Facilitating a Smooth Learning Agent Transfer from Unity Simulation to Autonomous RC Car

Chandler Stewart, Erica Truxton, Kaleb Brunt, Dr. Cen Li

1. Introduction

Reinforcement learning(RL) is an artificial intelligence (AI) training technique that allows a learning agent to train itself by initially taking random actions and adjusting the weights of its actions based upon a provided reward structure. Currently, reinforcement learning is primarily utilized when little interaction outside of a simulated environment is required. The most famous occurrences of reinforcement learning prowess, such as AlphaGo defeating Go world champion Lee Sedol, have all occurred on a computer with little real world interaction. As an undergraduate summer research team we sought to understand this simulation constraint by training a RL-model in a simulated environment and transferring it to a robot. Specifically, we decided to create an autonomous radio-controlled (RC) car capable of navigating a track.

The broad idea is to train the learning agent in a simulated environment with realistic physics and provide inputs that would be easily replicable when the model is transferred to the real world. Throughout the process we encountered many issues that forced our growth. Beginning the project with little neural networks background, we first needed to develop the baseline AI knowledge necessary to progress the project, so we completed a Udemy course (Slim, Rayan, et al.). The course taught the basics of neural networks, OpenCV for image manipulation, and Keras for RL-model training, among several other concepts. While actively working to build the RC car we encountered problems with simulation environments, software and hardware compatibility, and model optimization. Thus far, we have overcome all issues with exception to model optimization. Ultimately, the task of creating a self-driving RC car remains unsuccessful, but it has been an incredible learning experience and we intend to continue this project until its fruition.

2. Methodology

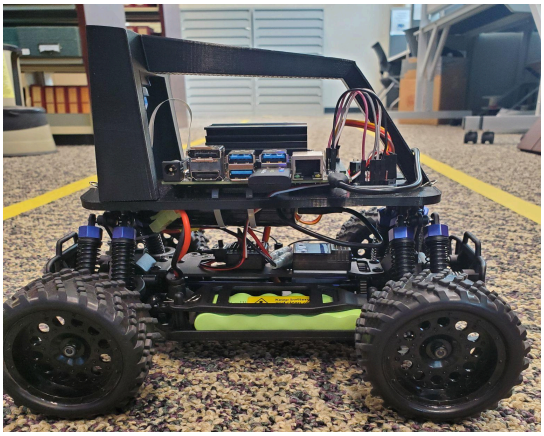
The project methodology can be broken up into six parts: (1) building the RC car (2) choosing and creating a training environment; (3) lane image processing; (4) training the reinforcement learning agent; (5) transferring the model between environments; (6) testing and model optimization.

Step 1: Building the RC Car

A pre-built RC car was purchased and modified based upon instructions provided by the online DonkeyCar community (“An Opensource DIY Self Driving Platform for Small Scale Cars.”). Pictures of the finished product are shown in figure 1. Additionally, the components and their purpose are listed below.

Figure 1. Assembled Remote Controlled Car

Front/Left Side View **Right Side View**



1/16 2.4Ghz Exceed RC Magnet	Base RC Car
Nvidia Jetson Nano 4GB Developer Kit & 64 GB SD Card	Stores and runs the program responsible for controlling the RC car's actions and learning agent inferencing
Sunfounder PCA 9685 Servo Driver	Connects to the Jetson Nano with four female to female jumper wires and is responsible for translating actions from Jetson Nano to the RC car
Anker Astro E1 Portable Charger	Powers the Jetson Nano
Pi Camera V2.1	Captures images and sends them to the Jetson Nano to utilize as input for inferencing
3D Printed Chassis	Houses all additional components
Panda 300Mbps Wifi Adapter	Used to connect the Jetson Nano to Wifi

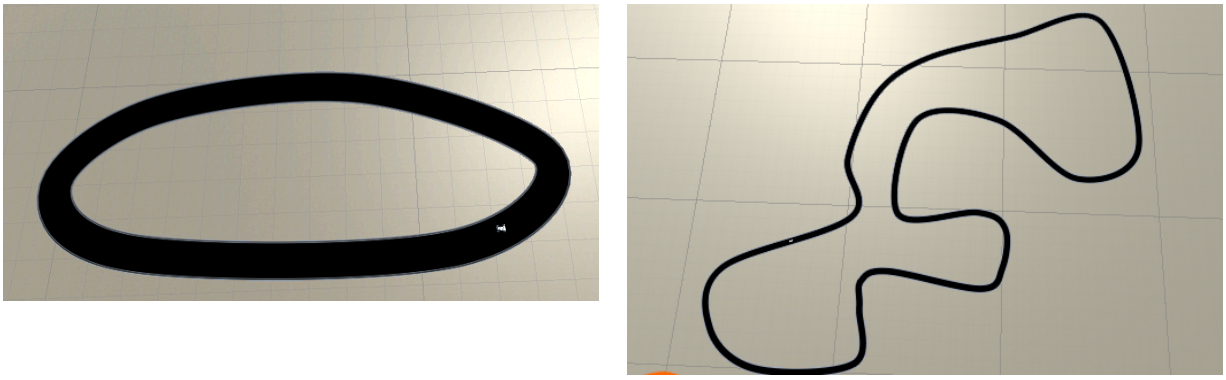
Step 2: Choosing and Creating a Training Environment

When determining a training environment we looked for several factors: realistic physics, adaptability, and training speed. Realistic physics is the most important of the three. It allows us the ability to ensure that the model is making the same actions in the simulation as it would in real life under similar circumstances. For adaptability, we needed to ensure that we had control over the environment itself. For example, the track the car drives on is very important because it needs to learn how to navigate turns with varying angles. In addition, the existence or lack thereof of trees, buildings, or other potentially distracting background noise must be accounted for. Lastly, prioritizing training speed allowed us to train various models in order to test variables such as maximum speed, camera visibility, and model inputs.

Initially, the CARLA Simulator was chosen (Carla-Simulator). This is a simulator built in Unreal Engine 4 primarily for the purpose of autonomous driving research. At face value it seemed to fit all of our needs although as we began training, its faults came to light. During our tests we discovered that the CARLA simulator was poorly optimized which resulted in incredibly slow training speeds. After much deliberation we decided to put CARLA to the side and continue with our second option, Unity. Unity coupled with their ML-Agents library is both very adaptable and well optimized allowing for fast training speeds. The main concern utilizing Unity is the issue of realistic physics. Luckily, we found an open source Github repository that resolved the dilemma(Yu, Felix). Once solved, we proceeded to create our own tracks for training purposes. Pictured below are the two tracks that were utilized for training. Figure 2 illustrates two tracks created in Unity that were utilized in our training. The first track was created primarily as a proof of concept and once training was successful, all subsequent iterations were performed on the more complicated track 2.

Figure 2. Example tracks used within training simulation

Track 1 **Track 2**



Step 3: Lane Image Processing

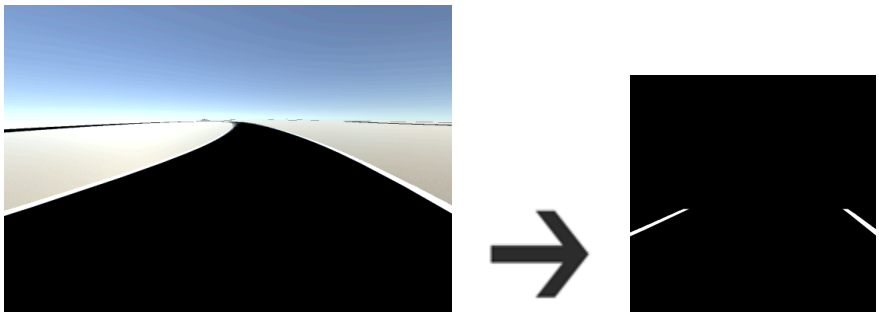
Lane image processing is utilized to recognize only the edges of the track that the car is driving on and ignore everything else in the background. The task at hand was created using OpenCV's image manipulation capabilities in Python.This method is keen in achieving not only efficient training of the AI; but successful decision making in the RC car itself.

There are two reasons why lane image processing is important in the development of our self-driving car. The first is to significantly cut down on the processing power and time to train our models. Originally our models were training off of 84x84 grayscale images. By implementing this method of only detecting the edges of the track and cutting out all extra background noise we found that we can successfully train a model in less than half the time that was originally required, which is now roughly 2 hours. The second reason we implemented lane image processing is because we want our autonomous learning agent to base its decisions (ie. angle and throttle) off of the location of the lanes. We then applied it to the camera of our Jetson Nano to replicate the images that Unity sends to the learning agent.

Our methods to achieve lane image processing differed between simulation and the hardware on the RC car itself. In Unity we altered how the camera interprets the environment which was through changing the layer in which objects appear. Therefore, we only see the front

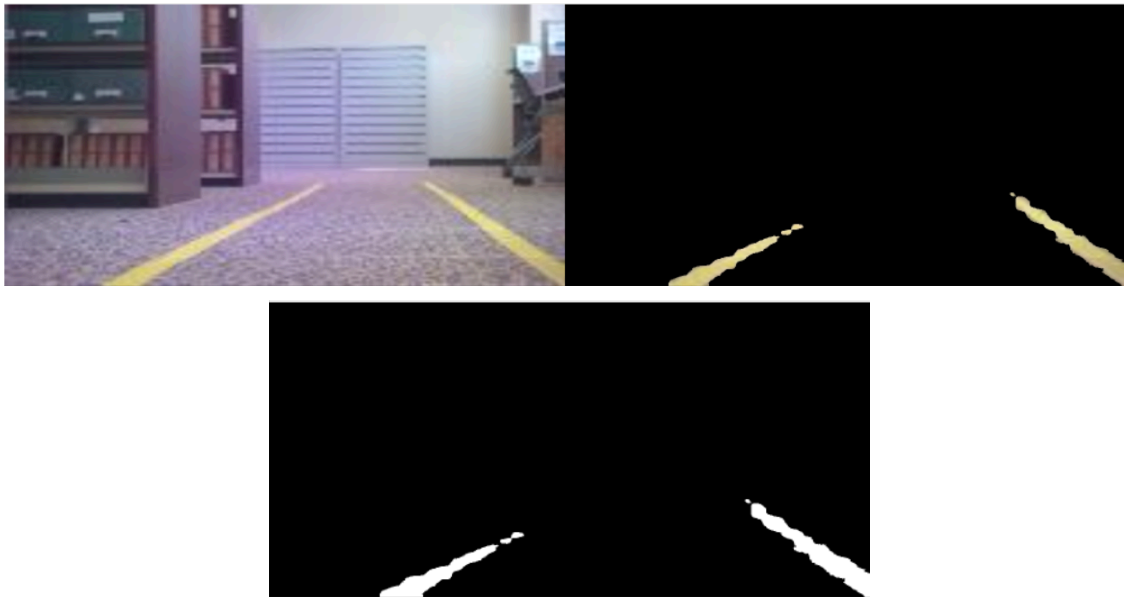
of the track that the simulated car is driving on and anything else in the background is masked out. We then converted the images captured to 84x84 grayscale. A transformation of those images is shown below in Figure 3.

Figure 3. Lane Image Conversion Captured by Unity Program



For our RC car to only capture the lanes of the track and mask the background we created a Python program that utilizes OpenCV to only recognize a certain color. We achieved this by converting the images captured from BGR to an HSV (hue-saturation-value) color space. The hue describes a color in terms of saturation, which represents the amount of gray in the color and the value describes the intensity of that color (“Multiple Color Detection in Real-Time USING PYTHON-OPENCV.”). An HSV color can be represented in a 1x3 matrix ranging from 0-179, 0-255, and 0-255 respectfully. We chose yellow to represent the color of the track and masked out all other colors in each image captured that weren't within a certain range of two different 1x3 matrices: [0, 59, 150] to [75, 255, 255]. We then overlaid any remaining yellow color with white. This allowed us to replicate what our simulation sees in real time. Figure 4 givens an example of transformation of the images captured by our RC car:

Figure 4. Lane Image Conversion Captured by RC Car



Step 4: Training the Reinforcement Learning Agent

The learning agent was trained in Unity using their ML-Agents library. ML-Agents includes a Python API that interfaces with the Unity environment for the purpose of model training. The library abstracts away a significant portion of the model architecture allowing us to focus on training rather than model composition.

The training process begins with providing the learning agent access to its two action options: throttle and steering angle. Initially it has no understanding of the implications of its actions and will randomly drive forward, backward, left and right. Although after numerous iterations of the simulated car being 'rewarded' for positive behaviors and 'punished' for negative ones it eventually learns the correct actions to stay on the track.

The reward structure is vital to training in order to encourage the learning agent to perform the correct actions. Training began by providing the simulated car with a reward of '0.1' for every second it was able to stay on the track while being punished by '-1' if it fell off. Utilizing this reward structure resulted in the car staying on the track but barely moving due to receiving no reward for movement. After some changes we settled on the reward structure of: 'speed x 0.01' for every second the car stayed on the track, while maintaining the original punishment. This reward structure resulted in the vehicle efficiently navigating the track and never falling off after only two to three hours of training time, although increasing the max speed of the vehicle also resulted in increased training time.

Additionally, ML-Agents allow for multiple vehicles to train at the same time increasing training speeds at the price of hardware resources. All models that we trained were trained utilizing four vehicles at the same time. The training hardware can be found below in addition to a video demonstrating a trained learning agent navigating the simulated track.

Link to a trained learning agent navigating the simulated track:
https://www.youtube.com/watch?v=tG1nwLcTZW0

Training Hardware

Processor: AMD Ryzen 7 1700

Memory: 16GB

Video Card: Nvidia GeForce GTX 1060 6GB

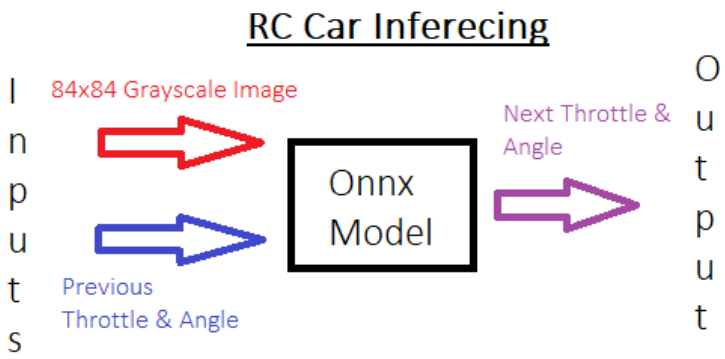
Step 5: Transferring the Model between Environments

After successfully training a learning agent in the Unity environment, the next challenge was the logistics of transferring the trained model to the RC car. The aforementioned DonkeyCar community maintains open source code available which we utilized to control the car, although they do not support the onnx file type that is output by Unity. This left us with two options: either convert the onnx file to one already supported, such as h5 used with Keras or tflite for Tensorflow Lite, or write our own inferencing class that supports onnx. The first option

seemed best at face value although after hours of errors while troubleshooting, we decided to write our own inferencing class capable of onnx support.

When inferencing is performed the model requires the same inputs it was trained on in order to provide an output corresponding to its next action. These inputs include: the image scaled down to 84x84 and converted to grayscale, and the car's previously output throttle and angle. The resulting outputs are two values each in the range of -1 to 1. One of the output values corresponds to the throttle while the other is the angle. Once the inference outputs are obtained they are converted to values understood by the vehicle and returned for action processing. An illustration of inferencing is shown below in figure 6.

Figure 6. Illustration of the RC Car's Inferencing



Step 6: Testing and Model Optimization

At this point in the project the RC car is capable of receiving and performing actions from an onnx model trained within a simulated environment. Additionally, the inference input data is closely correlated to the simulated training input data by utilizing lane image processing. Unfortunately, a multitude of variables remain that have the ability to cause the RC car to function erratically. Examples include differences in image depth or perspective, an incongruity between the simulation and real life physics, or simply the throttle and angle processing being performed differently from the simulation.

The first step we took to resolve these issues is to temporarily circumvent the RC car's camera and provide it with the frames of a prerecorded video from the simulation environment. This way, we are able to visually check and ensure that the RC car is making the same actions as would be performed by the simulated vehicle with zero inconsistencies in input data. The results of this test showed that the RC car did indeed perform similar actions as it would in the simulation (eg. maintaining constant throttle, turning left when the simulated vehicle would turn left and vice versa). This test showed that calibration was similar between environments, but did not yet confirm it was exact.

The second step was observing the vehicles actions when the learning agent receives the real life lane processed images as input. While observing we noticed that the vehicle would initially stay within the lane boundaries but would skew its angle slightly right leading it to eventually driving off the track. A bias of training data favoring right turns was our earliest idea

for the cause. However, multiple tests on other models intentionally trained to ensure equal distribution of left and right turns provided the exact same results. This has led us to believe a slight difference in calibration and angle processing exists between environments that results in the right skew. We are actively troubleshooting this issue, although this has marked our current stage of the project.

3. Results and Discussion

Currently, our method to train an autonomous learning agent using reinforcement learning through Unity's software and the ML-Agents library is successful. This is due to our optimized reward structure, lane image processing, and our ability to train multiple simulated cars at once, sending the data to a singular model. It takes roughly two hours for a simulated car to learn how to consistently drive around a track without falling off. We are incredibly pleased with these results and will continue to optimize further as we see fit.

Additionally, we have created a remote-controlled (RC) car equipped with wireless capabilities to send commands to the car, view the camera in real-time, process images to alter how the camera perceives the environment, and successfully utilize an onnx model trained through Unity to make decisions based upon simulated training. The RC car can successfully stay on a straight track and make correct decisions to keep itself on a path. Even if we start the car at a skewed angle, it has the ability to correct itself to stay within the track's edges. However, our RC car does not make the correct decisions with 100% accuracy. The next step is performing controlled tests to determine current causes of fault, such that it can take sharp turns to navigate around a curved track. While we are pleased with our current progress, this is an issue that we are actively troubleshooting and currently believe that either altering the calibration of the RC car or modifying the training environment will provide a resolution.

While the ultimate task of creating a self-driving car through reinforcement learning remains unsuccessful, our team will continue making strides to see this project to completion. Despite experiencing unanticipated hurdles, this project has been a wonderful learning experience and has given us the knowledge on how to utilize reinforcement learning to train a learning agent. We've learned to navigate Unity's software with its ML-Agents library, utilize OpenCV for tasks such as color recognition, transfer trained models from a simulated environment to an RC car, and much more. The lessons learned throughout this project have implications for future ventures with endless possibilities. Some ideas include making a mouse versus maze project where a 'mouse' accurately learns how to navigate through a maze using reinforcement learning, Using an infrared camera to detect certain colors using OpenCV's talent for image manipulation, or possibly evolving the RC car to provide collision detection. We are overwhelmingly grateful for the opportunities given to us by the S-STEM undergraduate research program, professor Cen. Li's guidance, and Middle Tennessee State University's Maker Space for giving us adequate resources to aid in the project.

References

"An Opensource DIY Self Driving Platform for Small Scale Cars." *Donkey Car*, www.donkeycar.com/.

Carla-Simulator. "Open-Source Simulator for Autonomous Driving Research." *GitHub*, github.com/carla-simulator/carla.

"Multiple Color Detection in Real-Time USING PYTHON-OPENCV." *GeeksforGeeks*, 10 May 2020, www.geeksforgeeks.org/multiple-color-detection-in-real-time-using-python-opencv/.

MSV, Janakiram. "Tutorial: Using a Pre-Trained ONNX Model FOR INFERENCING." *The New Stack*, 15 Dec. 2020, thenewstack.io/tutorial-using-a-pre-trained-onnx-model-for-inferencing/.

Slim, Rayan, et al. "The Complete Self-Driving Car Course - Applied Deep Learning." *Udemy*, Udemy, www.udemy.com/course/applied-deep-learningtm-the-complete-self-driving-car-course/.

Yu, Felix. "Train Donkey Car in Unity Simulator with Reinforcement Learning." *Train Donkey Car in Unity Simulator with Reinforcement Learning | Felix Yu*, 11 Sept. 2018, flyyufelix.github.io/2018/09/11/donkey-rl-simulation.html.