Simulation to Real World Knowledge Transfer

Áron Zoltán Váradi
STUDENT NUMBER: 2028407

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE IN COGNITIVE SCIENCE & ARTIFICIAL INTELLIGENCE
DEPARTMENT OF COGNITIVE SCIENCE & ARTIFICIAL INTELLIGENCE SCHOOL OF
HUMANITIES AND DIGITAL SCIENCES
TILBURG UNIVERSITY

Thesis committee:

prof.dr.ir. Pieter Spronck

dr. Merel Jung

Tilburg University
School of Humanities and Digital Sciences
Department of Cognitive Science & Artificial Intelligence
Tilburg, The Netherlands
June 2020

Table of contents

Simulation to Real World Knowledge Transfer

Áron Zoltán Váradi

In this study we investigated simulation to real world knowledge transfer. We trained an agent to drive a car around a track in a simulated environment using the Unity ML-Agents Toolkit. Then, to improve our agent, we tested how different inputs, rewards and outputs influence training. Finally, the agent was used to control an RC car in the real world. Similarly to the research of Andrychowicz and colleagues (2020), we achieved successful knowledge transfer, but instead of a robotic hand, we used a low-cost environment, while, also testing the capabilities of a novel tool in reinforcement learning (ML-Agents).

1. Introduction

1.1.Context and relevance

Deep reinforcement learning (DRL) has great potential in robotics domain, but there are several constraints of the real world that have to be overcome first. Simulation to real world knowledge transfer is one possible way to realize this potential (Rusu et al., 2016). Our research is mainly based on three previous studies (Andrychowicz et al., 2020; Rusu et al.,2016; and Tobin et al., 2017), all of which are focusing on the transfer of knowledge generated using DRL, from a simulated to a real environment. These experiments involved expensive hardware like the Fetch robot for Tobin or the Shadow Dexterous Hand for Andrychowicz.

In this paper we propose and test a paradigm, which is able to demonstrate sim-to real knowledge transfer in a more economical setting. This task also enables us to explore the capabilities of a relatively new tool for DRL — Unity Machine Learning Agents Toolkit (ML-Agents) — which claims to be a high quality environment for AI research and development (Juliani et al., 2018).

The goal of the project is to create a self-driving car analogue, which learns to drive using DRL. In order to achieve our goal, we connect a remote controlled car (RC car) to a computer, construct a track for it and place a camera on top. In this way, we can use DRL to train an agent to drive around a track. However, in the real world there are often obstacles in the way of training. There can be only a small number of agents training at the same time which means no large scale parallel training is feasible. Random exploration employed by DRL can endanger the real world components (Tobin et al., 2017) – for example, if we drive the car into the wall of the track, it might break. There are other physical limitations, such as the car's battery which is only able to power the car for 15-20 minutes.

The alternative to train the agent in the real world is to train it in a simulated environment and transfer the learnt knowledge to do the real world task. This ensures more favorable training circumstances, but brings up new problems. We cannot have a good enough approximation of the real world, because we do not have all the information (for exp.: max speed, acceleration curve, drag). Hence, there will be significant differences between the simulation and the reality. Therefore, our model should be robust to be able to cope with these differences. We surmise that this can be achieved by domain randomization (DR) (Tobin et al., 2017) which means adding variability to those parameters which are most likely to be affected by the sim-real differences.

The goal of the agent is to drive an RC car around a simple track in the real world. In order to train an agent capable of this task, first we created a simulated environment in the Unity game engine. After which we added randomness to certain environmental parameters, while at the same time trained the agent to make it more robust (DR). After the training, our agent was able to control the RC car in the real world.

The agent can receive two different types of input: visual or vector. Visual input refers to a constant image feed of the car captured from above the car. Vector input can be the position, direction, and/or velocity of the car. In the real world, visual input is generally easier to obtain (only a camera is needed), while in order to acquire vector input we need sensors, or we need to do preprocessing on the visual input. For the agent vector input is easier to use, training is quicker and requires less resources. As opposed to visual input, where the agent has to learn how to do preprocessing by itself, which multiplies the difficulty of the training.

1.2.Research Questions

In the following section we present our research questions and show how we plan to answer them.

I) Can we train the agent with visual input?

II) Can we train the agent with vector input?

  (a) How does the agent perform in the simulated environment?
  (b) How can we make the agent better?

III) Can we use the agent in the real world environment?

  (a) Is an agent trained without DR able to drive in the real world environment?
  (b) Is an agent trained utilizing DR able to drive in the real world environment?
  (c) Is training with DR better for knowledge transfer than training without

We divided the task at three segments:

In the first segment, we attempted to train the agent on visual input (I). We used PPO algorithm in the Unity ML-Agents with the default training parameters. Working with visual input is more difficult than with vector input and we did not get adequate results to continue this line of research.

In the second segment, we trained an agent with vector input in the simulation using Unity ML-Agents (II). We utilized the default algorithm in the package (PPO) and kept the default settings for the algorithm. We then measured the agent's performance (II/a). We wanted to maximize our chance for successful knowledge transfer, thus, we aimed to improve our agent (II/b). For this reason, we needed to find a viable reward structure, model input and model output.

In the last segment, we transferred the agent to the real world environment (III). We first trained an agent in the simulation (without using DR) and used it to drive in the real world environment (III/a). Then we trained an agent using DR, and used this agent to drive the RC car (III/b). Finally, we compared the performance of the two agents.

These experiments served as a base to measure how well Unity ML-Agents could be used as a platform for research, and whether our experimental paradigm could be the basis of further research in the field of simulation to real world knowledge transfer.

2. Related Work

During reinforcement learning, the agent learns which action to choose in which situation in order to maximize future rewards. To be able to do this, the agent receives rewards — in the form of positive or negative numbers, and information about the environment (Sutton & Barto, 2018). To solve problems in high dimensional spaces, reinforcement learning can be supplemented with deep learning techniques (François-Lavet et al., 2018). This combination of reinforcement learning and deep learning is called deep reinforcement learning, and it is used for mastering complex tasks such as playing Go (Silver et al., 2017) or Poker (Brown and Sandholm, 2017; Moravcik et al., 2017).

Reinforcement learning can be also used to solve real world problems — for example in the robotics field. However, it is greatly limited by sample inefficiency. This is caused by two main factors: (1) many algorithms start learning from scratch; (2) the selection of appropriate information is difficult, and the current algorithms are not that effective exploiting it. Because of sample inefficiency, it is necessary to generate a lot of data via exploring the environment. However, in the real world exploration is time consuming, and possibly dangerous. Doing random actions can hurt both the agent (for example a robot), and its environment (Nguyen & La, 2019). A solution for this problem is to train the agent in a simulated environment, then use same agent for the real world task. This method is called transfer learning, and it has many applications besides reinforcement learning. Transfer learning can be used in cases where there is not enough training data, and there is a model already trained in a similar domain. (Zhuang et al., 2019)

Transferring knowledge from a simulated environment to a real world environment is difficult, because the discrepancies between the two environments. This problem is called reality gap. (Pinto et al., 2017; Rusu et al., 2016). We can decrease the reality gap by domain adaptation – creating a more realistic environment, or cross the gap with robust agents. One popular way for creating robust agents is domain randomization (e.g., Tremblay et al., 2018; Mehta et al., 2020; Zhang et al., 2017).

Our research was partly based on the paper of Andrychowicz and colleagues (2020). They trained a robotic hand in a simulation, then successfully transferred the learnt policies to a real world robot. They used proximal policy optimization (PPO) for training, with extensive randomization to create a robust model. They separated the policy and the vision system. This means that the vision system was trained using supervised learning, and only the policy network needed to be trained via DRL.

Game engines, such as Unity, are often used to create simulated environment for reinforcement learning (e.g., Nakamura & Yamakawa, 2016; Aguero Quinteros, 2019). Some of these studies focus on simulation to real world knowledge transfer (for exp.: Lillelund, 2018; Andrychowicz et al., 2020). In 2017, Unity ML-Agents Toolkit was released, in part to help facilitate the research of reinforcement learning. ML-Agents is an open source package, which provides an easy-to-use interface for reinforcement learning, while utilizing the simulation complexities of the game engine (Juliani et al., 2018). Since its initial release in 2017, it was already employed in reinforcement learning research (e.g., Jain & Lindsey 2018; Burda et al., 2018; Min et al., 2019).

In our research, we train an agent with the Unity ML-Agents Toolkit using deep reinforcement learning. We then make the agent more robust using domain randomization, and transfer the agent to the real world.

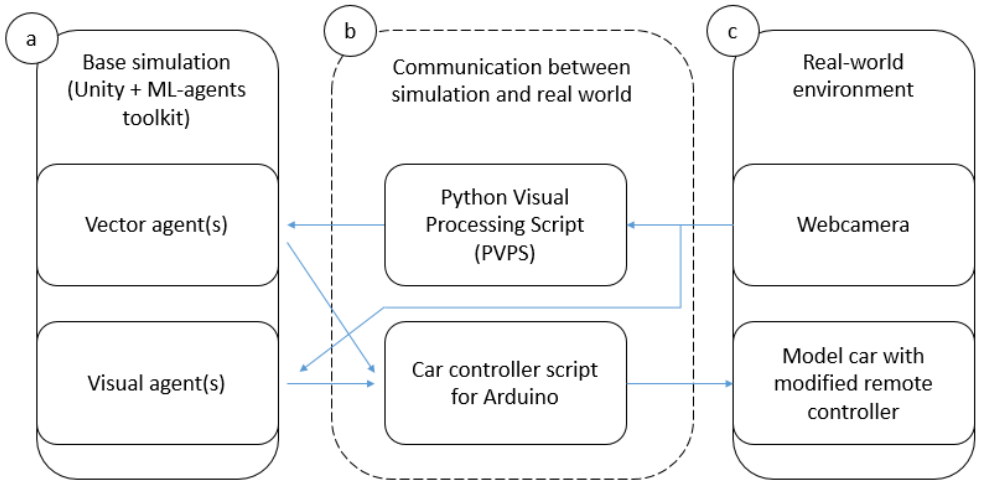
3. Method

The references to all the written codes and used packages with citations can be found in **Appendix A**.

3.1. Infrastructure

In this section we show how we used the agents in Unity to drive the RC car. Figure 1 was created to aid the understanding of this process. After the explanation we examine each part in detail.

Figure 1.
The agents can only work with the unity game engine (a), therefore we created an interface (b) through which they are able to communicate with the real world environment (c).


a
Base simulation (Unity + ML-agents toolkit)
Vector agent(s)
Visual agent(s)
b
Communication between simulation and real world
Python Visual Processing Script (PVPS)
Car controller script for Arduino
c
Real-world environment
Webcamera
Model car with modified remote controller

Our goal is to train an agent to drive a car around a track in a simulated environment, and then, test how that agent performs in a real world environment. In order to achieve this goal, we created a car simulation in the Unity game engine (3.1.1). We trained different agents in this environment using the ML-Agents toolkit. For inference, we either used the simulation, or connected the agents to the real world. We used two types of agents: visual agents – these can work directly with image input, and vector agents – these require preprocessed input (for exp.: positional and velocity data). Data preprocessing is done with the Python Visual Processing Script (PVPS)

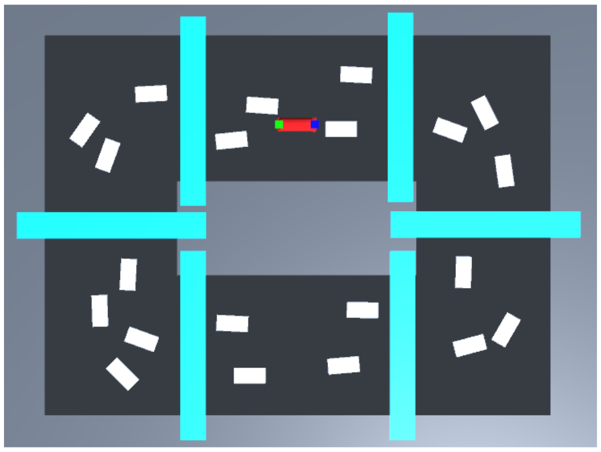
(3.1.2), and its results are forwarded to Unity through a dedicated socket. Agents can pass information to the real world car, because the real-world car is connected to a microcontroller (3.1.3) which is connected to the computer through a serial port.

3.1.1. The Unity project

The Unity environment consists of the car simulation, and the agent which is managed via the ML-Agents toolkit. The car simulation is conducted with wheel colliders, which are the native way in Unity to create 3D car simulations. With this method, the game engine creates the simulation in the background, only some top-level parameters need to be set up. Such as the mass of the components (wheels, body of the car), the strength coefficient (the maximum force that is applied to the car), and friction and suspension.

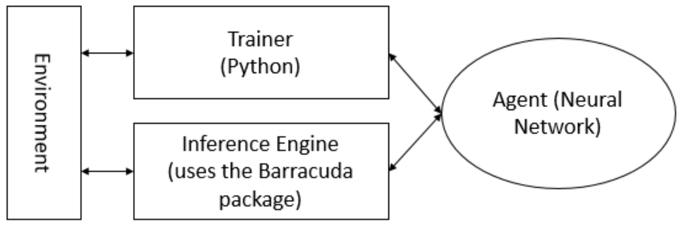
The car should follow a track (see Figure 2.) which is in a 640:480 rectangular shape. This shape was chosen to match the ratio of the real world camera's aspect ratio. The track is separated into sections by 6 checkpoints, which detect if the car passes through them. We use these checkpoints to control when to give rewards to the agents. There are also several waypoints scattered around the track. These are used as reset points for the car. When the agent makes a mistake (for exp.: it drives the car into the wall of the track), a waypoint is chosen randomly and the car resets there (resetting at a random waypoint is more effective than always resetting to the same point as shown in **Appendix B**).

Figure 2.
The simulated track from a top down view. The car is marked red, with the blue dot marking its front and the green marking its back. The white rectangles are the waypoints, where the car resets after hitting the edge of the track (grey), or after a given time. The agent can receive reward for reaching and/or getting closer to the next checkpoint.



The car is controlled by an agent. The agent is a neural network, which communicates with the environment through the trainer (during training) or through the inference engine (when doing inference) (Figure 3). We can set certain parameters of the agent in the Unity environment (e.g., type and size of outputs, rewards, when to reset) and can provide information to the agent, give it rewards or punishments when necessary. The hyperparameters for the agent however has to be set in separate file, which is then read by the Trainer during training.

Figure 3.
The agent's pipeline



3.1.2. Processing visual input with open-cv Python Visual Processing Script (PVPS)

The purpose of the visual processing script is to access the camera's output, find the center of two points on the car (marked with a blue and a green sticker), and transmit this data to the agent in Unity. In order to do this, the script relies on the cv2, imutils and numpy packages for the image processing and on the zmq package for data transmission.

The PVPS runs in a continuous loop. In each iteration the script grabs a frame from the camera's feed. The frame then converted to the HSV (Hue, Saturation, Value) colorspace. The HSV colorspace is preferred for object tracking, because it is less sensitive for changes in lightning conditions than the RGB colorspace (Ali, 2013). After the color conversion PVPS creates a mask for both dots and calculates their centerpoints. Finally, the coordinates of the centerpoints are broadcasted to Unity through a dedicated socket.

For this broadcasting we utilize the universal messaging library ZeroMQ (Hintjens, 2013). With ZeroMQ we establish a simple publisher-subscriber protocol through a dedicated socket. The PVPS acts as a server, which broadcasts the four coordinates (car front x/y, car back x/y) for each grabbed frame. A script in Unity subscribes to this broadcast, receives the coordinates, and places them inside four variables. These variables can be used as input for an agent.

3.1.3. Controlling the real-world car

The model car used in this experiment is a radio controlled 1:87 VW Transport. The remote for the car contains two potentiometers which regulate the signal for the speed and turning. We replaced the analogue potentiometers with digital ones that can be regulated by an Arduino Nano microcontroller. The microcontroller was programmed to be able to interpret and execute commands from a computer. The commands are sent through a serial port, and they contain which potentiometer to set, and to what value.

3.2.Used model

Proximal policy optimization is a group of policy gradient methods for DRL, developed by Schulman and colleagues (2017). PPO is natively implemented in the ML-Agents Toolkit (Juliani et al., 2018). With this package it is possible to train and deploy agents in a Unity-based simulation.

3.2.1. Measurements

Measurements in the simulation

Average cumulative reward (ACR) is measured automatically during training. In order to calculate ACR, all the rewards received by the agent are summed up for each episode, then divided by the number of episodes. The agent's goal is to maximize the reward, therefore, average cumulative reward measures what the agent was trained for. To be able to compare agents with different reward structures, we use the default ACR. This means, during testing we change all agent's reward structure to the default agent's (the default agent is explained under 3.3.1).

We can compare different agents using the ACR measurement directly. However, the ACR measurements are difficult to put in context. As a solution, we can convert ACR to Average consecutive laps. This can be achieved through dividing ACR by 7.4 – as an agent collects a reward of approximately 7.4 for completing a lap. Even though this is only a close approximation of the real average consecutive laps, since they are calculated with a simple division, the proportions of the different agents' performances stay the same.

Real world measurements

In the real world environment, we measure average consecutive laps manually. We count how many laps the agent can complete before making contact with the walls of the track. Unlike lap times, this measurement is insensitive to the speed of the car which was important, because the speed is not controlled by the agent, but constantly changes as the battery drains. Using average consecutive laps is possible, because the agents are imperfect, and sooner or later they collide with the wall — ten lap was the maximum we measured.

3.3.Experimental Design and Procedure

As we presented in the introduction, our research consists of three sections: I) Testing whether an agent can train successfully in the simulation using visual input alone; II) Training and agent using vector input, and improve on it; III) Taking the agents trained in the simulation (with and without utilizing DR), and testing them in the real world.

Before we can address the research questions we set up a default agent, to serve as a base for other agents. All other agents are modifications of the default agent. The default agent has 4x2 vector input – the coordinates of two points of the car (one at the front, one at the back), at two consecutive time steps (t and t-1). The default agent receives combined reward: a reward of one for every checkpoint reached and a small reward for every step in the right direction. Reward for going into the right direction is calculated by measuring the changes in distance for each coordinate (x, y) for the car, and multiplying it with 0.1 to keep the amount of reward small. It can be calculated as:

$$((d_{t-1}(carX, cpX) - d_t(carX, cpX)) + (d_{t-1}(carY, cpY) - d_t(carY, cpY))) * 0.1$$

Where d denotes distance, t denotes the current $t\text{-}1$ the previous time step and cp denotes the next checkpoint. There are two conditions when the agent resets: (1) The agent resets every time that it hits the wall, and if that does not happen, (2) and it resets at every 60 seconds. We set the first constraint because our agents cannot go in reverse. If the car hits the wall in a corner, it should reverse and try to take the corner again. Since the car cannot go in reverse, we reset the agent each time it hits the wall. The second constraint was set to stop the agent from driving indefinitely.

After each reset, the agent starts at a random waypoint (this is a better method than resetting to the same starting point see Appendix B). In order to keep our model simple, we constrain the agent to do three possible discrete actions: turn left or right or keep on going straight. We do not let the agent control the speed.

For the experiments we utilized the PPO algorithm with the given default parameters (see Appendix C) except for the number of steps, which we set to 4,000,000. These parameters apply for all of our models unless specified otherwise.

3.3.1. Experimenting with visual inputs

We want to know if it is possible to train an agent using ML-Agents to control a car in the simulation. To test this, we trained an agent on a simple 50x28 pixel grayscale image as input (this is the smallest size where the contours of the car are still clear). Apart from changing the input from vector to visual, this agent was identical to the default agent.

The goal of this setup was to create a simple visual representation, and see whether the agent is able to learn within a reasonable timeframe. We wanted to use it as a stepping stone for further experiments. However, this experiment was not successful, therefore we did not continue this line of research.

3.3.2. Experiments with vector input

Our aim is to improve the vector agents, thus, we devised three experiments. These experiments concentrate on different aspects of the agent: (I) reward structure; (II) input types; (III) output types. We did not experiment with tuning the PPO related hyperparameters, since there are several of them (complete list is shown in Appendix C), and training an agent took around 3.5 hours when training for 4 million steps.

I) In our first experiment, we compare the agents trained on three different reward structures:
   a) Big reward only: In this condition we give a big reward of 1.0 to the agent, each time it reaches the next checkpoint.
   b) Small reward only: The agent receives a small reward each time step, when it moves in the right direction.
   c) Combined reward: In this condition the agent receives both the small and the big reward.
   d) Combined reward with punishment (Default agent): In addition to the previous case, the agent also receives a punishment of -1.0 each time it hits the walls of the track.

II) In our second experiment we compare the effect of different types of vector inputs on learning:
   a) Velocity and position: The agent receives the velocity of the car (separately in x and y directions), and the coordinates of a single point of the car.
   b) One point, single time step: The input consists of the coordinates (x, y) of a single point of the car, at the current timeframe.
   c) One point, two time steps: In addition to the previous case, the agent also receives the coordinates for the previous time step.
   d) Two points, single time step: The input consists of the coordinates (x, y) of two points of the car, at the current timeframe.
   e) Two points, two time steps (Default agent): In addition to the previous case, the agent also receives the coordinates for the previous time step.
   f) Normalization: this condition is the same as the default, except the range of inputs were modified to fit inside the -1 / +1 range.

III) In the last vector input experiment we compare three different methods to control turning.
   a) Discrete 3 (Default agent): The agent can go straight, left or right. Both left and right turn uses the maximum turn angle (30°).
   b) Discrete 5: The agent can go straight, turn left or right with the maximum turn angle, or turn left or right with half of the maximum turn angle (15°).
   c) Continuous: The agent has precise control over how much it turns.

3.3.3. Knowledge transfer

There are several discrepancies between our simulation and the real world environment. These can be categorized into two groups: image based and environment based.

In the image based category we have at least five sources of discrepancy. Because of (1) camera distortion, the coordinates derived from the camera feed will not align up perfectly with the real life positions. It is also possible that (2) the camera slightly rotates on one or more axis, which adds to the misalignment of the coordinates. (3) Image tracking errors can cause the coordinates of the middle points to move around. There is also (4) a shift between the image processing and Unity's cycle. This can cause the coordinates at t and t-1 to become equal, thus rendering velocity estimation impossible. Processing an image stream is not completely real time, there is (5) latency both between (camera to PVPS, PVPS to Unity) and within (PVPS, Unity) the components.

There are at least three possible environmental — simulation to real world — differences: (1) maximum turning degree, (2) maximum speed, and (3) turn bias. Turn bias means the car is turning slightly in one direction when it should go perfectly straight. There is also a general inconsistency – all the variables mentioned above can change when the car consumes battery, or when the remote calibrates itself.

Our goal is to create a model which is insensitive to these errors and biases, and we try to achieve this with domain randomization. We add randomizations to the three variables which are most likely to be affected by the sim-real differences. The position of the center points. is changed by a small amount each time the agent makes an observation. The other variables are changed at the beginning of each episode. We vary the maximum turning rate between 20 and 40 degrees, the strength coefficient with +/- 40 %-points and add a turning bias in the range of +/- 3 degrees.

3.3.4. Experiments with knowledge transfer

In the following experiments we use two versions of the default agent. The first agent (without DR) is a trained for 8 million steps, where it seems to be close to its maximal capacity. To get the second agent (with DR) we trained the first agent for an additional 2 million steps, this time on the randomization environment.

After training we tested how do our agents perform on the (simulated) randomization environment as this could be an indicator of the agent's real-world performance.

Finally, we tested both agent in the real world environment. Since the model car gets slower with time, we took measurements for one minute with one agent then switched to the other agent. We measured only finished laps, and we did not stop the agent unless it hit the walls of the track.

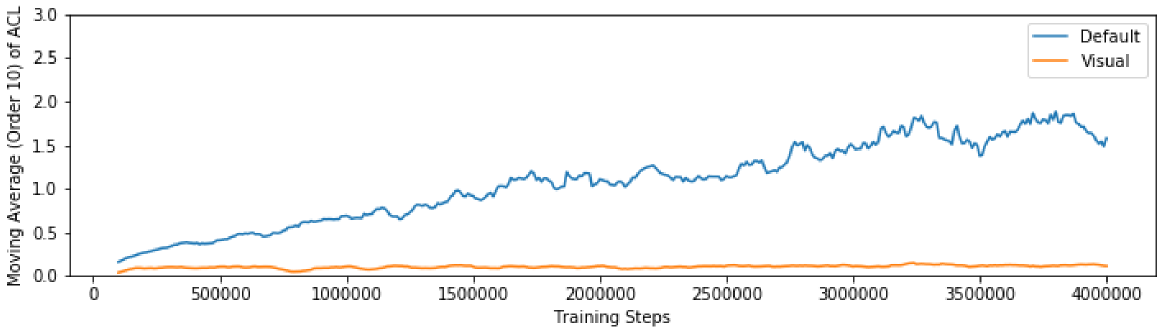
4. Results

For an agent the maximum achievable consecutive laps are 3.5 in the normal environment. In the randomization environment the maximum achievable consecutive laps are higher, 4.8. Both are limited by the 60 second time constraint, after which the agent resets automatically. In the real world environment, we did not have this constraint, hence the number of consecutive laps could go up a lot higher – in the if the agent does not run into the wall, it can run until the battery runs out.

The descriptive statistics, and a link to the data for all the trained agents are shown at Appendix D. The charts were generated using a Python script (see Appendix A).

4.1.Experiments with visual input

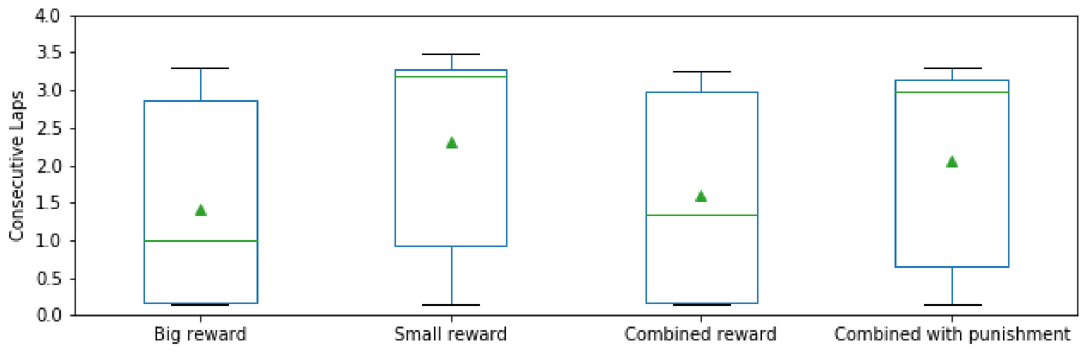
Figure 4.
Comparison of training with visual and vector inputs. The Average Consecutive Laps measurements were taken at every 10,000 training steps and smoothed with a simple moving average of order 10.



The agent receiving only visual input does not learn anything during 4,000,000 training steps. After training, it completes 0.1 laps on average, before connecting with the walls of the track. In comparison the default agent completes 2.4 laps on average after the same amount of training.

4.2. Experiments with different reward structures

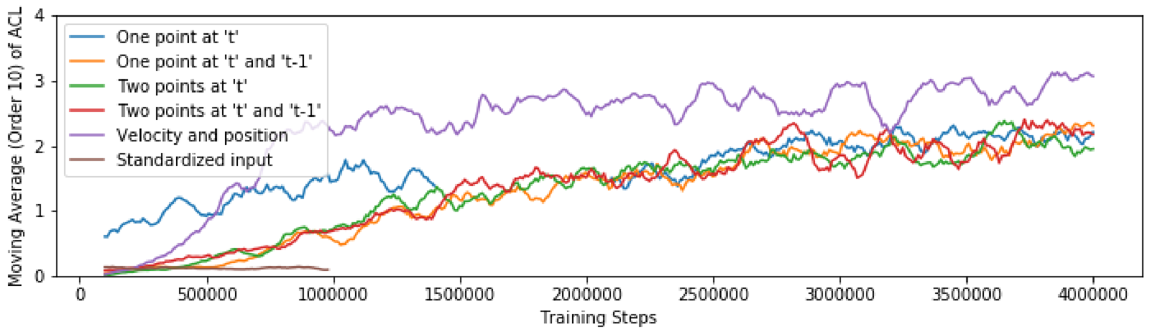
Figure 5.
Boxplot of the Average Consecutive Laps of different reward structures after training for 4,000,000 steps. The green line shows the medians, the triangles the means.



The agents in the “Big reward only” and “Combined reward” condition performs similarly poorly. While the agents in “Combined reward with punishment” and the “Small reward” conditions perform considerably better. This disparity becomes more prominent when inspecting the medians instead of the means. However, the minimum and maximum performance of all agents are similar.

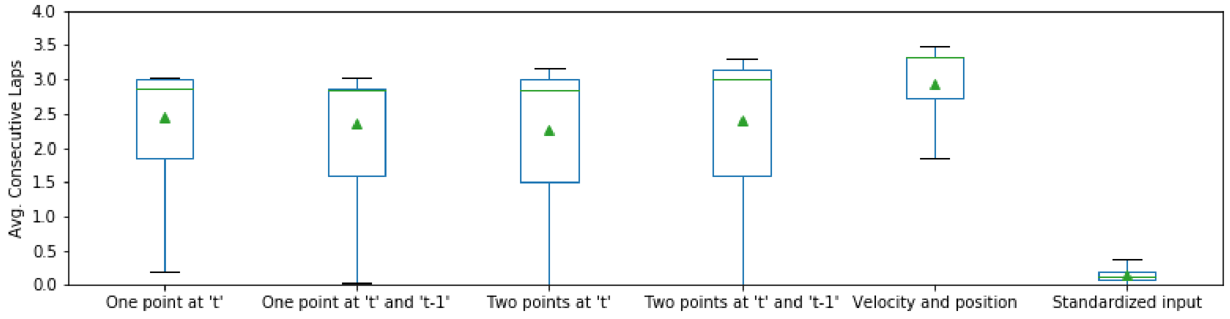
4.3. Experiments with different inputs

Figure 6.
Training with different inputs. The Average Consecutive Laps measurements were taken at every 10,000 training steps and smoothed with a simple moving average of order 10.



The agents in the “Standardized input” condition shows no learning. The agent in the “One point at ‘t’ time steps” starts out learning fast, but after that it converges with the agents in the “One point, two time steps”, “Two points, single time step” and “Two points, two time steps” conditions. The agent in the “Velocity and position” condition learns faster than the others (except one point at ‘t’ in the first 500,000 steps). It also shows better performance throughout the whole training process.

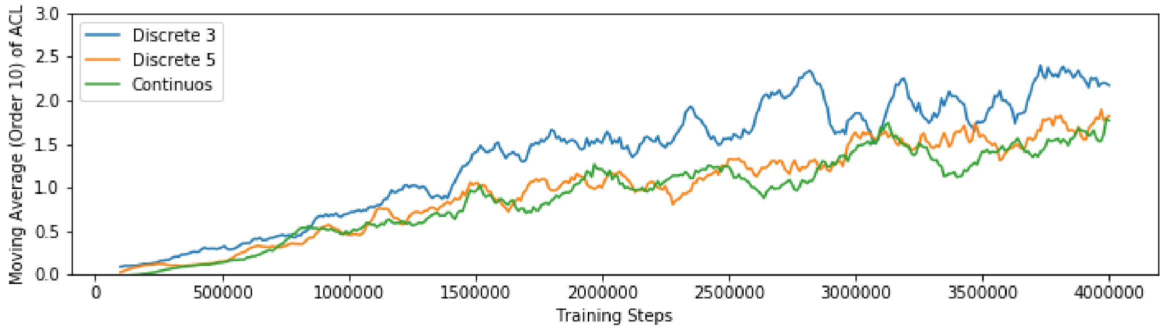
Figure 7.
Boxplot of the Average Consecutive Laps of different input structures after training. The green line shows the medians, the triangles the means.



Except of the agents in the “Standardized input” and “Velocity and position” all agents perform similarly. The agent in the “Velocity and position” condition reaches the maximum possible number of laps (3.5) and has low variance. The agent in the “Standardized input” condition shows constant low performance.

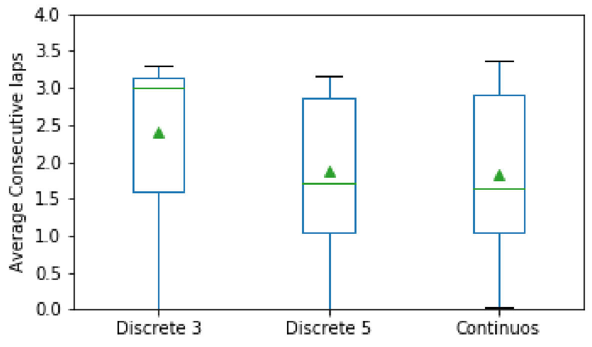
4.4.Experiments with different outputs

Figure 8.
Training with different outputs. The Average Consecutive Laps measurements were taken at every 10,000 training steps and smoothed with a simple moving average of order 10.



The agent in the “Discrete 3” condition trains faster than the other two agents. Since the blue line representing the agent in the “Discrete 3” condition is almost always above the lines representing the agents in the “Discrete 5” and “Continuous” conditions. The performance of the other two agents during training is similar.

Figure 9.
Boxplot of the Average Consecutive Laps of different input structures after training. The green line shows the medians, the triangles the means.

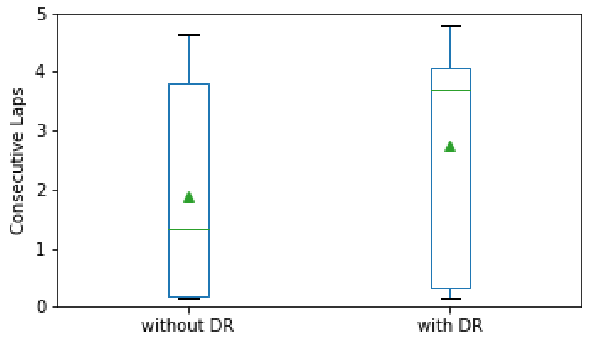


After training, the agent in the "Discrete" condition still performs better than the other two agents which are both performing at a similar result.

4.5.Experiments with knowledge transfer

4.5.1. Test of DR within the simulation

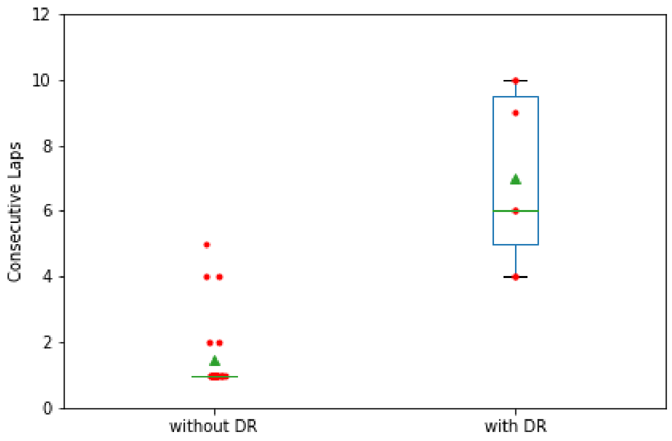
Figure 10.
Boxplot of the Consecutive Laps of the agent on the domain randomization environment before and after domain randomization. The green line shows the medians, the triangles the means.



The maximum number of laps in the randomization environment is higher (4.8) than in the normal environment (3.8). This is because the average speed of the car is higher in the randomization environment. The agent utilizing DR (M = 2.74, SD = 1.8) compared to the agent without utilizing DR (M = 1.89, SD = 1.7) demonstrates significantly better performance, t(198) = 4.46, p = 0.001 on the simulated (randomization) environment.. However, the agent without DR is still able to complete 1.89 laps on average.

4.5.2. Test of DR in the real world

Figure 11.
Boxplot (with scatterplot superimposed) of the Consecutive Laps in the real world environment. The green line shows the medians, the triangles the means, the red dots show the underlying data.



The performance of the agent utilizing DR (M = 7, SD = 2.7) compared to the agent without utilizing DR (M = 1.46, SD = 1.1) demonstrates significantly better performance, t(31) = 8.507, p = 0.001 on the real world environment.. The agent without DR most of the time (21 out of 26) only drove one consecutive lap. As opposed to the agent with DR, which always drove for at least 4 consecutive laps.

5. Discussion

In this thesis our aim was to devise and test an affordable paradigm to explore simulation to real world knowledge transfer. We created a top down car simulation in Unity, then utilized the ML-Agents Toolkit to train different agents with the PPO algorithm. We measured the cumulative rewards during both training and inference. Cumulative rewards are a good basis for comparing agents, since the agents are trained to maximize reward. However, cumulative rewards are not intuitive to understand. Fortunately, cumulative reward can be converted to consecutive laps — the number of laps the agent drove on the track before resetting — by a simple division.

5.1.Can we train the agent with visual input?

The agents can be divided into two categories based on their input: visual and vector agents. Agents with vector input generally learn quicker, but in the real-world visual input is easier to access. In our paradigm, we received only visual data from the real world model. We created a visual agent first.

When learning from visual input, the agent should deduct the position of the car from the image, while it also learns how to control the car. To avoid this problem, Andrychowicz and colleagues (2020) trained a neural network using supervised learning to match the visual data to vector data. Then they used the vector data for the reinforcement learning. Since our task is simpler, we experimented with combining the two steps. We fed the agent a 50 x 28 pixel grayscale image and let it train for 4 million steps. The model did not improve significantly during the training. We also experimented with different setups with similar results. For example, by using RGB image instead of grayscale, and by increasing the contrast between the car and the environment. From these results, we can conclude that **within our limitations (time and computer performance) it is unfeasible to train a car in the simulation with only visual input**. However, there are four options that could be interesting to explore:

1. We could train for a longer period. Although during a 4 million steps the agent did not show improvement, it is possible, that it needs more time to train.

2. Another possibility for improvement would be lowering the image resolution. This could result in faster training, but details (like the exact position of the agent) will be lost when compressing the image.
3. It is also an option to use consecutive images instead of a single image. For vector input the ML-learning toolkit provided a simple solution for this problem, with a parameter called "Stacked Vectors". When set to x (x is between 1 and 50) the agent receives the input not only for the current time step, but also for x-1 previous time steps. This is not an option for image inputs, therefore we implemented a script which did the same for visual input. Our script however slowed down the training process almost to a complete halt, leaving investigations in this direction unfeasible.
4. Finally, tuning the hyperparameters of the neural network (adding a recurrent layer, changing the visual encoder, learning rate, batch size etc...) could result in faster and more effective learning. This tuning however would cost a large amount of time, without the certainty of success.

Because of the previously mentioned difficulties, we stopped working with visual agent, and concentrated on vector agents.

5.2.Can we train the agent with vector input?

5.2.1. How does the agent perform in the simulated environment?

The default agent receives two coordinates of the car as input. It receives a big reward for reaching each checkpoint, a small reward each time step when it drives in the right direction, and punishment when connects with the wall. The agent is resets each time it hits the walls of the track, or in every 60 seconds.

After 4 million steps of training, the agent was able to drive around the track. The agent was able to drive for on average 2.4 laps, and in some cases it drove 3.3 laps – which is close the theoretical maximum 3.5 laps. With the success of the default agent, we could continue our experiments with vector input.

5.2.2. How can we make the agent better?

We constructed three experiments to find out how can we make our agent better. Each experiment focused on a different aspect of the agent: (1) What reward should the agent receive? (2) What should be the inputs of the agents? (3) What should be the outputs of the agents?

Experiment with reward structures

In this experiment we have four different input conditions: (1) “Big reward only”: the agent receives a big reward when reaching a checkpoint; (2) “Small reward only”: the agent receives a small reward for each step in the right direction; (3) “Combined reward”: the combination of the previous two conditions; and (4) “Combined reward with punishment”: in addition to the "Combined reward" condition, the agent gets punished when hits the walls of the track. This serves the purpose of teaching the agent what to avoid.

The “Big reward only” is a sparse reward condition. Agent with sparse rewards are expected to learn slower, since they have to do more random exploration before getting a reward. The advantage of the “Big reward only” condition is that the agent is incentivized to find the shortest route around the track. The agent wants to get to the next checkpoint as quick as possible, and since the agent cannot control the speed, that would mean the shortest way possible.

In the other hand the “Small reward only” condition gives out frequent reward, which reduces the amount of necessary exploration. The agent receives a small amount of reward when it goes in the right direction. The problem of this condition is how we defined "in the right direction. We gave the reward based on how much closer the agent got to the center point of the next checkpoint, which means that the ideal path for the agent is the shortest path connecting the centerpoints of the checkpoints.

"Combined reward" was meant to combine the positives of the two previous methods: the agent should be able to find the shortest path in order to gain the big rewards as fast as possible, but would need less exploration, because of the small rewards.

The "Big reward only" condition performed poorly, which is not surprising as it is supposed to be learning slower than the other agents. The "Small reward only" condition achieved considerably better result. The performance of "Combined reward only" is only marginally better than the performance of Big reward only". Adding punishments worked as expected, achieved close to the "Small rewards only" condition. In the future we suggest trying combining Small rewards with punishments as it might be better than all the experimental condition.

Experiment with Agent inputs

In this experiment we have six different input conditions: (1) "Velocity and position": the agent receives velocity in x and y directions and position data; (2) "One point, single time step": the agent receives coordinates of one point of the car; (3) "One point, two time steps": the agent receives the coordinates of one point of the car for the current and previous times steps; (4) "Two points, single time step": the agent receives the coordinates of two points of the car; (5) "Two points, two time steps" (Default agent): the agent receives the coordinates of two points of the car for the current and previous time steps; and (6) "Normalization": inputs were modified to fit inside the -1 — +1 range. It is advised to normalize the inputs for better performance (Unity ML-Agents Toolkit, 2020).

This experiment was based on the idea of what the agent should be able to deduct. When the agent knows the position and velocity of the car the agent has every necessary information to control the car. If the agent has one point of the car as input, the agent knows only the position of the car. If the agent receives two points of the car it can calculate the car's direction. And if the agent receives information from two different time steps it can calculate the velocity of the car. In the last two cases the agent will need more time to train, because it has to learn how to interpret the input.

The agent with "Velocity and position" input is expected to be the best both in learning speed and performance, because it has direct access to all necessary information. "One point at single time step" is expected to learn fast, but stop before reaching the maximum possible performance, since it cannot deduce direction and velocity. The agents in the "One point, two time steps" and "Two points, single time step" conditions should perform better. The agents in "Two points, two time steps" and "Normalization" condition both should be close in performance to the "Velocity and position" agent, with "Normalization" performing better than "Two points, two time steps". The last four agents should all learn slower than the "Velocity and position" agent.

Our predictions were wrong at two points: (1) "One point, single time step" did not stop learning early and (2) "Normalization" did not learn anything. For the bad performance of the "Normalization" agent we have no explanation. It might be because during normalization the coordinates lost from their precision, but this hypothesis requires further investigation. A possible explanation for the good performance of "One point, single time step" agent is if the task is too simple, and the position of the car is enough information for the agent to drive.

Experiment with Agent outputs

In this experiment we have three different input conditions: (1) "Discrete 3" (Default agent): the agent can drive left, straight or right; (2) Discrete 5: the agent can drive left, slightly left, straight, slightly right or right; (3) "Continuous": the agent has precise control over the turn angle

The goal of this experiment is to examine the tradeoff between precision and model complexity. "Discrete 3" has three output, therefore it is easy for the agent to learn what can it expect from each output. "Discrete 5" provides a more precise control, but should be harder to learn. The "Continuous" provides precise control. However, it has to use the continuous action space, which is handled differently than the discrete action space. ML-agent utilizes the Actor-critic architecture when working with continuous action spaces which might add more complexity to the training (Lanham, 2019).

In the results we saw that "Discrete 3" learns and performs better than the other two. This result shows us that the controlling the car by using only the maximum turn angles (and the option of going straight) is sufficient enough.

5.3.Can we use the agent in the real world environment?

The real world environment consists of an RC car in a racetrack, and a camera looking down at the track. In order to do inference in the real word environment, we need to be able to control the RC car from Unity, and receive the coordinates of the real world car. For the first task, we connected an Arduino microcontroller to the remote. The microcontroller was then able to take care of the communication with unity through a serial port. Translating the camera's output into coordinates was done by a custom Python script. The coordinates of the car were then sent to Unity through a dedicated socket.

Domain randomization was used by Tobin and colleagues (2017) to aid domain transfer. We intend to use the same technique in our research. The goal of DR is to make the agent more robust, by increasing the variance of the input. When training with DR small random numbers are added to the position of the centerpoints of the car and to the strength coefficient, the maximum turning rate was varied between 20°and 40°, and a bias was added to the car's control (+/-3°).

In the following experiments we answer these three questions:

1. Is an agent trained without DR able to drive in the real world environment?
2. Is an agent trained utilizing DR able to drive in the real world environment?
3. Is training with DR better for knowledge transfer than training without

Experiments with randomization in the simulation and in the real world environment

In this two experiments we have two agents: (1) "Without DR": This agent was trained on the usual environment, but for a longer period, thus it is close to its peak performance; (2) "With DR": This agent created by training the "Without DR" agent in the randomization environment for an additional 2 million steps. At this point the "With DR" agent was also close to peak performance.

This experiment is a check how well our agents can perform in the randomized environment. This can provide some data how sensitive is the "Without DR" agent to changes in the environment. The "With DR" agent outperformed the "Without DR" agent. This is not surprising, since it was trained on the randomization environment. However, the "Without DR" agent is in average completed 1.89 laps which means it was still able to drive. This result shows that the "Without DR" agent is robust enough to function in the randomization environment.

Next we used the same two agents, to drive the RC car in the real world environment. The results mirrors what we learned in the previous experiment. The "With DR" agent performed better than the "Without DR" agent. This confirms the results of Andrychowicz et al (2020), stating that domain randomization in a simulated environment can facilitate sim to real knowledge transfer. The performance of the "Without DR" agent also ties back to our previous result. Since the agent is able to complete a lap (sometimes more than one lap), the performance in the randomized environment proved to be a good predictor for the real world behavior.

To conclude, the agent using DR was able to drive the RC car in the real world environment. The agent without learning on the DR environment was less successful, but it was still able to complete a lap (or in some cases multiple laps.

5.4.Strengths and Limitations

Unity ML-Agents toolkit proposes itself as a potential research platform ((Juliani et al., 2018). Our research provided insight into the workings of ML-Agents, therefore we can address this claim.

On the positive side, it is easy to use, use of ML-Agents has a flat learning curve. After following their example materials, we were able to construct our own training environment for a car simulation. For us, the best aspect of ML-Agents is its tight integration with the rest of Unity, and the inputs, outputs and rewards could all be set from Unity. ML-Agents comes with its own sets of training algorithms, but it also can be used as a gym wrapper (Unity ML-Agents Toolkit, 2020), this case the user can utilize their own algorithms for training.

Using the standard training algorithms are easy, however this comes at the price of hiding complexity from the user. The implementation of the training algorithms is not explained in depth in their documentation, and the training scripts are highly complex. for this reason, we could not get the agent (neural network) out from Unity, to use it in our own script (see Appendix E)

Our aim was to create an experimental paradigm to demonstrate simulation to real world knowledge transfer. Our paradigm proved to be economical and easy to use. There are two main limitations with the RC car: (1) The battery drains quickly, and when it happens the speed decreases; (2) The car recalibrates each time when connecting to the computer or when the remote controller is switched on — each time different bias and/or speed settings. Another limitation comes from the communication between Unity and the PVPS, which is a sensitive process. When we tried to do extra calculations — for example calculations needed for counting laps for the real world car — Unity could not process the information coming from the PVPS on time.

6. Conclusion

The aim of this research was to replicate the study of Andrychowicz et al. (2020) using a more affordable research paradigm. In their research the trained an agent in a simulation, and used the same agent after DR to do the same task in the real world. We utilized a car simulator, and a real world RC car to demonstrate the same concept. We divided the task into three sections, each with its own research question:

First we tested whether we can train the agent with visual input. Unfortunately, we were not able to train successfully with visual input, mostly due to computational limitations.

Next we trained the agent with vector input. We created a default agent to measure the performance in the simulation environment. This agent was able to complete multiple laps, on average 2.4 but in some cases it got near the theoretical maximum of 3.5 laps. After making measurements with the default agent, we focused on how to improve the performance. Our findings showed two possible points of improvement: (1) Combining a small reward for each step in the right direction with punishment for hitting the track's wall could improve performance. (2) The agent could use only one point of the car as input, to make training faster.

Finally, we examined whether the agents can perform in the real world environment. Here we investigated how well an agent can function in the real world with and without training on the DR environment. Both agents were able to go around the track, but the agent in the DR condition performed better. Our results showed that training on the DR environment improves real world performance, thus confirming the results of Andrychowicz and colleagues.

Our paradigm can be used to illustrate the usefulness of DR in simulation to real world knowledge transfer. For further experiments however, working with visual input, or combining visual and vector input. In our study this was limited by both computer performance and the current capabilities of ML-Agents. Our paradigm could be improved by rewriting PVPS in a compiled language, enhancing its capabilities (detecting the walls of the track) and integrating it into the Unity environment.

7. References

Aguero Quinteros, J. R. (2019). Reinforcement Learning: Q-learning in Third Person Shooter Games.

Ali, N. M., Rashid, N. K. A. M., & Mustafah, Y. M. (2013). Performance comparison between RGB and HSV color segmentations for road signs detection. *Applied Mechanics and Materials*, *393*(1), 550-555.

Andrychowicz, O. M., Baker, B., Chociej, M., Jozefowicz, R., McGrew, B., Pachocki, J., ... & Schneider, J. (2020). Learning dexterous in-hand manipulation. *The International Journal of Robotics Research, 39*(1), 3-20.

Brown, N., Sandholm, T., & Machine, S. (2017, August). Libratus: The Superhuman AI for No-Limit Poker. In *IJCAI* (pp. 5226-5228).

Burda, Y., Edwards, H., Pathak, D., Storkey, A., Darrell, T., & Efros, A. A. (2018). Large-scale study of curiosity-driven learning. *arXiv preprint arXiv:1808.04355*.

François-Lavet, V., Henderson, P., Islam, R., Bellemare, M. G., & Pineau, J. (2018). An introduction to deep reinforcement learning. *arXiv preprint arXiv:1811.12560*.

Hintjens, P. (2013). *ZeroMQ: messaging for many applications*. " O'Reilly Media, Inc.".

Jain, M. S., & Lindsey, J. (2018). Semiparametric reinforcement learning.

Juliani, A., Berges, V. P., Vckay, E., Gao, Y., Henry, H., Mattar, M., & Lange, D. (2018). Unity: A general platform for intelligent agents. *arXiv preprint arXiv:1809.02627.*

Lanham, M. (2019). Hands-On Deep Learning for Games. In *Hands-on Deep Learning for Games* (pp. 201–204). Packt Publishing.

Lillelund, C. B. (2018). *Transferring Deep Reinforcement Learning from a Game Engine Simulation for Robots* (Doctoral dissertation, Master's thesis, Aalborg University CPH, Clille13@ student. aau. dk, 5 2018).

Mehta, B., Diaz, M., Golemo, F., Pal, C. J., & Paull, L. (2020, May). Active domain randomization. In *Conference on Robot Learning* (pp. 1162-1176).

Min, K., Kim, H., & Huh, K. (2019). Deep Distributional Reinforcement Learning Based High-Level Driving Policy Determination. *IEEE Transactions on Intelligent Vehicles*, *4*(3), 416-424.

Moravčík, M., Schmid, M., Burch, N., Lisý, V., Morrill, D., Bard, N., Davis, T., Waugh, K., Johanson, M. & Bowling, M. (2017). Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, *356*(6337), 508-513.

Nakamura, M., & Yamakawa, H. (2016, October). A game-engine-based learning environment framework for artificial general intelligence. In *International Conference on Neural Information Processing* (pp. 351-356). Springer, Cham.

Nguyen, H., & La, H. (2019, February). Review of deep reinforcement learning for robot manipulation. In *2019 Third IEEE International Conference on Robotic Computing (IRC)* (pp. 590-595). IEEE.

Pinto, L., Davidson, J., Sukthankar, R., & Gupta, A. (2017). Robust adversarial reinforcement learning. *arXiv preprint arXiv:1703.02702*.

Rusu, A. A., Vecerik, M., Rothörl, T., Heess, N., Pascanu, R., & Hadsell, R. (2016). Sim-to-real robot learning from pixels with progressive nets. *arXiv preprint arXiv:1610.04286.*

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347.*

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., ... & Chen, Y. (2017). Mastering the game of go without human knowledge. *nature*, *550*(7676), 354-359.

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

Tobin, J., Fong, R., Ray, A., Schneider, J., Zaremba, W., & Abbeel, P. (2017, September). *Domain randomization for transferring deep neural networks from simulation to the real world.* In 2017 IEEE/RSJ international conference on intelligent robots and systems (IROS) (pp. 23-30). IEEE.

Tremblay, J., Prakash, A., Acuna, D., Brophy, M., Jampani, V., Anil, C., ... & Birchfield, S. (2018). Training deep networks with synthetic data: Bridging the reality gap by domain randomization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops* (pp. 969-977).

Unity ML-Agents Toolkit. (2020). GitHub. Retrieved June 23, 2020, from https://github.com/Unity-Technologies/ML-Agents/blob/0.14.1/

Zhang, F., Leitner, J., Milford, M., & Corke, P. (2017). Sim-to-real transfer of visuo-motor policies for reaching in clutter: Domain randomization and adaptation with modular networks. *world*, *7*(8).

Zhuang, F., Qi, Z., Duan, K., Xi, D., Zhu, Y., Zhu, H., ... & He, Q. (2019). A comprehensive survey on transfer learning. *arXiv preprint arXiv:1911.02685*.

Appendices and Supplementary Materials

Appendix A — Code repository and used packages

In the following table, we summarize the used software and custom scripts. We note when other authors' code was used both here, and directly in the code. Standard libraries for Unity, ML-Agents and Python (like time, pickle, glob, random) are not mentioned separately. Used, well known packages are mentioned, but not cited. The source code for the project is uploaded to: github.com/varon95/CSAI_2020_Thesis

Table 1.

Used software
Unity (2019.3.0f6)
ML-Agents Toolkit (0.14.1) (Juliani et al., 2018)
Tensorflow (1.13.1)
Jupyter notebook
Arduino IDE

Table 2.

Custom scripts	
Arduino Code	CSAI_2020_Thesis/Arduino_code/VolkswagenControl.ino
Function	To communicate between Unity and the RC car
Language	C/C++
Used sources	Controlling the digital potentiometer (Arduino, 2018)
PVPS	CSAI_2020_Thesis/car_radio/carRadio.py
Function	To derive the coordinates of the real world car, and forward them to Unity
Language	Python
Used sources:	Color based object tracking (Rosebrock, 2020)
	Color calibrator for HSV colorspace (Maitra, 2018)
	Socket communication server in python (ØMQ, 2020)
Used libraries	numpy
	OpenCV (cv2)
	imutils (Rosebrock, 2019)
	pySerial (serial) (Liechti, 2017)
	zmq (Hintjens, 2013)
Chart generator	CSAI_2020_Thesis/charts/Plots.ipynb
Function	To generate charts and analysis for the Results section
Used libraries	pandas
	matplotlib
	numpy
Unity scripts	CSAI_2020_Thesis/Unity_Scripts
Language	C#
Agent	CSAI_2020_Thesis/Unity_Scripts/AgentScript.cs
Function	To manage agent behavior
Used sources	Unity3D-Python-Communication (Sornsoontorn, 2019)
Used libraries	NetMQ (Hintjens, 2013)
Communicator	CSAI_2020_Thesis/Unity_Scripts/RadioRequester.cs
Function	To recieve data from the PVPS
Used sources	Publisher / Subscriber protocoll (NetMQ, 2020)
Used libraries	NetMQ (Hintjens, 2013)
Virtual car controller	CSAI_2020_Thesis/Unity_Scripts/CarCont.cs
Function	To control the virtual car
Used sources	Tutorial for car simulation (AxiomaticUncertainty, 2018)
Checkpoints	CSAI_2020_Thesis/Unity_Scripts/checkpoints.cs
Function	Too check when the car makes contact with a checkpoint
Used sources	Checkpoint system (Nomibuilder, 2015)

References for software, packages, in code references

Arduino. (2018, May 17). *Arduino — DigitalPotControl*. Retrieved June 23, 2020, from https://www.arduino.cc/en/Tutorial/DigitalPotControl?from=Tutorial.SPIDigitalPot

AxiomaticUncertainty. (2018, November 26). *Unity Car Physics Tutorial — #1* [Video]. YouTube. https://www.youtube.com/watch?v=8xdXJtu6nig

Hintjens, P. (2013). *ZeroMQ: messaging for many applications*. " O'Reilly Media, Inc.".

Juliani, A., Berges, V. P., Vckay, E., Gao, Y., Henry, H., Mattar, M., & Lange, D. (2018). Unity: A general platform for intelligent agents. *arXiv preprint arXiv:1809.02627*.

Liechti, C. (2017). *Pyserial* (3.4) [This module encapsulates the access for the serial port.]. Retrieved June 23, 2020, from https://pyserial.readthedocs.io

Maitra, S. K. (2018, June 26). *OpenCV baby steps 4: Building a HSV calibrator*. Pi of Things. Retrieved June 23, 2020, from https://piofthings.net/blog/opencv-baby-steps-4-building-a-hsv-calibrator

NetMQ. (n.d.). *Pub/Sub*. Retrieved June 23, 2020, from https://netmq.readthedocs.io/en/latest/pub-sub/#subscriber

Nomibuilder. (2015, April 8). C# version of CheckPoint and Laps system [Comment on the article "Add Checkpoints and Laps Unity Car Tutorial"]. *Unity*. Retrieved June 23, 2020, from https://answers.unity.com/questions/290652/add-checkpoints-and-laps-unity-car-tutorial.html

ØMQ. (n.d.). *Weather update server in Python*. ØMQ — The Guide. Retrieved June 23, 2020, from http://zguide.zeromq.org/py:wuserver

Rosebrock, A. (2019). *imutils* (0.5.3) [A series of convenience functions to make basic image processing functions easier with OpenCV.]. Retrieved June 23, 2020, from https://pypi.org

Rosebrock, A. (2020, April 18). Ball Tracking with OpenCV. Retrieved June 23, 2020, from https://www.pyimagesearch.com/2015/09/14/ball-tracking-with-opencv/

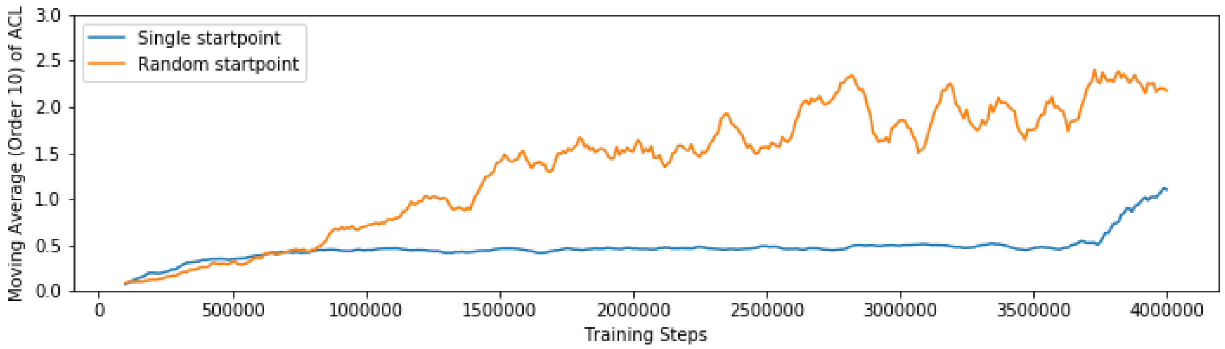
Sornsoontorn, C. (2019). *Unity3D-Python-Communication* [Github repository]. Retrieved June 23, 2020, from https:// https://github.com/off99555/Unity3D-Python-Communication

Appendix B — Why randomize resetting

Resetting the agent to the same starting point exposes the agent to certain parts of the track more. Therefore, we expected the agent to memorize path. While, if we train the agent with resets to random parts of the track, the agent should be able to generalize better.

Figure 12.

Comparison of the agents in the "Single start point" and "Random start point" during learning



The agent in the single starting point condition learned faster in the beginning, but hit a plateau after only 1 million steps, did not learn anything in the next 2.5 million steps and then started to learn again. It is possible that the agent learned that "turning right is good" since this is a good strategy for the first checkpoints but then it was unable to adapt to the changes in the track. If the starting point was somewhere else, the agent might have learnt a more complex pattern before getting stuck, or even learn go around the track, but it is certain to explore less than the randomly initialized agent.

Appendix C — Default parameters of PPO in ML-Agents

The explanation of the hyperparameters can be fould under the “Training with Proximal Policy Optimization” chapter (*Unity ML-Agents Toolkit*, 2020).

Table 3.

Hyperparameters

executable (with extension)	car_with_AI_test.exe
trainer:	ppo
batch_size:	1,024.00
beta:	0.01
buffer_size:	10,240.00
epsilon:	0.20
hidden_units:	128.00
lambd:	0.95
learning_rate:	0.00
learning_rate_schedule:	linear
max_steps:	4,000,000.00
memory_size:	128.00
normalize:	false
num_epoch:	3.00
num_layers:	2.00
time_horizon:	64.00
sequence_length:	64.00
summary_freq:	10,000.00
use_recurrent:	false
vis_encode_type:	simple
reward_signals:	
extrinsic:	
strength:	1.00
gamma:	0.99

Appendix D — Descriptive statistics

All the data used for the tables below can be found at: github.com/varon95/CSAI_2020_Thesis/tree/master/charts

Table 4.
Experiments with visual input

	count	mean	std	min	0.3	0.5	0.8	max
Visual	391	0.1	0	0	0.1	0.1	0.1	0.2
Default	285	2.4	0.9	0	1.6	3	3.2	3.3

Table 5.
Experiments with different reward structures

	count	mean	std	min	0.3	0.5	0.8	max
Big reward	223	1.41	1.2	0.1	0.2	1	2.9	3.3
Small reward	223	2.32	1.3	0.1	0.9	3.2	3.3	3.5
Combined reward	223	1.6	1.2	0.1	0.2	1.3	3	3.3
Combined with punishment	223	2.05	1.2	0.1	0.7	3	3.2	3.3

Table 6.
Experiments with different inputs

	count	mean	std	min	0.3	0.5	0.8	max
One point at 't'	347	2.46	0.8	0	1.9	2.9	3	3
One point at 't' and 't-1'	355	2.36	0.8	0	1.6	2.9	2.9	3
Two points at 't'	362	2.26	0.9	-0.1	1.5	2.9	3	3.2
Two points at 't' and 't-1'	356	2.41	0.9	0	1.6	3	3.2	3.3
Velocity and position	350	2.93	0.7	0	2.7	3.3	3.3	3.5
Standardized input	500	0.15	0.1	-0	0.1	0.1	0.2	0.8

Table 7.
Experiments with different outputs

	count	mean	std	min	0.3	0.5	0.8	max
Discrete 3	356	2.41	0.9	0	1.6	3	3.2	3.3
Discrete 5	383	1.87	1	0	1	1.7	2.9	3.2
Continuous	378	1.82	1	0	1	1.7	2.9	3.4

Table 8.
Test of DR within the simulation

	count	mean	std	min	0.3	0.5	0.8	max
without DR	100	1.89	1.7	0.1	0.2	1.3	3.8	4.7
with DR	100	2.74	1.8	0.1	0.3	3.7	4.1	4.8

Table 9.
Test of DR in the real world

	count	mean	std	min	0.3	0.5	0.8	max
without DR	26	1.46	1.1	1	1	1	1	5
with DR	7	7	2.7	4	5	6	9.5	10

Appendix E — Using an agent outside of Unity

For our use-case, the greatest problem was the format of the ML-Agents compatible neural networks. These networks can be one of two formats: (1) .nn, which is native to ML-Agents, and cannot be exported to other format with existing tools; and (2) .ONNX — Open Neural Network Exchange format (https://onnx.ai/). This can be used separately from unity, for example in TensorFlow. Doing inference in TensorFlow using an ONNX model would have been easy to directly integrate with the PVPS. This way, there would have been no need to use Unity for inference, making the whole process simpler. Unfortunately, when ML-Agents gave the agent extra inputs during training, and what these inputs were, and how to reproduce them remained hidden from the user. This means, that we could not gave the same input to the agent as the ML-Agents and therefore we could not do inference outside of Unity.