1 and RS2, the rewards were shaped to encourage the robot to move towards the ball, and to frame the ball within the field of view of the camera while doing it, thereby providing it with the information that it is the collision with the ball that provides the best reward.

It is important to note that reward shaping inherently introduces human bias. As the programmer introduces their own idea of how to complete the task as the correct approach, they encourage the robot to perform its task a certain way, limiting its exploration of other possibilities.

The reward systems were as follows. For SR, the robot would only get a reward of 100 when it hit the ball and could achieve no other rewards. For RS1 and RS2, two different approaches were tried. If the robot collided with the

ball, RS1 would receive 100 points as reward and RS2 would receive 10 points. If the robots camera was looking at the ball and moving towards it, RS1 would receive 4 points, while RS2 would receive -0.1. If the robot was looking at the ball and moving away from it, RS1 would receive -2 points, and RS2 would receive -0.2 points. If the ball is not in the camera field of view or the robot stood still, RS1 would receive -4 points and RS2 would receive -0.4 points. For an overview of the reward systems, see Table 2. Each state the robot enters therefore has a corresponding reward. The corresponding rewards are therefore sent to the reward topic each time a new image is sent to the image topic.

Table 2: Reward systems

Rewards	Ball is hit	Ball seen moved towards	Ball seen moved away	Ball not seen or standing still
SR	100	0	0	0
RS1	100	4	-2	-4
RS2	10	-0.1	-0.2	-0.4

It is possible for the robot to gain rewards by colliding with the ball without the ball being in the field of view of the camera, e.g. by backing into it. This will not teach the model that the ball is the goal, as it has no knowledge of the ball being hit. The reward shaping systems are made to encourage the robot to look at the ball as much as possible. Consequentially, the robot is therefore more likely to see the ball when it collides with it - teaching it that it is the collision with the ball that provides it with the reward points.

4.3 Learning Setup

As mentioned before, one step of RL is done each time a new state (image) is received from the simulation. The RL script runs for a maximum of 200 steps for each episode. An episode is a sequence of RL steps, that ends when the goal has been achieved or a certain time has passed. At the end of an episode in our RL algorithm, the scene is reset and another episode begins. When the episode starts, the robot is set to the default position and the ball is spawned at a random location in the environment. It then proceeds to run the algorithm step by step in the new episode.

In the beginning of the RL, the epsilon value of 1 will cause the robot to perform entirely random actions. This forces it to explore the environment and learn from as many possibilities as possible. The epsilon value decays over time by a decay factor of 0.99999, to a minimum of 0.001 after about 420.000 steps or 9.5 hours of run time. This causes the robot to perform the estimated best actions by the model, except for 0.1% of the time. As mentioned before, keeping a low epsilon value to perform random actions helps exploring the environment, possibly avoiding local reward maximums in the model.

The learning need to stop at a point where the task could be consistently performed by the robot. For this project, we deem the robot consistent at

performing the task, when it has hit the ball 200 times in a row.

4.4 Transferring Learning to Real World

A RL model that has learned from enough experiences in the simulation, is fit with weights that will provide the best action to perform given a certain image state. When the learning is finished, the model can be saved to a single file. This file can then be used purely for predictions of actions given a certain image. This can be applied to a simulated robot for verification of the learning, or it can be applied to a real robot.

To show the capability of transferring learning from the game engine simulation to a real robot, the RL model taught in the Unity3D simulation is used to control a real Turtlebot2 robot. The connection with the Unity3D simulation and the RL algorithm is set up to be easily integrated with the real Turtlebot2. The simulated robot can therefore be replaced by the real robot counterpart with few adjustments. This is done by connecting to the Turtlebot2 instead of the Unity3D environment through the ROSbridge package. The input images from the Turtlebot2 is re-sized to 80x60 and fed to the RL model. The model then predicts the best actions to perform and sends them to the Turtlebot2, controlling the robot exactly as it controlled the simulated counterpart.

5 Results

The experiments are separated into two phases. The first phase is training a RL model with the simulated robot until it can consistently hit the ball in every episode. The RL algorithm was run until this was achieved. It was chosen that when the robot hit the ball 200 times in a row, it was deemed consistent at achieving its goal. The second phase is transferring the models taught in the simulation to the real environment and testing the models ability to control the real robot. This section describes the results of these experiments.

5.1 Simulation Results

To get the RL model to learn in this environment, three different reward systems was tried (see Table 2). A sparse reward system (SR) (see Section 4.2) was the first. The learning model did not converge successfully as the state space were too large to achieve enough random successes to learn from. The robot would therefore stand still after 10 hours of training, believing this as the best possible actions to take to achieve maximum rewards. This is probably due to the amount of random actions taken to be too few. It had therefore not gathered enough experience of hitting the ball to teach the model that rewards were given by doing that. The model could probably be taught with enough hours of random movement, but due to the sheer amounts of hours needed, the approach was deemed unsuccessful.

The first reward shaping system (RS1) (see Section 4.2) was able to perform the task but not consistently. The reward system was made to encourage the robot to move towards the ball, but due to the reward of moving towards the ball had a greater magnitude of 4 than the punishment magnitude of -2 from moving away from the ball, if the robot moved back and forth while looking at the ball, it would receive a surplus of rewards. The robot would therefore find the ball and then proceed to oscillate between moving forwards and backwards, only sometimes moving close enough to actually hit the ball. The reward shaping of RS1 was therefore deemed to be flawed, as it did not successfully guide the robot towards the ball.

The second reward shaping system (RS2) (see Section 4.2) was created to counteract the possibility to receive a surplus of rewards. All rewards the robot could receive was therefore negative, except when hitting the ball. This eliminated the exploitation of the previous reward system as the only way to get more rewards for an episode is to find the ball and drive into it quickly. This caused the robot achieve consistent completion of the task.

The simulated robot with the RS2 reward system was trained by the RL algorithm for 29.5 hours before hitting the ball 200 times in a row. During these 29.5 hours, the RL algorithm ran 975510 steps in 10322 episodes. To see the progress of learning for the RL see Figure 12. This graph shows the average score of the previous 100 episodes on the y axis and the number of episodes on the x-axis. The graph shows the score almost being consistent at around -10 at about 3000 episodes and forward. The RL algorithm achieved 100 successes in a row at 4316 episodes, or after 13.5 hours. The improvement of the system to achieve 200 episodes in a row therefore took about 16 hours.

In the graph it can be seen that some large dives in score are happening. This is due to the RL algorithm being restarted after it is paused. For example after the algorithm had achieved 100 balls hit in a row at episode 4316, it was automatically stopped to check that it was able to learn from the reward system. The algorithm was then restarted to achieve 200 hits in a row and be deemed consistent. The dive in performance is due to the experience memory being wiped. The RL algorithm therefore starts the training on batches with very few available experiences from memory. This causes the algorithm to temporarily overfit the model, thereby causing loss in score.

The algorithm was able to learn the task of finding and colliding with the ball. The RL model taught in the simulation was saved. The saved model contains the RL policy, enabling another robot to use the model for action prediction. An action prediction script was created, where the RL model was imported to estimate the best actions from the input images. The RL model in the action prediction script did not learn from the experiences. Testing the action prediction script in the simulation, showed that the robot was able to find and collide with the ball on every try. However, to test the capability of transferring learning from the game engine simulation to a real robot, the model was used for action prediction of a real Turtlebot2 robot.

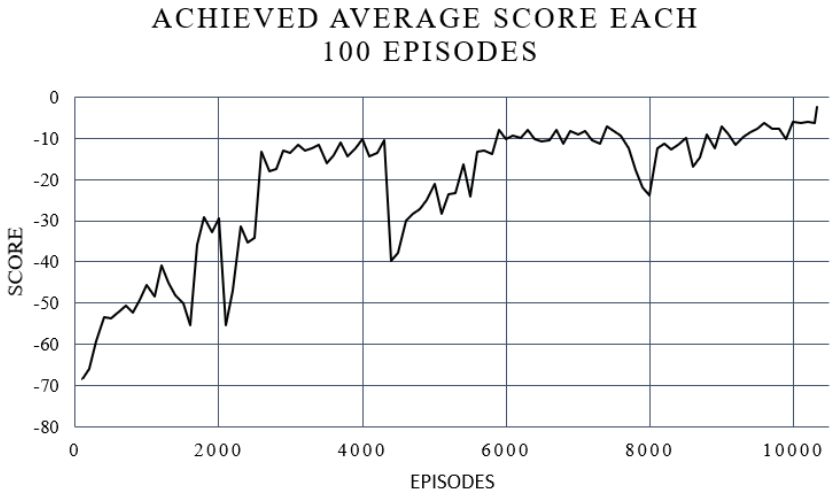

Figure 12: Graph of the score achieved with the RS2 reward system. Each data point is the average score of the previous 100 episodes.

5.2 Real Robot Results

The primary goal of the experiment was to test the plug-and-play capabilities the game engine had when changing from a simulated robot to a real robot. We are looking at the systems ability to read the input and control the robot without the need to change anything expect for connecting to the real robot. The secondary goal was to test the transfer of learning from the simulation to a real counterpart and how well it would perform in a real environment.

For the prediction script the only changes needed was to change the names of the topics the subscriber and publisher, and to re-size the input image from the real robot to 80x60. The names of the topic had to be the exact same names as the topics used by the real Turtlebot2. The names of the topics in the simulation was slightly different to ensure that information was not sent and received from the wrong robot. The image sent from the Turtlebot2 robot is 640x480. The RL model only takes images of size 80x60. The images from the Turtlebot2 is therefore re-sized to fit the RL algorithm. This caused the script to obtain usable images from, and send actions to, the real Turtlebot2. When these names were changed and images re-sized, the robot was fully controllable by the RL algorithm when connected.

The trained model was loaded by the prediction script, and used to tell the robot which actions to take. The robot was then placed in a real environment (see Figure 13). A blue ball similar to the one in the simulated environment was placed in the environment. The real environment was a clear floor area with furniture and lab equipment around it. Some of the floor was covered in white

Figure 13: The environment used to test the real robot with the trained model from the simulated environment. The robot was generally unable to locate and drive into the blue ball. This were probably due to the simulated environment being too different from the real environment.

plastic and some of the floor area was dark grey. The real environment therefore did not look as the simulate environment, and the learning was expected to not to successfully transferred to the real robot.

The model was able to control the robot, but the robot would not act as in the simulated environment. It would turn around to search for the ball, but with much more inconsistent movements, turning back and forth a lot before turning slightly more to one side. It did, however, seem to recognize the ball at times, as the robot would sometimes stop its slight turning when the ball was in front of it. It did not go towards the ball, but would jigger back and forth in incoherent movements.

Moving the robot to a position where only a white wall and white floor area was visible for the camera, made the robot able to recognize the ball and move towards it (see Figure 14). The white walls and floor looked similar as the environment in the simulation. We hypothesis that as this part of the real environment was similar to the simulated environment, the training was transferable in this specific part of the real environment. To see the behaviour of the real robot controlled by the RL model, a YouTube video can be seen here: `https://youtu.be/PV2imw-lxH4`

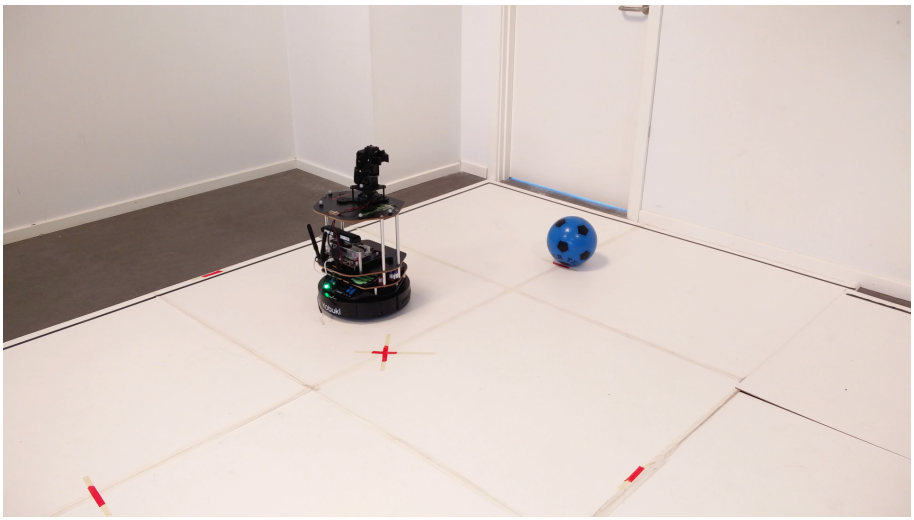

Figure 14: When putting the robot with a white wall and white floor in its view, such as in the simulated environment, the robot was able to recognize the ball and drive into it.

6 Discussion

6.1 Interchangeable robots

The learning model was transferred to the real robot and able to control the real Turtlebot exactly like the simulated counterpart. The ROS compatibility with Unity3D made the message act the same in the simulation as for the real robot. Therefore, the simulated robot was easily interchanged with the real Turtlebot2. This shows the capabilities of using a game engine to simulate the robot. Robots are easily implemented in the game engine, while having a great physics engine, ability to render realistically, create realistic replications of real environments, and providing the ability to customize lighting conditions.

The focus of the report is to show the plug-and-play capabilities of using a game engine to simulate a robot, and the successful control of the real robot counterpart with only few adjustments highlights the usefulness of modern game engines as robot simulators. The experiment therefore support the use of game engines for simulating a robot for reinforcement learning due to the easy interchanging between real and simulated robots, and the ability to realistically mimic the real environment of the robot.

6.2 Transferring learning

For this project, the simulated training environment for the RL algorithm did not realistically replicate the real world environment. As a consequence, the Turtlebot2 was not able to complete the task of hitting the blue ball in the real

environment. This underlines the importance of simulating the real environment realistically to enable the learning to be usable in a real setting. Also, the lighting conditions of the simulated environment need to replicate the real environment's lighting, as the pixel values of the input image are highly dependent on the lighting on surfaces. Another possibility is to randomize the lighting conditions in the simulation, thereby generalizing the RL to many different lighting conditions, including the potentially changing lighting conditions of the real world environment.

7 Future Work

7.1 Reward System

The reward system of the RL algorithm was essential for the robot to learn to hit the ball. It is therefore important for future works to implement a working reward system. For this project, the RS2 reward system (see section 4.2) was essential for the robot to learn how to perform the task consistently. However, it is possible that another reward system would increase the effectiveness of learning.

For example, the robot was taught that having the ball in sight is better than not, even when moving away from it. This is because backing away from the ball would provide less penalty points than not looking at the ball. For improvement of the RS2 reward system, having the reward penalty be -0.1 only when moving towards the ball and looking at it and -0.4 in every other occasion could improve learning. This would discourage the robot to move away from the ball as an attempt to minimize penalty. If the penalty for not looking at the ball and for looking at and moving away from the ball was the same, the only way for the robot to minimize penalty would be to move towards the ball. Therefore, we hypothesize that a reward system where penalty points are only minimized when looking at the ball and moving towards it, could improve learning.

7.2 Realistic Rendering and Randomizing Light Conditions

Using realistic rendering and replication of the real environment and randomizing light conditions is an important future experiment for this research. In the current experiment, the simulated and real robot sends RGB images of size 80x60 pixels. This raises the question if realistic rendering even matters when the details are scaled down to such a small image.

Future experiments with highly realistic rendering, dynamically changing lighting, and high res image as input for the RL algorithm is essential to prove the need for realistic rendering, and thereby the usability of the game engine as a robot simulator.

7.3 Improving Robot Controls and Learning Algorithm

The Q-learning algorithm used for this project is said by Lillicrap et al. [20] to not be applicable for continuous action spaces. A robot is conventionally controlled by continuous action spaces, as each action such as velocity control, can be controlled with small floating point values. The Q-learning algorithm in this project chooses between 9 actions to perform, while a conventional robot control would provide a very specific value for velocity control.

For future work, the implementation of a RL algorithm such as the DDPG with capability of estimating continuous actions will provide the robot simulation with more conventional control, which would help the robot navigate better in continuous space.

The Q-learning algorithm also uses more computation to learn in continuous spaces than models such as the DDPG actor-critic model. It is believed to be essential to implement a more fitting model for continuous spaces, if the robot is to learn more complex tasks with higher resolution images as input. We believe that for future works the use of DDPG would prove beneficial to teach a robot to perform tasks that are more complex than the current.

7.4 Test against Gazebo and V-rep

Another future work is to test the RL learning capabilities of using a game engine such as Unity3D against the RL capabilities of other robot simulators such as Gazebo and V-rep. By implementing the same environment to the best ability of the simulators and game engine, and using the same RL algorithm, the resulting learning models is then transferred to a real robot. Testing if the realistic rendering of the game engine would improve the learning of the RL algorithm could further support the usage of game engines as robot simulators.

8 Conclusion

In this project, the Unity3D game engine was used to enable fast, safe and cheap reinforcement learning for a simulated Turtlebot2 robot. The focus of the report was to emphasize the plug-and-play capabilities of using a game engine as a realistic simulation for a robot to act in without the risk of causing damage while learning.

The communication between the simulated robot and a Q-learning reinforcement algorithm was controlled through the ROS middleware. The simulated robot was trained with raw RGB pixel data from a mounted camera as its only knowledge of the environment, to perform the task of driving into a blue ball. After 29.5 hours of training, the simulated Turtlebot2 was able to collide with the ball 200 consecutive times. The resulting reinforcement learning model was saved and used to predict the correct actions of a real Turtlebot2 robot in a real environment, with the intention of completing the same task.

The real Turtlebot2 was interchangeable with the simulated counterpart for the reinforcement learning system. The plug-and-play capabilities of the created

game engine system made it easy to train a reinforcement learning model in the simulation and use that model to control a robot in a real environment. The game engine simulation was able to simulate the robot with realistic and precise controls.

The learning from the simulation were not successfully transferred to the real robot. This is believed to be due to the simulated environment not realistically resembling the real environment for the robot. Additional testing, with realistic rendering and changing lighting conditions in the game engine simulation, and a comparison with the reinforcement learning transfer performance of other robot simulations, are proposed future work for the project to further assess the usability of game engines for simulating robots for reinforcement learning.

References

[1] L. Tai, G. Paolo, and M. Liu, "Virtual-to-real deep reinforcement learning: Continuous control of mobile robots for mapless navigation," in *Intelligent Robots and Systems (IROS), 2017 IEEE/RSJ International Conference on*, pp. 31–36, IEEE, 2017.

[2] Y. F. Chen, M. Everett, M. Liu, and J. P. How, "Socially aware motion planning with deep reinforcement learning," *arXiv preprint arXiv:1703.08862*, 2017.

[3] L. Pinto and A. Gupta, "Supersizing self-supervision: Learning to grasp from 50k tries and 700 robot hours," in *Robotics and Automation (ICRA), 2016 IEEE International Conference on*, pp. 3406–3413, IEEE, 2016.

[4] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, *et al.*, "End to end learning for self-driving cars," *arXiv preprint arXiv:1604.07316*, 2016.

[5] W. Schultz, P. Dayan, and P. R. Montague, "A neural substrate of prediction and reward," *Science*, vol. 275, no. 5306, pp. 1593–1599, 1997.

[6] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*, vol. 1. MIT press Cambridge, 1998.

[7] M. Večerík, T. Hester, J. Scholz, F. Wang, O. Pietquin, B. Piot, N. Heess, T. Rothörl, T. Lampe, and M. Riedmiller, "Leveraging demonstrations for deep reinforcement learning on robotics problems with sparse rewards," *arXiv preprint arXiv:1707.08817*, 2017.

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

[9] S. Levine, P. Pastor, A. Krizhevsky, and D. Quillen, "Learning hand-eye coordination for robotic grasping with large-scale data collection," in *International Symposium on Experimental Robotics*, pp. 173–184, Springer, 2016.

[10] M. Johnson-Roberson, C. Barto, R. Mehta, S. N. Sridhar, K. Rosaen, and R. Vasudevan, "Driving in the matrix: Can virtual worlds replace human-generated annotations for real world tasks?," in *Robotics and Automation (ICRA), 2017 IEEE International Conference on*, pp. 746–753, IEEE, 2017.

[11] T. Kounalakis, G. A. Triantafyllidis, and L. Nalpantidis, "Weed recognition framework for robotic precision farming," in *Imaging Systems and Techniques (IST), 2016 IEEE International Conference on*, pp. 466–471, IEEE, 2016.

[12] Z. Wang, Z. Liu, Q. Ma, A. Cheng, Y.-h. Liu, S. Kim, A. Deguet, A. Reiter, P. Kazanzides, and R. H. Taylor, "Vision-based calibration of dual rcm-based robot arms in human-robot collaborative minimally invasive surgery," *IEEE Robotics and Automation Letters*, vol. 3, no. 2, pp. 672–679, 2018.

[13] T. K. Stephens and J. J. O'Neill, "Ronny boards the elevator: Toward multi-floor navigation," *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, p. 3111, 2017.

[14] K. Pfeiffer, A. Escande, and A. Kheddar, "Nut fastening with a humanoid robot," in *Intelligent Robots and Systems (IROS), 2017 IEEE/RSJ International Conference on*, pp. 6142–6148, IEEE, 2017.

[15] ORSF, "Why Gazebo?." `http://gazebosim.org/`, 2018. [Online; accessed 27-April-2018].

[16] C. Robotics, "V-REP virtual robot experimental platform." `http://www.coppeliarobotics.com/`, 2018. [Online; accessed 13-May-2018].

[17] J. Tremblay, A. Prakash, D. Acuna, M. Brophy, V. Jampani, C. Anil, T. To, E. Cameracci, S. Boochoon, and S. Birchfield, "Training deep networks with synthetic data: Bridging the reality gap by domain randomization," *arXiv preprint arXiv:1804.06516*, 2018.

[18] Unity Technologies, "Unity homepage." `https://unity3d.com/`, 2018. [Online; accessed 27-April-2018].

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

[20] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[21] I. Chen, "Gazebo Rendering Abstraction." https://www.osrfoundation.org/gazebo-rendering-abstraction/, 2015. [Online; accessed 27-April-2018].

[22] NVIDIA, "Introducing NVIDIA Isaac." https://www.nvidia.com/en-us/deep-learning-ai/industries/robotics/, 2017. [Online; accessed 26-April-2018].

[23] J. Marin, D. Vázquez, D. Gerónimo, and A. M. López, "Learning appearance in virtual scenarios for pedestrian detection," in *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pp. 137–144, IEEE, 2010.

[24] J. Xu, D. Vázquez, A. M. López, J. Marín, and D. Ponsa, "Learning a part-based pedestrian detector in a virtual world," *IEEE Transactions on Intelligent Transportation Systems*, vol. 15, no. 5, pp. 2121–2131, 2014.

[25] D. Vázquez, A. M. López, and D. Ponsa, "Unsupervised domain adaptation of virtual and real worlds for pedestrian detection," in *Pattern Recognition (ICPR), 2012 21st International Conference on*, pp. 3492–3495, IEEE, 2012.

[26] D. Vázquez, A. M. López, J. Marín, D. Ponsa, and D. Gerónimo, "Virtual and real world adaptation for pedestrian detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 36, pp. 797–809, April 2014.

[27] M. Bischoff, "Announcing ROS#." https://rosindustrial.org/news/2018/1/8/announcing-ros, 2018. [Online; accessed 13-May-2018].

[28] T. Brys, A. Harutyunyan, H. B. Suay, S. Chernova, M. E. Taylor, and A. Nowé, "Reinforcement learning from demonstration through shaping.," in *IJCAI*, pp. 3352–3358, 2015.

[29] L. El Asri, R. Laroche, and O. Pietquin, "Reward shaping for statistical optimisation of dialogue management," in *International Conference on Statistical Language and Speech Processing*, pp. 93–101, Springer, 2013.