Freie Universität Berlin

Masterarbeit am Institut für Informatik der Freien Universität Berlin

Smart Driving Agent based on Deep Reinforcement Learning

Robert Zifrid
Matrikelnummer: 4764352
robertzifrid@zedat.fu-berlin.de

Betreuer: Ricardo Carrillo Mendoza
Eingereicht bei: Prof. Dr. Raúl Rojas
Zweitgutachter: Prof. Dr. Daniel Göhring

Berlin, 14.November 2019

Abstract

In this thesis a deep reinforcement learning agent is trained in an environment made with the Unity game engine. The new ML-Agents API allows the simulation to communicate with a Python backend, which enables research with familiar tools. The simulation features a huge variety of different adjustable parameters and can be run in parallel. The agent successfully learned to follow the track and is robust to various environmental changes. Furthermore, the future deployment of Neural Networks on the AutoMiny car is prepared. The AutoMiny cars are developed by the FU Berlin (Institute of Computer Science). The model car features the Nvidia Jetson Nano, which allows the hardware accelerated inference via Deep Neural Networks.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

14.November 2019

Robert Zifrid

Contents

1 Introduction

Autonomous Driving is one of the major challenges of the 21st century. Various tech companies, politics, and international research institutes work at full blast on solutions towards this challenge. The main target is to make personal transport more comfortable, more efficient and safer.

The Freie Universität Berlin does a lot of research on this challenge as well. The institute of computer science developed a high-tech model car for autonomous driving over the years. This project provides a basis for new approaches and opportunities for the research on autonomous driving.

1.1 Motivation

While public transport is widely digitalized and partly even fully automated, the solutions for personal transport are very sparse due to the more complicated setting. A lot of car producers integrate numerous sensors and intelligent driving assistant software, but all of them still require a human driver behind the wheel. The need for human supervision limits the use of these systems.

The goal of Autonomous Driving is to fully replace the human driver with intelligent software. This kind of software uses various methods from AI that have recently become more popular due to improvements in processing power, memory, infrastructure and research.

Especially Reinforcement Learning methods make it possible to train an agent without concrete predefined behavior or patterns. The idea is to train their behavior with a special reward system, which indirectly controls the behavior of the agent.

To enable students to gain hands-on experience in this field of research, the institute developed small model cars and an artificial training environment. While many different algorithms can be easily implemented, some experiments require setups that are hardly manageable.

That's the reason why digital simulation environments are an appealing solution. These digital environments are always available, cheap and almost entirely customizable. Many scenarios can be more easily be implemented in a simulation, which allows the user to focus on conceptual issues. The use of a modern game engine for creating the simulation, allows building plausible real-time environments.

1.2 Goal of this work

The main goal of this work is to develop a digital simulation environment and to make it available for further research on the driving agents. This simulation should come up with new graphical opportunities and be created with a common game engine. A special focus should be set on the adjustability of the environment to emphasize the advantages over the lab environment.

Additionally, a reinforcement learning agent should be trained in that implemented environment.

1.3 Methodology and Workflow

The development of the simulation is done with the Unity game engine. Unity offers a specific API, which makes it possible to connect the simulation environment with a Python Backend. Therefore, an implementation of a Reinforcement Learning Agent in Python is also part of this work.

Development of the simulation has gone hand in hand with the machine learning Python end of the project. The main idea is to establish a complete machine learning workflow as soon as possible. This helps identifying bottlenecks and problems early on.

1.4 Structure

This work is divided into nine main chapters that describe the overall approach to solving the problem. The following list describes the content and the intent of each chapter.

Introduction
This chapter gives an introduction into the topic of this project. Here we define the main problem as well as the motivation to work on this topic. Furthermore, there is a short overview on the main structure of this work.

Fundamentals
This chapter is used to clarify the theoretical fundamentals used for solving the problem. It starts by explaining the major concepts of Machine Learning, dives deeper into the topic of Reinforcement Learning and ends with a description of the Unity game engine.

State of the Art
In this chapter we describe the state-of-the-art techniques and solutions of Reinforcement Learning and the concrete car model based on related work and previous projects.

Method and Solution
This chapter describes the approach on solving the main problem. It gives an overall overview of the created system, further information on the simulation as well as a description of the deployment process.

Implementation
This chapter gives a description of the development environment concerning the used software, modules and hardware. Additionally, here we describe and define the main parameters of the simulation and deployment.

Results and Interpretation
In this chapter we describe the observations made during various simulation episodes and try to justify them. Additionally, we visualize the outputs of the convolutional layer and try to interpret the systems view of the input data.

Conclusion
This chapter contains an overall conclusion of the investigated approaches and their results. It summarizes the entire work on this project.

Outlook
In this chapter we refer to promising optimizations and possible simulation extensions.

Appendix

2 Fundamentals

In recent years big companies such as Google, NVIDIA and Facebook focus a lot of their research on AI. There is a variety of applications in daily life for machine learning solutions. Google for example uses convolutional Neural Networks to automatically label street addresses of Google Street View recordings [8]. Usually these systems learn via a given set of input examples and corresponding output labels. Computationally intensive methods are used to train Neural Networks to associate these inputs with their associated label. This kind of machine learning with known input and output pairs is called supervised learning. Advancements in this area caused the current interest in AI-research. The great interest lead to further improvements and sophisticated methods. All of this provides a generic algorithm for solving many different problems. This allows us to tackle problems that were previously very hard to solve, if at all possible. In practice designing and fine tuning these Neural Networks is still a challenging task and subject to ongoing research. There is no known universal way to solve arbitrary problems. There are many different incarnations of Neural Networks, which specialize in solving different tasks. Once the right architecture is found, it's still hard to train it, as this involves tweaking a lot of hyper-parameters, albeit practical experience and solid understanding of the underlying concepts.

Opposed to supervised learning, there is also unsupervised learning (and semi supervised learning), which is actually encountered in practice since most data is unlabeled. As already hinted at, unsupervised learning does not have access to input and output pairs like in the supervised setting, instead it is reliant on only some form of input.

Unsupervised learning is also heavily featured in this thesis in the form of deep reinforcement learning. Because the goal is to train the agent to follow the lane in this thesis via reinforcement learning, one can imagine that there is no absolute correct steering point at any given moment. When driving a car or riding a bike, the steering is quite forgiving, meaning that slight differences in steering wheel position would also be acceptable. Only longer periods of off-steering will require major corrective adjustments. Reinforcement Learning is discussed in more detail in the section Reinforcement Learning or in [21] [23].

2.1 Machine Learning and Deep Learning

Artificial Intelligence is a subfield of computer science that originally could be characterized as the goal of automating tasks usually performed by humans [4]. When the media is headlining huge advances in AI, it is usually referring to advances made in machine learning or especially in deep learning.

Machine learning aims at recognizing patterns in data on its own. This is a different approach to symbolic AI and expert systems, where rules have to be entered manually. Before Machine Learning rose to its current glory, symbolic AI dominated the field.

Deep learning is a subfield of machine learning and especially of Neural Networks. It evolves over different techniques for training Deep Neural Networks. We refer to networks as deep, if the network has more than one hidden layer (layers between the input and the output layer). Accordingly, networks with one hidden layer are referred to as shallow networks. By stacking multiple layers, we hope to achieve different representations of the data (possibly hierarchical representations), thus making a certain task, such as classification, easier. For more details refer to these sources [7] [4]

2.2 Reinforcement Learning

Reinforcement Learning follows a common scheme. It deals with an agent acting within an environment at its very core. The agent bases its actions on the observations it makes in the environment. The mapping of observation and actions is performed by the policy. After taking an action, the environment usually changes its state and provides a reward. In nature this reward is more so something the agent (the animal brain) perceives, rather than something given by the environment [21]. The reward can be positive or negative (punishment) and serves as a feedback signal for the agent to evaluate its actions. The reward doesn't have to be a direct consequence of the last action, but is usually given for reaching a goal. Achieving that goal is in most cases the result of a sequence of actions. This indirect feedback causes the attribution problem, which describes the difficulty of judging a given action in a given state as good because of this indirection.

Finding an appropriate Reward function that results in the agent learning the desired behavior is very difficult [21]. There are some more key concepts which are required for better understanding of this field.

Reward and discounted Reward
RL algorithms often use the discounted rewards instead of the raw rewards. The idea behind this is to control how greedy the agent should behave in regards to the rewards. (Should immediate rewards be favored over long term rewards?). The parameter that controls this behavior is usually denoted $y \in [0, 1)$ (often $y \in [0.9, 0.999]$). Future rewards are weighted by this gamma essentially based on the formula $G_t = \sum_{k=0}^{\infty} y^k \cdot R_{t+k+1}$, where G_t denotes the discounted reward at timestep t and R_i denotes the reward received at timestep i [21, p. 55]. Notice that later rewards are weighted with a higher superscript by y, which means that these later received rewards have less of an influence (they get discounted) as $y < 1$. If y is smaller (closer to zero such as 0.9 e.g.) rewards in the future get reduced and current rewards are favored over long term rewards.

(State-)Action Value (Q) function
The Action-Value function ($Q(a|s)$) assigns every action an estimated value that describes how good the action a is, given being in state s. (Deep) Q-learning basically aims at creating an accurate estimate of this function and then just chooses the best action at each point in time.
Instead of learning the Q function for every action at every state, it turns out to be better to have the Q function broken down in to $Q(a|s) = V(s) + Adv(a, s)$ since you don't need to learn the value function state for every action[23, p.72].

State Value function
The Value function $V(s)$ (depending on the state) was proposed, which estimates the average (cumulative) reward from this state onward. It basically measures how good the given state is. Generally, we prefer to be in good states. If choosing an action that leads to being in a good state, this action should be favored over actions that lead to ending up in worse states.

Advantage function
The Advantage function describes how much better or worse an action is as opposed to others (the action with average action). By rearranging the definition of the Q function, the advantage function can be defined as $Adv = Q - V$ (see above). Notice that in PPO we defined the advantage as $Adv = R - V$, with R being the discounted reward. So instead of the Q value for that action in that state, we rolled out the episode and what the discounted reward will be for taking that action. Therefore, this is an estimate of the Q function to some extent.

On vs Off-policy
RL algorithms can be categorized into On-policy and Off-policy algorithms. On-policy algorithms use the same policy to evaluate their action as they do to choose an action. On the contrary, Off policy algorithms use a different policy to collect data (experiences) than the policy they try to optimize. Typically, Off-policy algorithms use a replay memory to choose their action. The memory gathered in this was produced by a different policy (maybe just an older version but still, this is considered a different policy).[23] [21]

Model based vs non model based
Model based methods have a model of the environment, which they can use for planning by querying the model to see the results of taking certain actions and to observe how the environment behaves [21, p.7]. According methods without such a model are called non model based methods.

2.3 Artificial Neural Networks

The variety of different network architectures allows applying Neural Networks to many different applications. The artificial neurons are often grouped into layers. Many of the structures found in deep learning are briefly presented in this section. For more details please refer to this source[22].

2.3.1 Layer and Architectures

There are various types of layers found with specific properties and functions. A certain combination of different layers defines the architecture of a Neural Network Model. In the following listing you will find the most important layer types with a short description. Notice that, as deep learning is an active area of research, new architectures and improvements are frequently proposed.

Convolutional Layer
Convolutional layers make use of the locality principle and thereby reduce the parameter count. Applied to images (2D Convolutional layers), it is assumed that close by pixels are highly correlated. Similarly, a pixel in the top left corner is barely related to a pixel in the opposite corner of the picture. In low level layers, the convolutional kernels typically learn to detect low level features such as edges or color blobs. The intermediate activations are feature maps that are basically images themselves. Therefore, convolutional layers can be stacked intuitively. The different activations within a layer can be viewed as different channels. Successive layers operate on the activations of the previous layer. This hierarchical feature structure can detect more complicated features, such as corners. Visualization of deep convolutional networks hints at detection of complex concepts, such as eyes and many more. As the convolution operation moves the kernel matrix (filter) over the input image, a translation invariance is achieved, which often represents a useful feature in computer vision tasks. An image of a cat is still an image of cat, even if the cat is located at a different position within the image. Different convolution types (1D,2D,3D) are usually used for different input types [4].

Pooling Layer
Pooling layers reduce the input via the according operation. A MaxPooling layer for example usually regards a 2x2 pixel sized window and relays the maximum of it. This window is then moved with stride of 2 (typically avoiding overlapping). This reduces the width and height by half. Notice that this is done on each feature map. The operation also introduces small translation invariance [7](2d maxpooling). The operation, window size and stride size are hyperparameters.

Dense Layer
The Dense layer represents the most basic structure in Neural Networks. They receive vectors as inputs. In convolutional architectures they are associated with being the reasoning part, whereas the convolutional part is responsible for detecting useful features. As every input is connected to every neuron in this layer the number of trainable parameters increases quickly with more inputs.

Flatten and Reshape Layer
The Flatten layer is typically used as a bridge between the 2D data of a convolution operation and the first dense layer. Without the Flatten layer the two layers would be incompatible. The Flatten layer achieves this by reshaping the 2D activation maps to a linear vector. The Reshape layer servers the same purpose of adjusting mismatching shapes of activation volumes.

Dropout Layer
Dropout layers randomly set a fraction of neuron outputs to zero during training. Whether this happens to a neuron is chosen randomly with a probability p (the dropout rate). The layer does not have an effect during inference. This is a common regularization technique. The network should rely on a small subset of neurons. If neurons are randomly missing (their output is set to 0 regardless of the input), the network can no longer rely on these specific neurons. This way overly big weights should be avoided.

Residual Connections
Can be understood as shortcuts within a neural network. They build an alternative data flow in order to re-inject low level features into higher level layers. This is helpful, as consecutive layers not only rely on the previous layer, but have access to lower level features. Also, this fights the vanishing gradients problem, as the gradient can reach the lower level layers additionally through the residual connection [4].

Addition and Concatenation Layer
Addition layers are often found in residual and inception type architectures, where they serve the purpose of recombining the different processing streams. Concatenation layers serve a similar purpose. They come in handy when combing visual data and vector information. The Flatten layer of a convolutional part of a network or some of the following dense layers could be concatenated with some vector information. One could imagine that the network is fed additional information in this manner. The additional information (speed, imu or calibration data such as camera angle etc.) could help the network learn and deal with different settings (varying camera angles etc.).

2.3.2 Activation Function

Several different activation functions were proposed over the years. In the beginning the sigmoid activation function was commonly used, as it has the property of squeezing the output into the range $(0, 1)$. This activation got out of favor, as it suffers from the vanishing gradients problem. As the derivative of the function is almost zero for huge input values, huge errors only lead to small gradients, with diminishing effects in deeper layers. Following proposals aimed at addressing this issue. ReLU is often a good default. Now follows a non-exhaustive list with some hints of useful features of common activation functions.

- Sigmoid squeezes the output into the range of (0,1) which can be interpreted as a percentage or probability. Although it is used in intermediate layers, it might be appropriate in the output layer for a single neuron.
- Softmax function is a commonly used activation function for classification task. The softmax function normalizes the output of multiple layers to sum up to one. Setting the number of output neurons (in a classification scenario) allows interpretation of the output as a probability distribution if trained with one-hot-encoded output labels, suggesting that the labels are disjoint.
- Tanh is similar to sigmoid but its outputs lie in the range $(-1, 1)$. This feature is also useful in control tasks, where outputs like motor controls can go in different directions.
- ReLU is more robust to the vanishing gradients problem and has therefore become one of the default choices. Its output is the identity of if the input is less than 0 and else 0. Some variants try to improve on it, such as leaky ReLU, which also allows for small negative gradients.
- Linear is often chosen for regression tasks, as it is not directly limited to a range.

2.3.3 Loss Function

The Loss function defines the objective that the Neural Networks tries to optimize. This can also be seen as a cost function (for making errors), that the network tries to minimize. There are common Loss functions for certain tasks, but certain problems require a carefully crafted Loss function in order to train a network successfully.

2.3.4 Training

Training a Neural Network is often a challenging task, as a lot of parameter tweaking is required. Nevertheless, this process has recently dramatically improved for supervised learning due to new techniques, which are characteristic to deep learning. Deep Reinforcement Learning introduces hyperparameters, which can be very hard to get right (such as the reward function). One major benefit is that data does not have to be labeled as in supervised learning.

2.4 Data Augmentation

In most cases, Machine Leaning algorithms only have access to a fixed set of data to train on. This training set should contain a wide variety of representative data to train a network that generalizes well to new data from the same distribution as the training data.

The problem with creating such representative data sets is to handle that challenge with limited resources like time, costs or special conditions that could not easy be realized. A commonly used solution to this problem is the concept of Data Augmentation. This strategy takes the existing data and alters it randomly according to different methods that preserve their original class. The resulting augmented data will be included in the training process. The method aims at making the trained model more robust and invariant to certain transformations.

Data augmentation can be well understood in the context of image classification. There are various methods to change the existing image data. Transformations like scaling, shifting, flipping or rotating should lead to a classifier that is invariant, even when these transformations are applied to the images. Furthermore, one can add specific filters like noise, blur or color filter to the original image, which intends simulating of special environmental conditions. A rotated image of a cat remains a cat.

All in all, Data Augmentation is a useful strategy to artificially extend the existing data set. But the choice of concrete methods of Data Augmentation depends on the specific use case. Notice that certain images change their class, if certain transformations are applied. When it comes to object recognition, it is common sense to use position and perspective changing methods like flipping and rotation. But in the case of numeric digit recognition, arbitrary rotation is critical, because the class of the image could change (e.g. by applying a 180 degrees rotation to an image of the number six, it may seem to be a nine).

2.5 Transfer Learning

Once the agent manages to drive in the simulation, the next step would be to deploy the network in the model car. Although different measurements have been taken to let the agent generalize as well as possible, it is not to be expected that the agent will be directly able to drive in the lab. Especially the car dynamics will defer from the one in this simplified simulation. Still, the assumption is that the network can be transfer learned to the new circumstances much more quickly. This is beyond the scope of this thesis. Notice that driving in the lab might be bootstrapped with the already well performing vector field force approach. Imitation Learning algorithms (Behavior Cloning), which should be just mentioned here, are potentially suited for this task.

It is a common scheme in transfer learning to disregard the dense part when adopt-

ing a network to a new but similar task, and to add a new dense part on top of the flattened output of the convolutional part. The network is then trained for a while on the new task, while the weights of the convolutional part are frozen. Ideally the convolutional part already detects universally useful features.

2.6 Video Game Engine

Video game engines are very sophisticated pieces of software. They coordinate multiple subsystems such as rendering, physics, audio, animation, networking and much more in order to support game designers to create games that entertain people all over the world. Different genres rely on certain systems more heavily than others. The spectrum of games ranges wildly. Some games try to stand out by their gameplay, while other focus on aspects such as graphics, realism, storytelling or physics. The branch of games that aims at telling stories via especially plausible environments, often pushes the limits of the technology of the time. A few games feature a very detailed and realistic simulation of the real world, fulfilling the purpose of immersing the players in these worlds. They present what is already possible and, since the game industry is gaining economic power, even more sophisticated portrays of the real world can be expected in the future. Creating these virtual worlds is usually a huge team project, where skills in physics, rendering, software architecture, sound and art (just to name a few) are essential [11]. Players can usually interact with these simulations in real-time, meaning all calculations have to fit at least in the time frame of 30 milliseconds (30 FPS -> 1sec / 30 = 33 milliseconds).

2.6.1 Unity Game Engine

In the past, game engines were often made from scratch for each game. But some studios put a lot of effort into keeping their engine general for their genre. Unity and Unreal engine are engines that basically can power any game. Unity allowed many enthusiasts to enter the world of game development by providing a powerful and yet simplistic engine to realize their ideas. Cross-platform support, ease of use and the great learning material (video tutorials, good documentation) pushed its usage. As tutorials and workshops are organized at the FU for game development, those individuals might be interested to work in future projects with this system.

2.6.2 Physics Subsystem

Unity features a commercial grade physics subsystem for games. This kind of physics engine aims primarily at delivering plausible physics effects for real-time applications. [11] [16] [18] The built-in physics system features rigid body, ragdoll physics and particle effect. For this project it was essential, that the physics engine could simulate at least basic vehicle physics. Unity features a special type of collider that includes many parameters, eliminating the need to manually extend the physics engine.

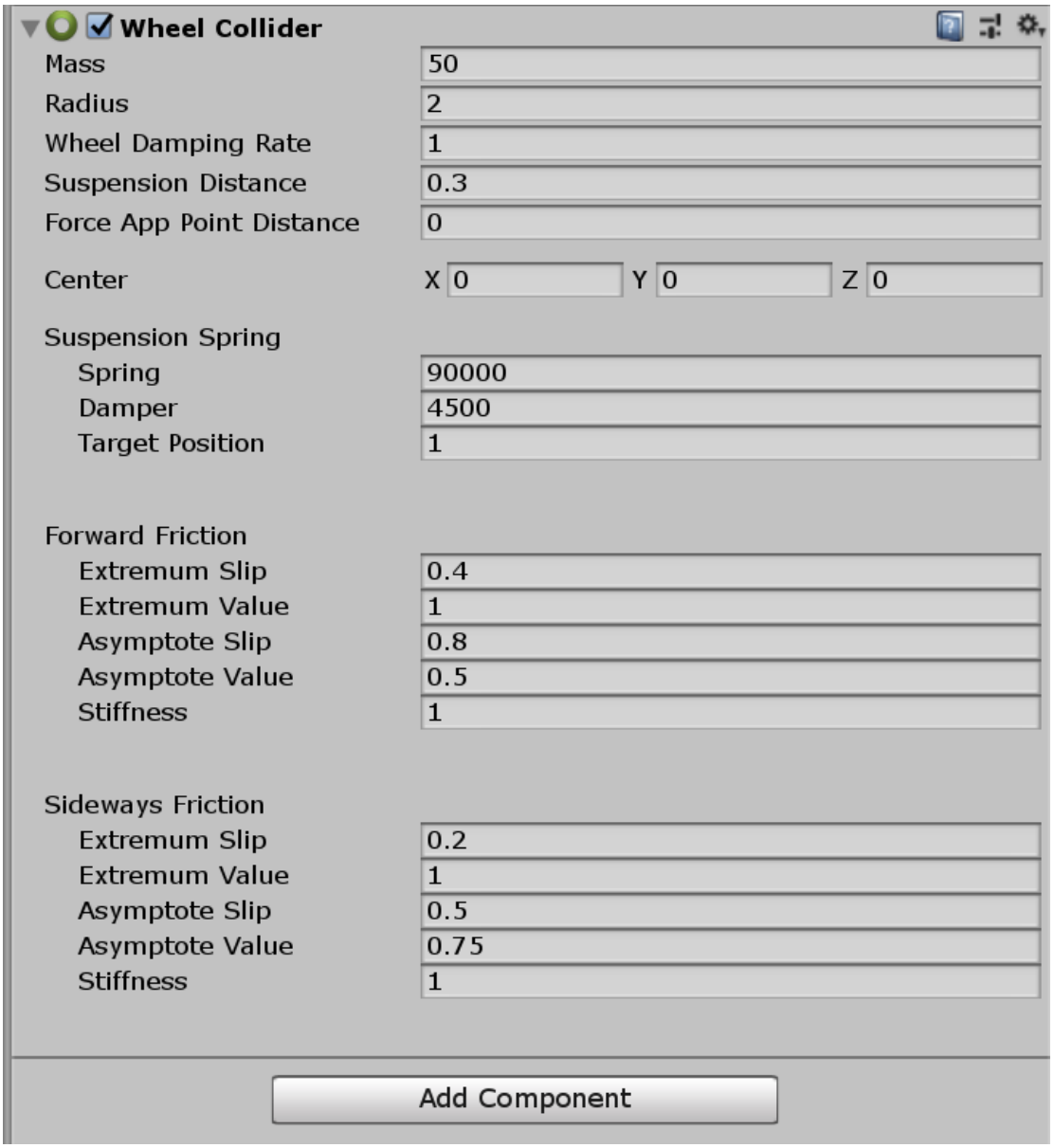


Figure 1: Screenshot of Unity build-in wheel collider parameters

This figure shows the wheel model used in this project which is built into Unity. Major features, such as drift and suspension, are captured by this model, which suffices for the intended purposes of this project. The steering of the wheel was modeled with a simple Ackermann model.

The main elements of the physics engine, like springs and hinges, allow the implementation of some further effects. Unity also announced their plan to feature a new physics engine in the future [15], which will further improve the physics simulation. The different engines should be interchangeable.

2.6.3 Unity ML-Agents

The Unity team created a package for training (reinforcement learning) agents in an environment, that is simulated using the powerful Unity game engine. Especially a Python API is implemented, which allows Python scripts to communicate with the simulation / game. Python support is great for compatibility with the already established machine learning infrastructure. Great documentation teaches the basic usage of this API with step by step instructions on setup and implementation of an example environment. The package already includes some interesting environments with pretrained models.[12]

3 State of the Art

3.1 Deep Reinforcement Learning

Applying Deep Neural Networks to Reinforcement Learning issues spawned the field of Deep Reinforcement Learning. This thesis features the PPO algorithm that was developed by Open AI in 2017 [25]. Open AI also provides environments (called gyms) for testing and developing RL algorithms [1]. More details on the algorithm can be found in the PPO section 4.3.

Keras is a widely used high level framework for training Neural Networks. It depends on libraries that implement Deep learning primitives such as Tensorflow, CNTK and Theano. Recent versions of Tensorflow even integrate Keras, so its accessible without a separate installation.

The Keras-RL project [20] implements some popular RL algorithms such as

- Deep Q Learning (DQN)
- Double DQN
- Deep Deterministic Policy Gradient (DDPG)
- Continuous DQN (CDQN or NAF)
- Cross-Entropy Method (CEM)
- Dueling network DQN (Dueling DQN)
- Deep SARSA

That project does not implement the more recent algorithms: A2C, A3C and PPO, as of the time of writing.

3.2 AutoMiny

The corresponding model car of this project is called AutoMiny. This model was developed at the Freie Universität Berlin and is used for educational and research purposes on the topic of autonomous driving vehicles and robotics. The car (on a scale of 1:10) comes with a variety of sensors. Additionally, this concrete configuration supports calculations on a GPU provided by the NVIDIA Jetson Nano (see figure 2).

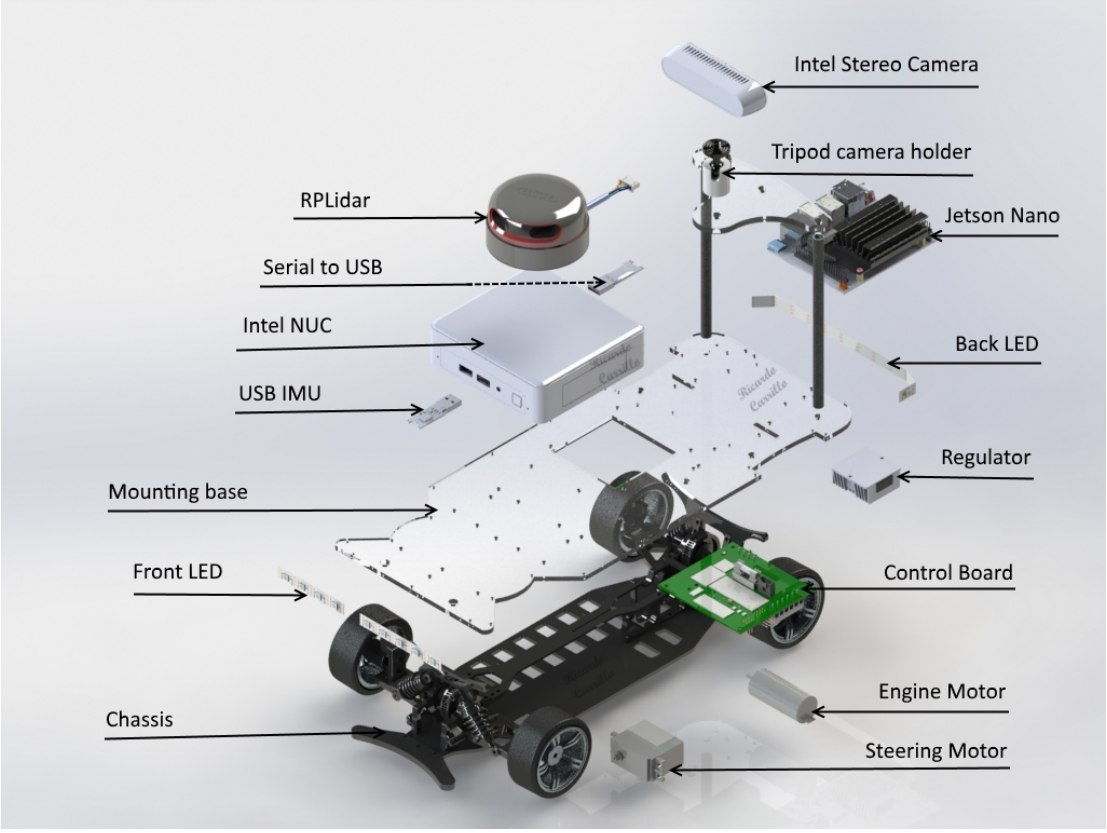


Figure 2: Jetson variant of the AutoMiny car model taken from [3]

The perception sensor system consists of an infrared stereo camera (Intel Real Sense D435), a LIDAR 360 laserscanner (RPLIDAR A2M8 360 one beam) and an inertial measurement unit (BOSCH BNO055 USB Stick).

The model can be controlled via an external interface (smartphone or Xbox controller). Furthermore, it can be programmed via ROS melodic on Ubuntu to drive autonomously.

ROS is a commonly used robotic operating system that provides a set of software libraries and tools. Its main architectural element are nodes. Each node implements some sort of functionality. This flexible system heavily relies on the Observer Design Pattern [6], where topics can be subscribed by each node or made available to other nodes.

3.3 AutoMiny Car Model by Helliaca

This project[13] focused on integrating the ROS framework into Unity. The students created a detailed model of the lab by manually creating the required assets, meaning they had to use a 3d modeling software (blender) to sculpt structures. (They even included the iconic chess pieces found in the lab.(see figure below)) While Unity allows creating basic shapes such as cubes or spheres, these more complex structures need to be modelled manually. Also, textures need to be mapped to the right positions via UV-coordinates. This workflow needs much time and especially expertise with the software tools. Once created, the assets can be easily imported (via drag and drop) into the Unity scene. The section 8.1 describes ideas how in the future this laborious work might be simplified. Several of these assets were reused in this project.

(a) Modeled AutoMiny Car

(b) Modeled Props

4 Method and Solution

4.1 System Overview

The entire system consists of two main components as visualized in the following picture:

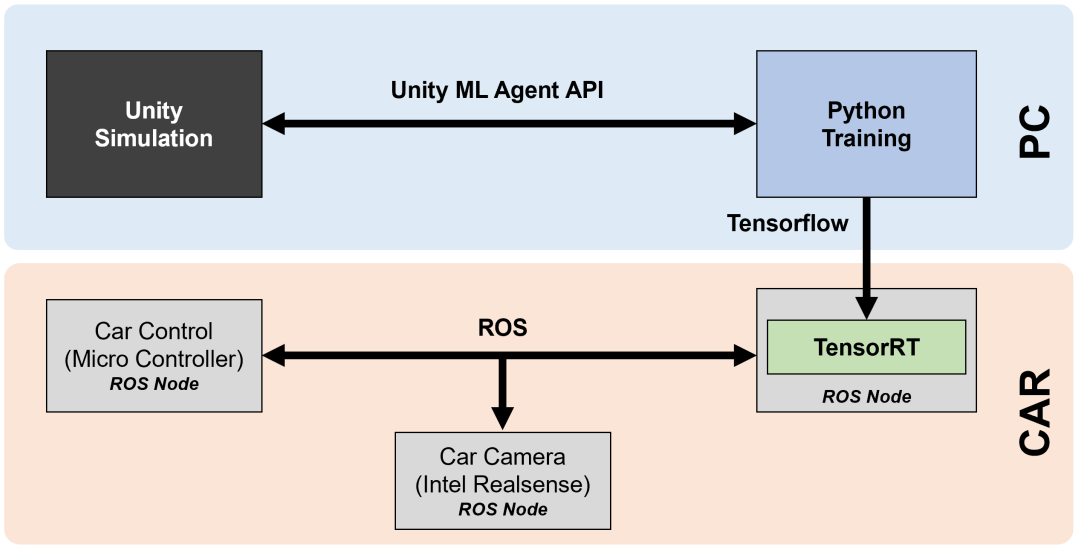


Figure 4: Schematic Overview of the System

The first part is the upper (blue colored) PC component. On the left-hand side, we see the Unity Simulation that communicates with the Python Training Module via the Unity ML Agent API. During training, the Python Backend module and the Unity simulation communicate over the Unity ML Agent API, as observations and actions need to be transmitted. Training Neural Networks will take place inside the Python Backend.

Once training is completed, deployment requires the model to be converted to a frozen Tensorflow graph. This frozen graph can then be optimized with TensorRT for inference.

The second part is the lower (orange colored) CAR component where the main module (ROS-Node) includes the TensorRT inference module. This main Node receives camera frames from the Camera Node (Intel Realsense connected to the intel NUC). The camera frames are analyzed by the TensorRT inference engine that applies the trained network. The main node publishes the according topic to the ROS network. The ROS Node Car Controller performs commands accordingly.

4.2 Architecture

The architecture was initially chosen to be simple to start with. This supports faster iteration in the development of the algorithms. The complexity of the model was planned to gradually increase if the agent would not be able learn the desired behavior. This simpler architecture succeeded at the given task. Generally easier solutions are to be preferred over more complex ones. See Occams razor [7, p. 111].
A simpler network was also favored for different reasons:

- Working with edge devices usually means working with limit resources. To avoid issues during deployment this is the more reasonable approach. Surprisingly, the hardware turned out to be performant. In that case there are resources left, which allow for further tasks to be performed.
- The PPO algorithm has no replay buffer, but it still has to buffer the small rollout of the policy for multiple agents. Additionally, these observations were extended by data augmentation. These batches have a noticeable impact on system memory.
- The simpler architecture decreases the amount of required computing power.

Figure 5 depicts the network architecture (more specifically the output shapes of the layers) of the actor that was successfully trained to stay on track.

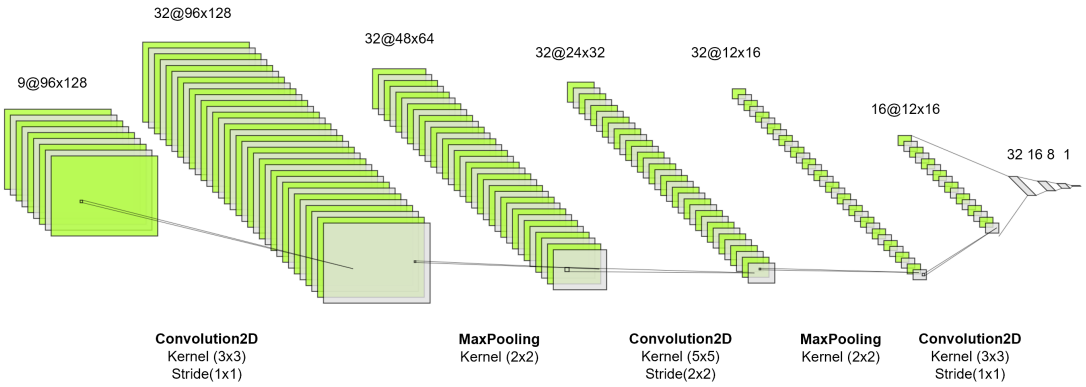


Figure 5: Actor Model Architecture of the successfully trained Agent

The input are the last three images stacked along the color channel. The environment decision interval was set to be 16. This means that observations are recorded at every 16th physics simulation. The simulation uses an adjustable update rate of 50 Hz (0.02 s between to simulation steps). The time between two consecutive images is then also 0.02s*16 = 0.32s. This means also that the agent is queried at rate of 3.125Hz (1/(0.02s*16)) for an action (about 3 times each second).

The intuition behind providing it more than one frame is that agent can learn to approximate speed, and acceleration, which will have an influence on the steering behavior of the car. There are many alternative ways of providing multiple frames to the model.

The following convolutional layers use the ReLU activation function and represent a common structure with the interleaved MaxPooling blocks. The second convolutional layer is special in the sense that is uses slightly bigger kernels and a stride of 2, which additionally down samples the activation maps. The convolutional layer feature L2 kernel regularizers and the dense layers have dropout layers for regularization.

The output layer uses the tanh function to output an action between -1 and 1. As the actor network represents a normal distribution, the output is interpreted as the mean of a normal distribution with a fixed sigma = 0.2. The fixed sigma was accommodated by excluding the entropy term in the loss. Another option would be to have an additional output for that sigma. The speed was not controlled by the agent as it makes achieving the desired behavior more complex and can be handled well by classic methods (control theory). Unfortunately, controlling the speed is out of the scope of this thesis but it is a very interesting subject and can lead to surprising behavior as section Observation indicates.

The critic has a similar architecture as is shown in figure 5. It has a slightly higher parameter count as it lacking one convolutional layer and the output of the Flatten layer therefore becomes larger. No dropout is used in the dense layers. The output is a single neuron with linear activation which is common for regressions tasks (it regresses the value function).

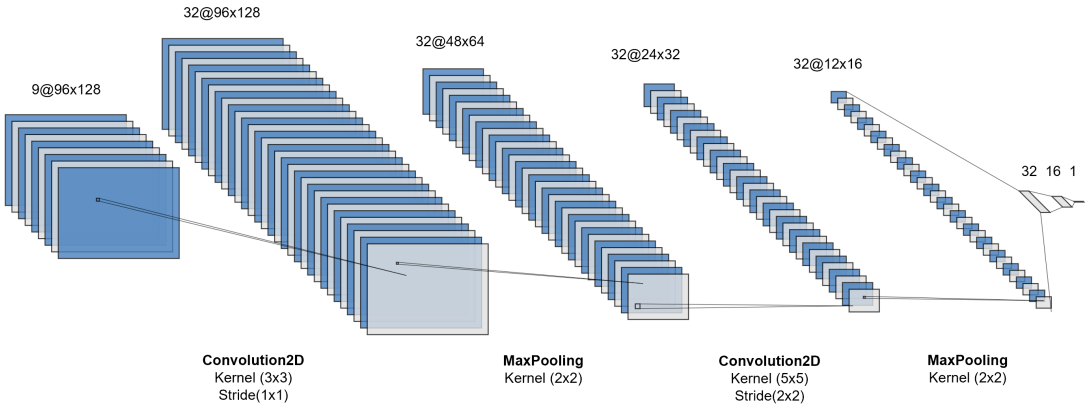


Figure 6: Critic Model Architecture of the successfully trained Agent

More details on the parameters can be found in the accompanying code repository.

At one point many different architectures were implemented, to test if they worked well with the developed framework. That included versions of these architecture types:

- 2D convolution
- 3D convolution
- LSTM
- Residual connections (inception blocks are similar)
- Shared actor and critic model

Because the simulation was not that technically mature, it is insensible to make a statement about their performance. The eventually more complex architectures were tested, but did not achieve any noticeable advantage, which is why they were not further pursued. Future work could very well investigate that.

4.3 Proximity Policy Optimization

The general setup of reinforcement algorithms consists of an agent acting in an environment that provides feedback via rewards. See section Reinforcement Learning in Fundamentals.

Proximity policy optimization is an On-policy reinforcement learning algorithm based on policy gradient optimization. This algorithm is used throughout this thesis. The implementation is based on the implementations from [23][5]. This extended Keras implementation specifically allows the collection of observations from multiple agents in parallel. This fashion of parallelism is similar to the A2C kind of implemented parallelism and corresponds to the suggested implementation of the paper [25].

Furthermore, the PPO algorithm is an extension of the trust region optimization algorithm (TRPO[24]). The main extension is happening in the loss function. Instead of using a constraint to keep the update to the policy small, PPO clips the loss accordingly. This shows improved results and simplifies the implementation [25].

Each iteration of the PPO algorithm begins by collecting T observations for each actor. The algorithm obtains an initial observation when the environment is reset. The observation is fed to our actor policy network. The policy provides an action to run the simulation for a single time increment. (Notice that the policy network typically represents a normal distribution. The action is a sample drawn from the represented probability distribution).

Running the simulation yields a reward, a new state, a done flag and usually an information structure that might include additional information. The old state, reward and action taken are recorded to the current batch of the agent. The new state/observation is then used for choosing the next action for this policy rollout. This inner loop of observing and taking actions is repeated, until our batch has size T or the episode has ended (done flag is true).

The collected rewards are then discounted [25][23]. Doing this incentivizes the agent to prefer actions with immediate rewards over taking actions that yield rewards in the future (although they may be bigger), if y is closer to zero (y is usually in range (0.9,0.999)). In other words, y values closer to 1 make the agent more far-sighted.

Once the batch is complete, the critic network predicts the value (estimation of the expected reward) of each state observation. The advantage (of taking one action over the other) is calculated as discounted reward – value. The advantage is also gathered in the batch.

Once all actors have collected their batch, the batches are merged together. State, ad-

vantage, predicted action and action are used to train the actor network. The critic is then trained on the state and discounted reward afterwards.

It is possible to join the actor and the critic to share, for example the convolution feature extractor. In this case they train on a joined loss.

The loss that PPO is optimizing consists essentially of three losses:

- The first loss is the loss V, which is basically the loss of the critic. This is the well-known mean squared error. The critic tries to predict the reward (discounted), so being in the given state what could we hope for. This is a classical regression task.

- The main loss of the actor is a clipped version of the TRPO algorithm as discussed previously. Basically, the clipped version of policy ratios times advantage should be maximized. Note that advantage values > 0 mean that this action should be more likely, so in turn the ratio needs to get increased. On the other hand, if the advantage value is smaller than zero, the action is considered not significant (not good), so the likelihood should be decreased. Clipping this ratio assures that the policy does not change to drastically. In other words, the updated policy should stay in a (trusted) region around the old policy (since the name TRPO).

- The last loss is an entropy loss that is designed to motivate exploration of the action space.

4.4 Simulation

Simulating an environment offers many benefits and possibilities:

Reduced Hardware Issues
Working with real hardware often comes with many pitfalls (real-world problems) such as missing drivers, incompatible software versions, broken parts and so on. As of the beginning of this project the car did not exist in its final version. Some parts were also not available, which postponed assembly. Due to this, developing started with an older version of the model car that featured the Nvidia Jetson TX1. This was removed and replaced by the Jetson Nano. As the Jetson Nano boots from a micro-SD –card, which includes the OS and all data, so the card can be plugged into another Jetson Nano and still function. Either way, a simulation doesn't suffer from those kinds of hardware issues (only the hardware that runs the system, but that is far less likely to fail than the bleeding edge technology).

Fast Iteration of Ideas
Faster implementation of ideas (e.g. day and night cycle) allows to cover a huge variety of different conditions. In a simulation you can change conditions by editing the corresponding parameter. In this case, where the simulation is made with Unity, the simulation is no longer a black box and can be changed to any desired behavior. For example, creating different lighting conditions in the lab might work by flipping the light switch, but basically only creates different scenes, whereas in Unity one can control the sun changing its position. In this way many different kinds of lighting can be generated. One could also position different light sources. With Unity planning on integrating real time raytracing, much more realistic lighting can be achieved. This affects global illumination, reflections and shadows. Interestingly enough this is also an AI application, because although Nvidia RTX series of graphics card features dedicated hardware for ray tracing, it uses deep learning to denoise a much lower quality image of the scene [17]. But not only lighting can be easily manipulated in the simulation. Controlling other cars in the simulation can be as simple as just setting its transformation (position and rotation) according to a fixed path. It also offers a playground for non-deep-learning-based algorithms. It might also help decide which sensor / information is best to use.

Data Augmentation
Variation is added if the model is parameterized (size of car, track design). Environment parameters can be varied during training such as the angle/ camera position and much more. This yields a very sophisticated data augmentation scheme that should make the solutions more robust to the varied parameters.

Simplified Coordination
Coordinating experiments is easier as the lab is not blocked. Setting up an experiment in the lab can be very time consuming and quite difficult to coordinate, especially if it involves more than one car. Often things need to be put in place, networks need to be reconfigured.

Simulating Malfunctions
The simulation can often offer data that is hard or impossible to get in the real world. For example, segmentation of an image into different categories could be done by using a manual shader (which will be much easier with shader graph tool). Also, it is very difficult to recreate a scenario where hardware fails, like a flat tire, malfunctioning brakes, or a defect sensor in the real world.

Simulating Remote Places
Different places can be modelled, to check whether algorithms are robust to that environment. For example, photogrammetry is a method for high detail reconstruction from multiple views of objects and scenes. Simulating remote places might be expensive or not possible.

Reduced Costs
Costs can be reduced as multiple experiments can run in parallel and be easily repeated.

Unsupervised Learning Support
Simulations are safe for experimentation. They are naturally suited for unsupervised learning, where the agent is expected to fail and make mistake to explore their environment.

Simple Hardware
Off-policy algorithms need large amounts of memory (especially with images). Saving huge amounts of data from real life experiments can be difficult due to limited memory, extra overhead etc.

Building such an exhaustive piece of software can be considered a very ambitious undertaking, especially considering the limited time frame. As there was no prior or little experience working with a game engine, many tools and workflows had to be learned.

The benefits of the long-term value of this work is considered to outweigh the mentioned issues.

4.5 Porting to AutoMiny

Adapting the results to model car means that corresponding ROS nodes have to be implemented, which receive the camera feed, run inference and publish the corresponding topics. For faster inference it is recommended to let TensorRT optimize the network for inference. This process usually involves reducing the floating-point precision from 32 Bit precision to 16 Bit precision and discarding computations that are not necessary for inference. The reduction of floating-point resolution usually results in tolerable accuracy decrease which is traded for huge runtime performance gains.

Additionally, the network inference is hardware accelerated, as the car featuring the Nvidia Jetson Nano is used. Alternatively to the TensorRT approach of freezing the graph, one could also use the Tensorflow library on the Jetson Nano which also includes an integration of Keras (tensorflow.keras).

Implementation of the according nodes, Jetson SD card image and a description of how to convert the model (including a colab notebook with appropriate versions of libraries) will also be made available in the accompanying code repository.[26]

5 Implementation

5.1 General Information

This section is especially interesting for practitioners who want to build upon this project, understand certain design decisions or want to reuse solutions to some of the many challenging parts of this project.

5.2 Development of the Simulation

Naturally a very basic prototype was developed first with fixed streets and props placed in the scene. By utilizing free assets from the Unity asset store a scene can be quickly put together. Once this is done, the ML-Agents API has to be integrated and checked, if the data arrives at the Python end as expected. The Python end should be able to import all planed libraries and make sure they function properly. Lastly, the Python API should be able to send commands back to the Unity environment, which in turn should react correspondingly.

Once this pipeline is established, development of algorithms and features can begin. (Aside from this main goal the deployment of the car was pursued in parallel. Especially the Jetson Nano had to be prepared, which required installing different libraries. This typically is quite a tedious task on such cutting-edge technologies. Although it now integrates nicely with the AutoMiny car, it was prepared for the worst case where it had to do everything on its own (fetching the camera feed from the realsense directly, communicating to the motor driver and running inference).

Development took place on three machines with Nvidia GPUs. A second rig was also used to try different settings (hyperparameters) in parallel. Working with three (the third was a laptop) PCs inevitably led to a managed process for synchronizing the progress across all machines. Code would be managed by version control (git) and resources such as the environment could just be copied to the other system. The initial system features an older version of ML-Agents, but fortunately updating is as easy as changing the according two folders of the sdk.

5.2.1 Integration of Unity ML-Agents

The Unity ML-Agents package integrates naturally into the Unity workflow. Mainly one has to import the package (as easy as clicking a button) and set up following parts:

- Academy
- Area (optional)
- Agent
- Brain (Learning, heuristic or player)

The brain represents the decision-making policy. Currently there are three types: learning, heuristic and player. The API allows controlling an agent manually, which comes in handy during debugging. For simpler environments/behavior (or for comparison) a heuristic brain can be implemented, that bases its decisions on a simple algorithm (e.g. using some heuristic). The last option is the learning brain that abstracts a machine learning algorithm. Simple architectures can be directly trained within Unity, which also offers an inference engine for faster predictions. More importantly, the learning brain is also the port to the Python API. Being able to use the known tools (Python and all its libraries) is an important feature, as already acquired knowledge can be reused and applied to this new technology. This clear interface allows development mostly independent of the simulation implementation, which is essential for reusability. The interface of the Unity simulation is conveniently designed close to the already widespread open AI gym interface [1]. If necessary, a wrapper can easily be implemented for compatibility.

The academy coordinates all agents in a scene by relaying their observations to the according brain. Given an observation (visual, vector or text), the brain infers an action to take and sends it back to the academy, which relays the action back to the agent.

An agent optionally acts within an area. The area is responsible for resetting the environment. This could be something like repositioning targets, changing tracks, etc. A clean implementation simplifies the deployment of multiple agents within the same environment. Training agents in the same environment allows saving resources such as assets and the runtime environment only having to be loaded once.

The agent acts in the environment based on its brain. As described earlier the brain therefore requires an observation and returns an action. The agent then acts on that action by interacting with the environment (or changing its internal state). This usually means applying forces to motors, changing some attribute etc.

5.2.2 Environment Parameters

An important goal of training a model is to make it generalize well. Meaning it should also make right predictions on data which it has never trained on but which is similar. The training data should be representative of the domain. Machine learning practitioners often face the problem of overfitting the machine learning models to the training data, which can be understood as memorizing the training data or some part of it. Apart from achieving good results on the training set, the model might totally fail to make a sensible decision if presented with new examples. See section Data Augmentation in Fundamentals (This goal is similar to making the model robust to certain types of noise and distortions.) By including more variation (parameters) to the environment, we pursuit the goal of making the network robust to all these factors. The network should therefore generalize better. While this helps fighting overfitting, it makes training the network also more involved. One approach is to use Curriculum learning to gradually increase the difficulty (see Curriculum learning). As of the time of writing the parameters of table 1 are implemented.

Table 1: Environment parameters as of time of writing

Environment parameter	**Description**
Start Parameters	
Spawn	specifies the position along the circuit (in range -1,1) where agent should begin the episode
Spawn Rotation Offset	specifies the rotation in regard to the direction of the path the agent is facing when it spawns
Spawn Side Offset	specifies in units how far the agent should be translated to the side relative to the center of the path
Target Lane Track Offset	specifies the lane the agent is supposed to drive on and in which direction
Track Segment Parameters	
Lane Width	specifies the width of one lane
Border Width	specifies the width of the lane border
Lane Texture Noise	specifies how much noise should be applied to the lane texture
Stripe Length	specifies how long the stripes in the middle of the track are
Lane Color	scalar that is used to set the color of the lane. the scalar interpolates the color according to fixed gradient
Ground Color	same as Lane Color but for the surrounding ground
Border Height	specifies how high the border should be (vertical offset of the lane border)
Procedural Generation	
Skyline Perlin Scale	controls the structure of the background buildings
Mountain Perlin Scale	controls the structure of the mountain range
Perlin X, Y	control the structure of the skyline and buildings
Reposition Sun	flag that if set, positions the sun in the upper randomly
Car Parameters	
Car Mass	weight of the car (indirectly influences the steering behavior)
Car Base, Rear Length Car Turn Radius	parameters that the steering behavior
Camera Pitch, Yaw	specify the rotation of the car camera
Zone Parameters	
Border Zone Shift	specifies the shift of Border Zone towards the lane
Border Zone Timer	lets the agent fail the episode if it stays in the Border Zone for the specified amount of time steps

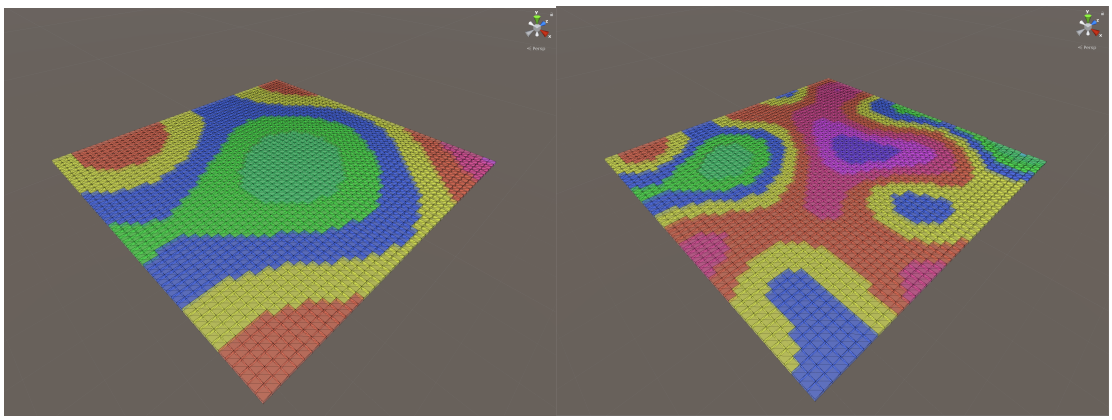
5.2.3 Procedural Generation

Unity features an asset store, where many assets are freely available, while others are for purchase. Once you import an asset, it is usually ready for usage and you can just drag and drop it into your scene. This way you can model a scene quite easily. Although it requires some artistic skill to orchestrate a scene that is plausible and appealing. However, this will usually suffice for a prototype. Depending on the scene, more or less fine tuning will be required. One way of circumventing this problem is using low poly assets (low number of polygons) and simple lighting. Nonetheless, building a visually stunning scene requires a lot of effort. Physically based rendering gives a more intuitive approach, see section Raytracing and GAN for Asset Generation.

In order to provide the network with a variety of data, the scene should be augmented with a variety of different backgrounds and other visuals in order to improve the generalization of the network. Modeling this desired variety by hand is not feasible. This issue is tackled with procedural generation. Instead of placing every object in the world by hand, they are placed via code. Not only can objects be placed via code, but also any attribute can be set through code. Especially mesh data (the vectors, triangles and UV texture coordinates) can be generated and manipulated via code. This technique is heavily used in this project for generating the streets, mountains and city background.

One reoccurring scheme in game development is to use some noise pattern to vary the environment and therefore make it more interesting. Perlin Noise is widely used for this purpose, as it has a continuous structure and it works in octaves. This feature of this fractional noise can be observed in figure [7]. There one can see that the pattern on the left can be found on the right image in the bottom left corner. These patterns are generated using Perlin noise at different scales and the corresponding coordinates. The structure inherently reminds of natural terrain, such as we will see in a moment.

Figure 7: Procedurally generated Perlin noise patterns at different scale parameters



For the city skyline for example some buildings are predefined as Unity prefabs, which allows easy initiation. A n x m matrix according to a noise pattern is sampled. For every number in this grid a corresponding prefab is initiated (the number can be converted to an index accessing the list of possible buildings). As these are symmetrical, they are not rotated, but this could be achieved by generating a similar grid that contains the rotation information or any other information (for example color). This technique can be used in very creative ways. A forest for example could be realized by replacing the prefabs with trees. For a difficult lab setting on the other hand, multiple props (the chess pieces, boxes, etc..) could be scattered within a desired area. Extending the idea of generating additional layers of detail in this way could generate detailed buildings, built out of cleverly combinable blocks.

A variation of this is used for generating mountain-like structures. In this case the height is sampled from the Perlin noise and the normalized height is used to interpolate the color.

Figure 8 shows how the mountain like structure is created by placing the height according to samples of Perlin Noise. The color of the vertices depends on their normalized height.

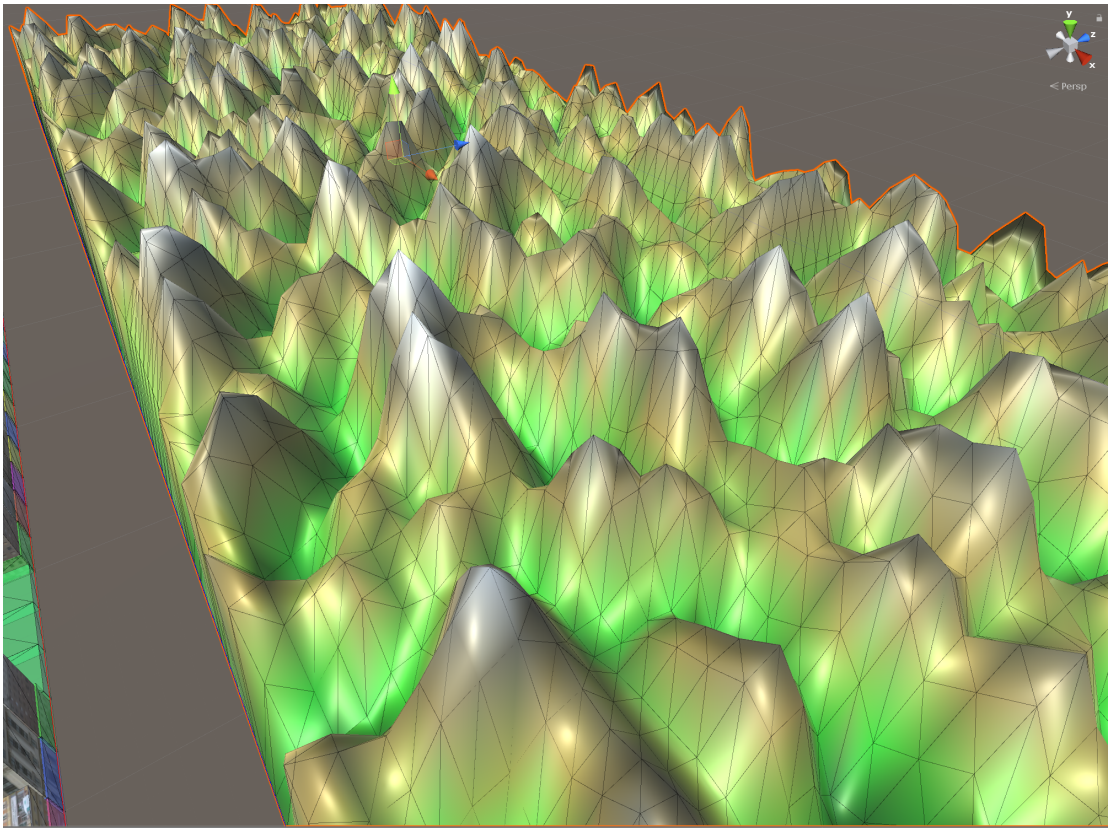


Figure 8: Procedurally generated mountain range from Perlin noise

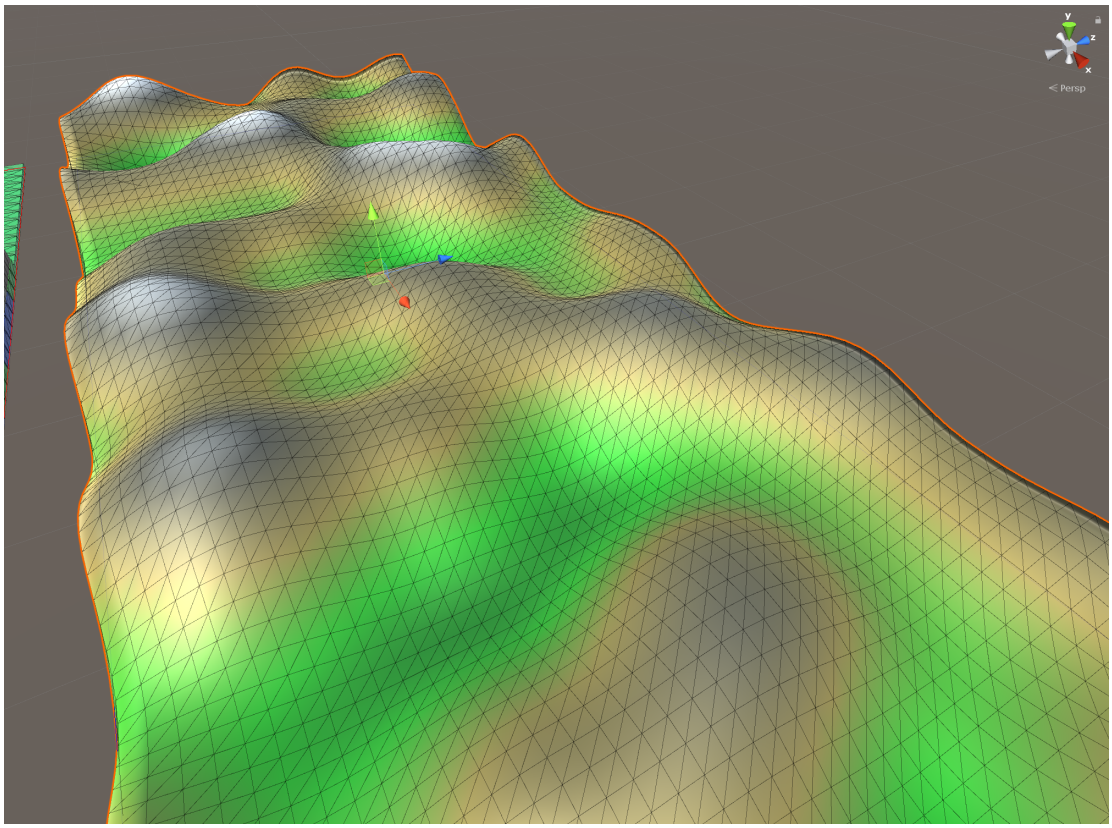


Figure 9: Hill-like structures generated from Perlin noise

In figure 9 we see how a different scale parameter to the Perlin Noise results in a more hill-like structure (although the color range should be adjusted). In this figure the grid placement in the x- and z- axes of the vertices is more clearly visible.

The streets are a hybrid of being placed manually and being created procedurally. Each street segment is essentially controlled by the parameters type and radius/length. The type depicts whether it is a left curve, a straight street or a right curve. For the curved roads see the above figure 10, the radius parameter r specifies the radius of an imagined circle. The middle lane of the curved road would then be the circumference. The angle alpha (alpha = 90° in the figure 10) states how much of the circle segment the road covers.

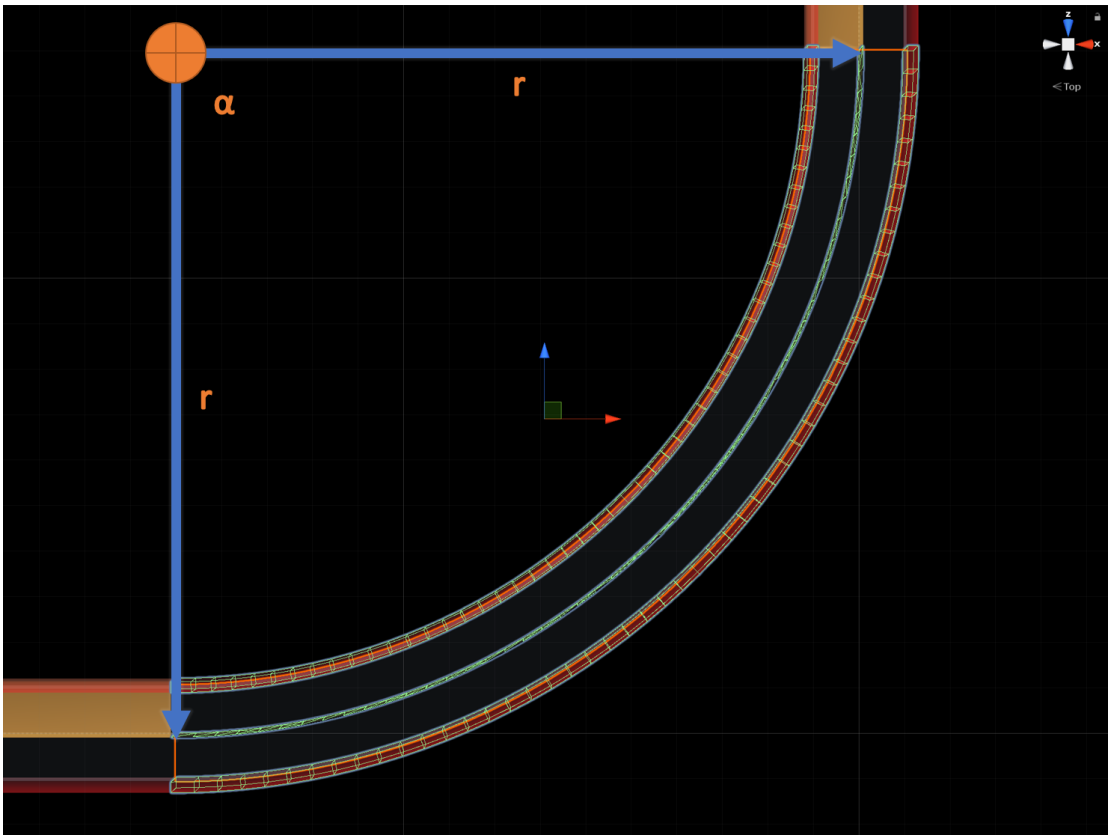


Figure 10: Procedurally generated curved road

For the straight road this parameter states how long the road segment is. The street editor Object allows then to automatically connect the individual segments together according to the desired order. Each segment takes care of calculating necessary geometry (center of the lane, the triangles to the curb and the curb itself). Originally the rewards were mainly given by detecting in which area (zone) the agent was located. Then it could be penalized for driving on the opposite lane or crossing the sideways. This was implemented by adding collision geometry that integrates into the physics system. When collisions appear, certain events are triggered where handling functions can be set up to deal with the situation accordingly. A hurdle was that this requires convex geometry. Convex shapes are also more efficiently tested for collisions. Therefore, the street calculations are broken down into an adjustable number of steps, which allows control of the fidelity of the geometry.

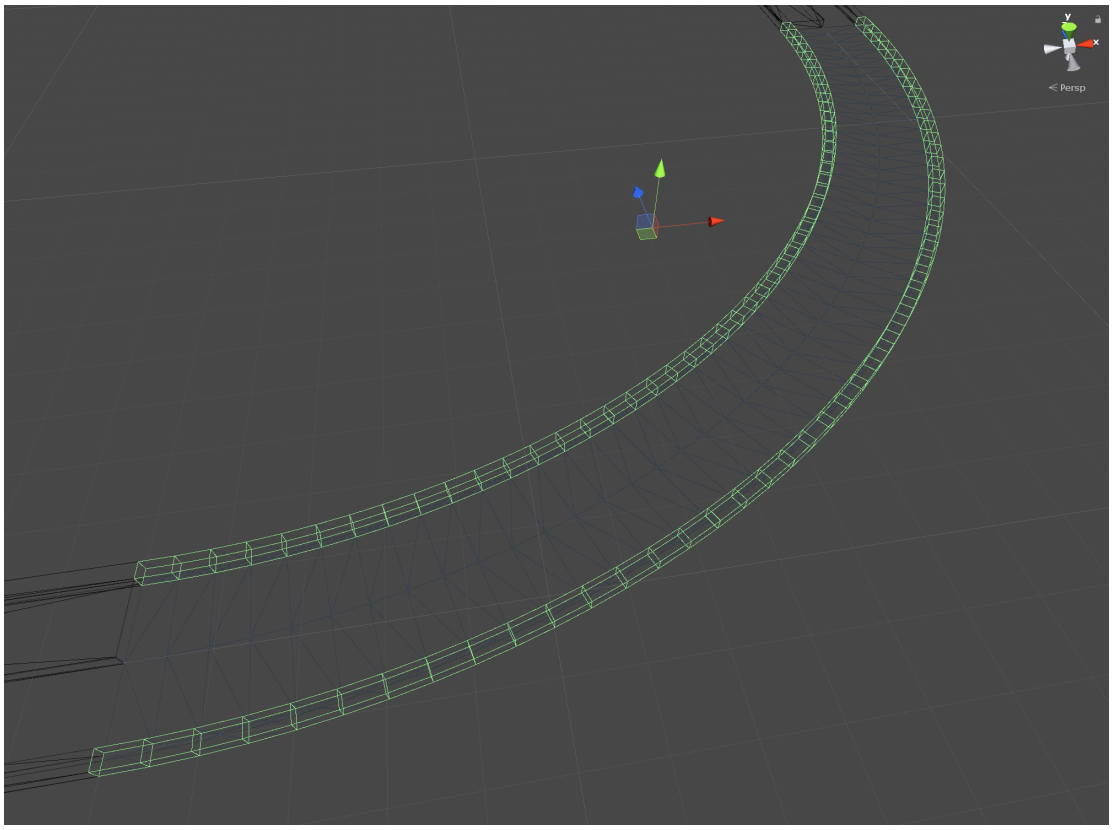


Figure 11: Procedurally generated collision geometry for zone-based punishments

Figure 11 highlights the left and right danger zones, which indicate that the agent is no longer on track (it is driving on the sidewalk or crossing the lane). For a clearer view the middle lane zone and the oncoming traffic zone are not shown in that figure, but they work in a similar way. In the editor these zones are indicated by different colors (the danger zones are red in Figure 10). The agent is unable to see them, as this an information that is not available outside the simulation. The unity camera allows excluding certain layers from being rendered by this camera. The objects just need to be assigned to the correct layers.

As part of the procedural generation, the middle of the lanes is calculated, which is useful for driving non-agent-controlled cars, placing objects or measuring the distance between the car and the ideal path it should drive (in the sense of being centered on the right lane).

5.2.4 Rewards

Many environments have some kind of natural intrinsic reward. Video games often provide a score that can be used as a reward signal. Often these rewards are given only for success or failure, so most of the time the agent does not get any feedback (by rewards) on his actions. The rewards are therefore given sparsely, which means that a clear relationship between actions and rewards is difficult to deduce. Typically, additional reward signals are constructed to circumvent this. This method is called reward shaping and requires careful tuning, as agents often find ways to maximize this reward without learning the desired behavior.

Further approaches are just mentioned a here:

- Hindsight Experience Replay [2]
- Auxiliary Tasks [10]
- Attention [19]

As for experimentation, different rewards have been implemented:

- Zones in this project are collision geometry that triggers a reward or punishment if the agent enters it. For example, corridor- like volumes are dynamically calculated when the road changes (see section Procedural generation). Many parameters allow the adjustment of these volumes. One can change whether it should overlap with the street; how accurately the road should be approximated and of course the reward/punishment given. A similar zone is calculated for the opposing lane and the small region in-between lanes, which allows detecting whether the agent is cutting corners or driving too close to the middle of the road.

- As the lane is calculated it allows measuring different distances. Implemented are the distance to the target lane and the angle between the street course and the car.

Due to the flexible nature of the system, further rewards can be implemented and experimented with in the future.

As the environment was implemented in an iterative fashion alongside the agent training, it is not sensible to compare the achieved rewards between training runs. Furthermore, the rewards were treated as hyperparameters that influence the objective of the agent and this option was preferred to altering the loss function of the PPO algorithm (to emphasize the generality of the approach, although the implemented interfaces allow to alter the loss if required). To have some guidance on how the agent performs, one can monitor the episodes length. Longer episodes indicate that the agent managed to stay in the simulation longer, without getting of track or colliding

with something. There is also an adjustable maximal length at which the agent will get reset automatically. This prevents agents from being stuck in a situation forever and can be used to make assumptions for resource management.

Yet the maximal episode length cannot be used as reliable measure, as the environment gets gradually more difficult due to curriculum training. The agent will usually need time to generalize to new conditions (lighting, etc.).

The targeted behavior is intuitively clear. The simulation allows to observe the agent, so one can monitor from time to time its progress. This also might spot issues with implementation such as bugs, poorly chosen parameters or exploitable specifics of the implementation. You might observe surprising behavior. See section Observations.

Rewards	**Description**
OffCenterPunishment	This value is multiplied with the distance from the car to the center of the target lane. This punishment is received by the agent at every timestep.
OffAnglePunishment	This value is multiplied with the normalized angle (0,1) between the path tangent direction and the forward direction of the car. If the car is facing the opposite direction this punishment is scaled with 1.
FailingPunishment	This value corresponds to the punishment for completely leaving the course.
CrashPunishment	This value corresponds to the punishment for colliding with something in the scene such as other cars.
TargetReachedReward	This reward is given when passing a checkpoint. When passing a checkpoint, the checkpoint will be placed further along the track.
CloserToTargetReward	This reward is given each frame the car moves in the correct direction. This can be mainly used as a reward for not failing.
DangerzonePunishment	This punishment applies at each time step if the agent crosses the lane border on either side.
OncumingTrafficPunishment	Same as DangerzonePunishment but for being on the wrong lane.
LaneTouchingPunishment	Same as DangerzonePunishment but for crossing the middle lane.
FurtherFromTargetPunishment	Opposite to CloserToTargetReward. Potentially used to avoid driving in the wrong direction (also backwards).
SteerDeltaFactorPunishment	This tracks the last steering command and punsihes rapidly changing steering commands.

6 Results and Interpretation

6.1 Visualizing the Convolutional Layer

This section visualizes the actor network, which successfully learned to follow the track by showing the individual activation maps as in [4]. This kind of visualization of the layers allows to examine the network in an intuitive way. This method can hint at certain characteristics of the network. A more sophisticated evaluation of the network is beyond the scope of this work. The corresponding code for creating more such visualizations will be available in the accompanying code repository.

The images in figure 12 represent the input to the actor network. These consecutive frames were sampled at frequency of 3.125 Hz (1 / (0.02s*16)). In the background we see the procedurally generated buildings. The facades are textures from the pixel2pixel dataset [9]. This temporary solution should indicate that those textures or more complex geometry could be generated as in section Asset generation. To model the robotics lab at the FU more closely one could replace these with different assets or just place the assets at fixed positons as in the ACM simulation.

Figure 12: Consecutive input frames for the actor model

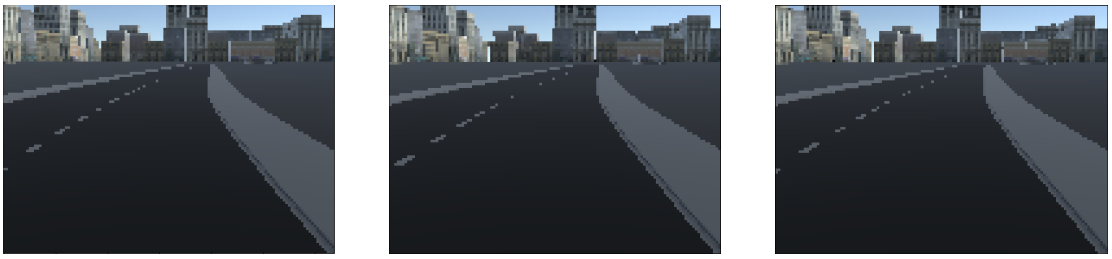


Figure 13 shows the activation maps of the first layer given the input shown before.

These early layers are responsible for detecting low level features such as edges in the input image. One can clearly see how certain filters highlight obvious edges due to the different colors for the lane and the lane borders.

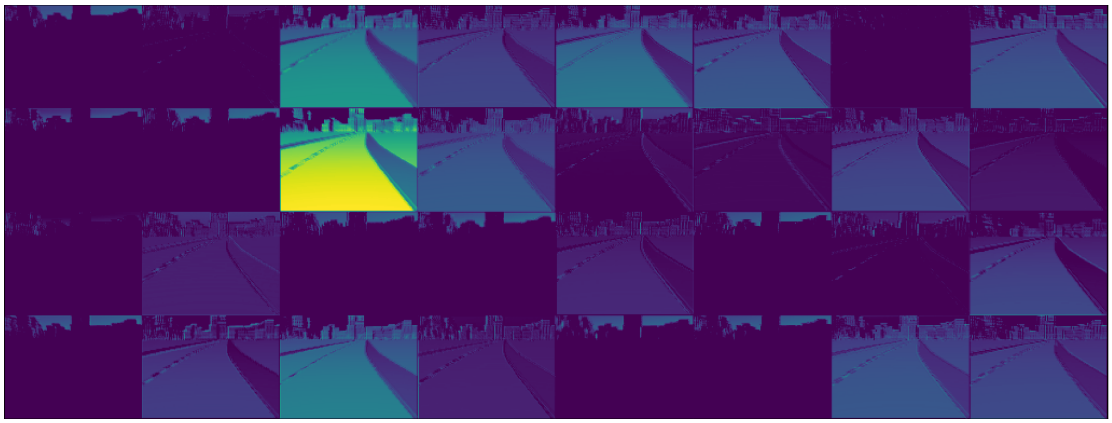


Figure 13: Activation Maps of the first Layer (Convolution) in the actor model

(All following activation maps have been rescaled for a better view. Additionally, the viridis colormap is applied to these activation maps, which are essentially grayscale images) The next layer is a MaxPooling layer, which highlights certain features, but also blurs the image to some extend as the resolution is reduced.

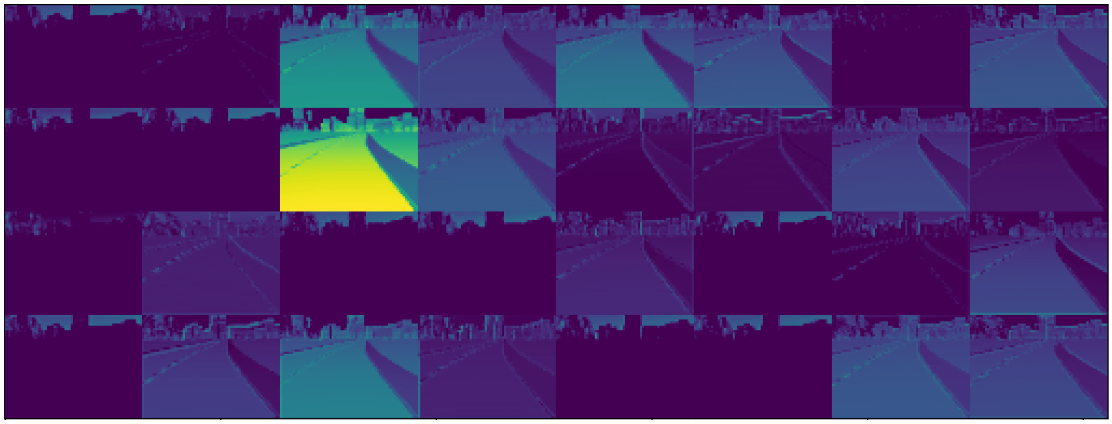


Figure 14: Activation Maps of the first Layer (Pooling) in the actor model

The next convolution layer contains more dark feature maps, but put attention to the remaining ones.

In the third row we have two feature maps which are somewhat opposite to one another, yet not just negations. While the filter at row 3 and column 2 highlights the border lane, the one right next to it (in row 3 and column 3) highlights the track. They are not just the opposite, since the objects to the left and to the right are visible in both activation maps. It is not surprising that the right lane border is still visible in many activation maps, as this is expected to be an important feature for not leaving the track. Notice that not all lines are still visible, as the stripped markings in the middle are missing as well as the left lane border.

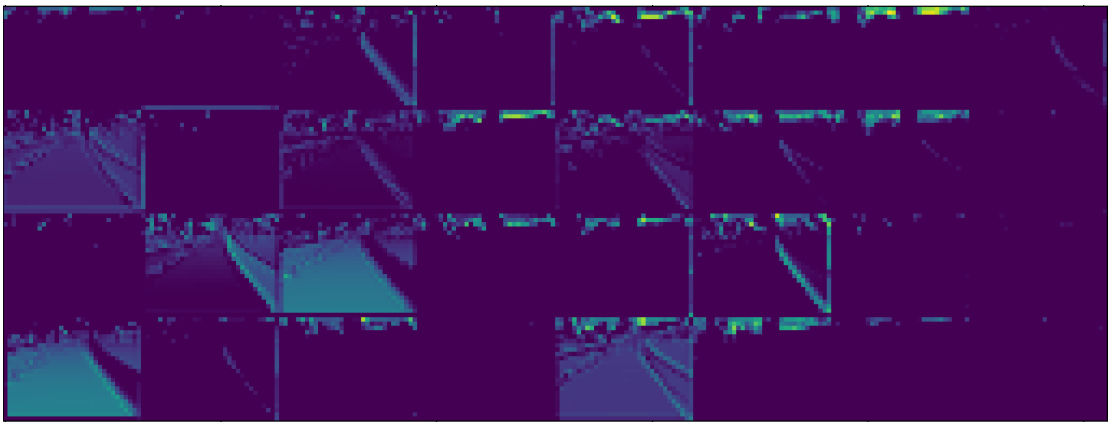


Figure 15: Activation Maps of the second Layer (Convolution) in the actor model

The following MaxPooling layer reduce the size further. By now the activation map size is 12x16 pixels. Notice that the fact that some filters are almost empty does not necessarily mean that they are poorly trained. It is more likely that the features that they detect are just not present in this sample or are very subtle. Also the absence of a certain feature can be an useful feature.

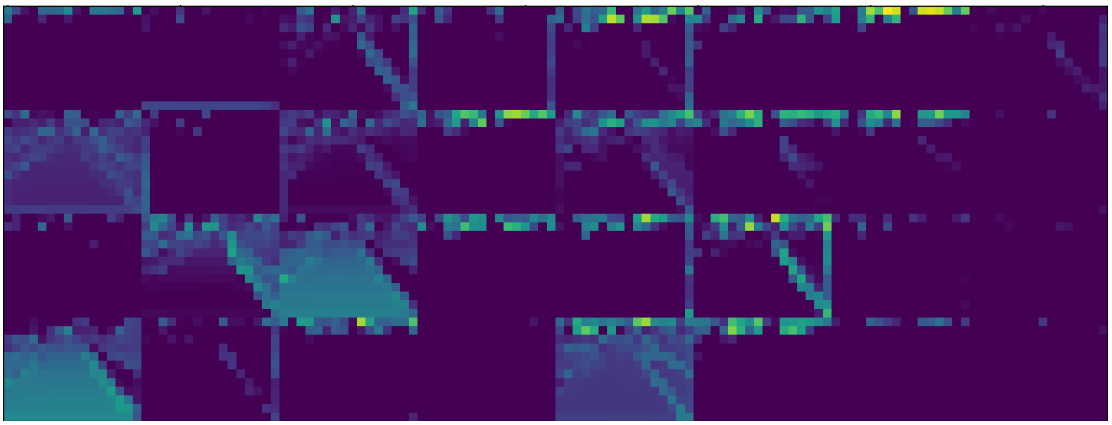


Figure 16: Activation Maps of the second Layer (Pooling) in the actor model

The final convolution layer mainly developed filters that highlight parts of track. Important features that are intuitively useful for staying on track can be seen in these activation maps. As the agent was rewarded for driving on the right lane, a plausible explanation could be that it navigates itself according to the rightmost thing that reassembles the lane border.

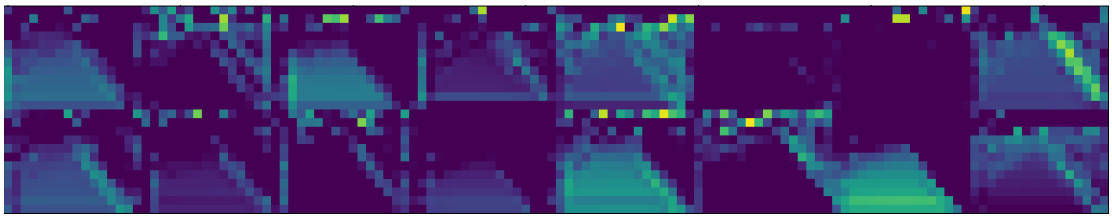


Figure 17: Activation Maps of the third Layer (Convolution) in the actor model

6.2 Observations

Certain observed behaviors are very interesting and often surprising.

- This scenario occurred in an early the stage of the project, where the agent was allowed to control speed and steering. The agent learned to stay on a straight road, but would loop between driving forward and backward. This behavior was caused due to a certain choice of rewards and their implementation. The agent was given a fixed amount of rewards (say r=1), if it managed to pass a checkpoint (a precomputed point along the road). A punishment of half that amount (say p=0.5) would be subtracted, if the agent got further away. If the agent passed a checkpoint, the next checkpoint would be a little bit further along the road. If the agent moved away, the checkpoint would be reset to the closest checkpoint. In this scenario the agent can exploit the environment in the following way. It drives forward until it reaches a curve, then reverses for some time and moves forward again. This is repeated until the episode ends. Thus, by driving for example 10 checkpoints forward (e.g. 10m) the agent will collect 10 times the reward r=1. By driving the 10 checkpoints backwards, it will get punishment p=0.5 ten times as well. In total it achieved 10xr -10 x0.5 =+5 rewards. The more often it repeats this cycle, the more rewards it will gain. Learning this behavior is certainly easier than correctly driving the curve (failing will end the episode and earn a large negative reward). The agent has thus found a local minimum, the resulting behavior is undesirable. From then on, the car was only in control of steering and the implementation of the checkpoint (closer to target reward) was adjusted, so it would not be reset. If this checkpoint implementation or a similar mechanic for rewards is intended, one should be aware and will most likely want to set the punishment as the same or more than the reward. Originally, the intention was to help learning by giving the agent more feedback (achieving sub-goals was rewarded in order to avoid sparse reward).

- For a while, the chosen track had too many sharp curves. After the behavior previously described, the acceleration was set to constant. This had the consequence of the car being too fast to be physically able to manage the curves. Therefore, the ability to monitor the speed and regulate it with a pd Controller was implemented. The track was redesigned to accommodate the targeted speeds. The network was queried at a 50 Hz rate and learned to reduce its speed about 10km/h by setting the wheels to a different position almost every

frame. This rapid turning resulted in a strong brake action. On average it was in the correct direction, which enabled the car to drift for a while and sometimes catch the curve.

- If the environment is too difficult, the agent might try to end it as fast possible, which can manifest in the agent driving maximally to the left or right. So, if the agent always receives a negative reward, it makes sense to end the episode as soon as possible.

- With a poor choice of parameters, the agent might get stuck in a local minimum. This usually manifests in the prediction of always the same value. In some cases, when the sigma for example is also predicted be the network, it might get quite low. This will result in dividing by a very low number (possibly becoming zero) in the loss, which will then result in NAN. Some implementations will not fail which can be very irritating.

7 Conclusion

The implemented system provides a smooth segue into computer science. The developed simulation enriches the existing laboratory environment with new alternating digital environments. This makes Reinforcement Learning based training of driving agents much more efficient due to increasing availability, decreasing costs and parallelism. The digital environment could easily be adjusted and randomly initialized to combine different trainings into one digital training session. On the whole, the simulation is a good resource for different kinds of AI training sessions and an ideal extension to the existing laboratory environment.

The developed self-driving agent supports this assertion. The agent was trained in the developed simulation environment and is now able to drive autonomously on a digital test course. The developed architecture can be used for further research and further approaches as well.

To sum it up, the result of this project enriches the existing research with new environments, training methods and approaches for a self-driving agent.

8 Outlook

This section is intended to inspire future work on the subject and to underline how this thesis could be of help. This is of course by no means an exhaustive list and some issues are still not fully solved. If not stated otherwise the author is not aware of any such ongoing research on this rapidly evolving field.

8.1 Asset Generation with GANs

Creation of different assets such as textures, material and 3d models is a very laborious task that requires a lot of working hours of experienced designers. Automation of this tedious process would reduce costs and allow more focus on the content. The paper [27] made a first step into this direction and describes a system for easily creating materials, given a set of examples and a means to explore a latent space that highlights potential interesting regions. The book [14] gives examples on how to use 3d convolutional GANS to create simple structures such as furniture.

And this not only applies to video games or simulations as in this case. Generated faces like this would make every character in such environments unique, but could also solve data privacy issues in a whole area of applications.

Taken this idea to the extreme, an adversarial setting could choose the curriculum to train the agents.

8.2 Multi Agent Simulation

Reinforcement learning offers a manifold of applications in the created simulation. This environment can easily be extended to simulate more complex infrastructures, such as cities.

The Unity ML-Agent API allows the handling of multiple agents within the same scene. Besides that, a lower resource footprint allows resources to be shared. This means that agents can interact with one another either in a competitive or cooperative way. The competitive scenario could be a racing track for example, where the agent drives against other instances of its policy (or slightly different ones such as in AC3 or genetic algorithms). Another non car related example would be the simulation of soccer games. Potentially new and quite different strategies could be explored.

The cooperative side is also very interesting to experiment with. Multiple car agents within one simulation can learn to minimize a common cost function (time, traffic jam, fuel usage) cooperatively. If these cars were not restricted to the usual traffic laws, they might discover better ones. By loosening the constrains (the agent fails if it leaves the track) in this simulation the agent might also learn interesting behavior.

Traffic in general could be optimized. It would be of general interest to find better street planning or regulation with traffic lights. The semaphores in modeled cities could be subject to optimization.

8.3 Photorealism

With the introduction of hardware accelerated raytracing on consumer graphics cards [17], many interesting applications can be expected. Support for this technology is in the beginning phase as of time of writing, but it is already integrated to some extend in Unity with its high definition rendering pipeline. Photorealistic renderings of the scene should reduce the gap between simulation and simplify deployment. Distorting factors such as reflections, god rays and lens flare would be already present in the training data. Depending on the integration implementation, different operations and therefore sensors could be realized more efficiently.

Understandable Networks
One issue with Neural Networks is that they generally operate as black boxes. It is hard to pin down what caused the network to give its answer (Why does the network choose this action over others?), which is an especially urging question for driving autonomously. Convolutional layers offer the techniques for visualizing the internal workings of the network, which allows to interpret them a little bit better. The following ones are the major one [4]:

- Visualizing activation maps
- Overlaying a computed heatmap over the input image according to the prediction
- "deep dream" (maximizing features in an image that activate a certain part of the network)

The heatmap approach is especially interesting, as it highlights the location in the input image that most contributed to the given prediction. If this prediction is unreasonable, one might be more careful. The heatmap could be compared to a segmentation to double check whether important classes such as pedestrians and other cars contributed to planned actions. This could be also tried for attention-based networks in a similar fashion.

A Appendix

A.1 Repository

You can find the entire project on Gitlab with additional documentation and resources. [26]

A.2 Used Packages and Modules

The system is based on the following Packages and Modules:

- Unity editor
- Mlagents
- Tensorflow
- TensorRT
- https://github.com/LuEE-C/PPO-Keras
- Keras
- TensorRT
- dependecies

Bibliography

[1] Open ai environments. https://gym.openai.com/envs/#classic_control, 2019.

[2] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *CoRR*, abs/1707.01495, 2017.

[3] P.W.D. Charles. Autominy - an autonomous model car for education. https://autominy.github.io/AutoMiny/, 2019.

[4] Francois Chollet. *Deep Learning with Python*. Manning Publications Co, 2018.

[5] Louis Clouâtre. Ppo-keras. https://github.com/LuEE-C/PPO-Keras, 2017.

[6] Johnson Vlissides Gamma, Helm. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[7] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[8] Ian Goodfellow, Yaroslav Bulatov, Julian Ibarz, Sacha Arnoud, and Vinay Shet. Multi-digit number recognition from street view imagery using deep convolutional neural networks. In *ICLR2014*, 2014.

[9] Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. Image-to-image translation with conditional adversarial networks. *CoRR*, abs/1611.07004, 2016.

[10] Max Jaderberg, Volodymyr Mnih, Wojciech Marian Czarnecki, Tom Schaul, Joel Z. Leibo, David Silver, and Koray Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. *CoRR*, abs/1611.05397, 2016.

[11] Richard Lemarchand Jason Gregory. *Game Engine Architecture*. CRC Press: Taylor and Francis Group, 2014.

[12] Arthur Juliani, Vincent-Pierre Berges, Esh Vckay, Yuan Gao, Hunter Henry, Marwan Mattar, and Danny Lange. Unity: A general platform for intelligent agents. *CoRR*, abs/1809.02627, 2018.

[13] MURREY KAHL. Autonomos model car simulator. https://github.com/Helliaca/AutoModelCar_Simulator, 2019.

[14] Josh Kalin. *Generative Adversarial Networks Cookbook*. Packt Publishing, 2018.

[15] Shawn McClelland. Announcing unity and havok physics for dots. https://blogs.unity3d.com/2019/03/19/announcing-unity-and-havok-physics-for-dots, 2019.

[16] Ian Millington. *Game Physics Engine Development Second Edition*. CRC Press Taylor and Francis Group, 2010.

[17] NVIDIA. Nvidia turing gpu architecture whitepaper. https://www.nvidia.

com/content/dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf, 2018.

[18] Robert Nystrom. *Game Programming Patterns*. Genever Benning, 2014.

[19] Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. *CoRR*, abs/1705.05363, 2017.

[20] Matthias Plappert. keras-rl. https://github.com/keras-rl/keras-rl, 2016.

[21] Andrew G. Barto Richard S. Sutton. *Reinforcement Learning: An Introduction Second Edition*. The MIT Press, 2018.

[22] Raúl Rojas. *Neural Networks: A Systematic Introduction*. Springer-Verlag, 1996.

[23] Shanmugamani Saito, Wenzhuo. *TensorFlow Reinforcement Learning Quick Start Guide*. Packt Publishing, 2019.

[24] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization. *CoRR*, abs/1502.05477, 2015.

[25] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.

[26] Robert Zifrid. Sdabodrl. https://git.imp.fu-berlin.de/robertzifrid/deepdrive/tree/master, 2019.

[27] Károly Zsolnai-Fehér, Peter Wonka, and Michael Wimmer. Gaussian material synthesis. *ACM Trans. Graph.*, 37(4):76:1–76:14, 2018.