Project Don't Crash

Jaimin Patel, Naga Sanketh Vysyaraju, Abhivineeth Veeraghanta, Abhilash Karpurapu,
Haripriya Dharmala, Shreya Kunchakuri

Problem Statement

The goal of this project is to build an agent to perform autonomous driving in a simulated environment. Autonomous driving is an important area of research as it can make roads safer by helping to reduce the number of avoidable accidents caused by human error in driving [1]. It can also allow those with disabilities or senior citizens to be able to travel independently and safely. Simulation is key to self-driving research because we can test customized situations without facing real-world consequences. For our project, we have been using AirSim [2], a simulation environment created for autonomous driving research. We would like to test various objectives, such as the longest distance without crashing, longest time without crashing, speed, and interacting with moving pedestrians. We want to build on AirSim's framework by employing Computer Vision algorithms for perception and training Imitation Learning and Reinforcement Learning agents to perceive and act in our simulation. Furthermore, we wish to compare the agent's performance when trained on Reinforcement learning and Imitation learning.

Environments

Finding suitable simulation environments for autonomous driving was an important first step. A number of environments were assessed including racing game environments, toy simulators and environments made specifically for experimenting with self-driving cars. In selecting our environment, tradeoffs between installation requirements, the complexity of the environment, and features offered were important factors influencing our decision. The majority of the simulators are based on either Unity or Unreal Engine with the latter not facilitating virtual machine installation. Given that there are multiple objectives of the project, choosing an environment such as a racing game did not provide the flexibility to create custom objectives. We opted to explore open-world autonomous driving simulators such as Carla, Voyage Deepdrive, Microsoft AirSim, Duckietown, and Udacity's Self-Driving Simulator. In the initial weeks, each team member selected an environment to explore and attempted to install it on his/her system.

Carla [3] which uses Unreal Engine seemed like our best option, it is a relatively new open source simulator. It offers a range of functionality including a scalable multi-client architecture, an API for simulation, traffic scenario generation, pedestrian behavior customization, access to various types of sensor data, and pre-trained baseline agents. The main caveat was that it is only installable on Windows or Linux non-virtual machines. For the three team members with access to Windows machines, the setup and installation process was plagued with bugs and

after weeks of attempting to fix them through looking at forums and posting issues to the GitHub page, we decided to not move forward with it.

Voyage Deepdrive [4] is an open-source simulator made by self-driving company Voyage, which also offers a range of functionality for research with access to environments such as a realistic 25 block urban map with AI agents based on San Francisco. Unfortunately, this environment is made only for Linux non-virtual machines and hence we couldn't set it up.

Duckietown [5] is a simplistic environment for robotics, consisting of a robot agent operating among roads and dynamic objects such as ducks and cars. This environment extends beyond simulation with hardware available to set up in research labs. Duckietown also hosts the AI Driving Olympics, in which teams complete self-driving tasks with real hardware. We successfully installed and tested this environment, although easy to use, we found its functionality to be limited for our required objectives. There was only a single speed setting for the robot and the environment itself was small containing only two small intersections and four roads.

Udacity Self-Driving Simulator [6] is a Unity-based simulation environment set up for Udacity's Self-Driving nano-degree, it was made open-source recently. It features an environment in which a car can be controlled through the keyboard or an API. The scope of the simulator again was limited for our goals as there is limited customization of the environment and lack of the presence of dynamic objects such as other cars or pedestrians in it.

Microsoft AirSim [2] was the environment we selected. It is an open-source simulator which uses Unreal Engine for self-driving and quadcopter research. It provides pre-built semi-realistic environments such as city and neighborhood maps. It also offers the functionality to create your own environment from a template where static or dynamic objects can be created. The main limitation of the pre-built environments is that they only work on Windows machines. Other useful features include a recording log consisting of images, and data such as speed, steering angle, position, acceleration. A Python API is also available to control the car autonomously with template agents and algorithms. This API is where we implement our agent to control the car.

Prior Work Related to Autonomous Driving

One of the first systems to employ imitation learning to train an agent was the NAVLAB car based on the ALVINN model [7], in 1989. This agent learned to predict steering angles and follow lanes on public roads. Muller et al [8] were able to perform obstacle avoidance in a cluttered backyard with their robot car, DAVE. They used a CNN based architecture to perceive the distance between the car and obstacles, using images captured by DAVE's two cameras.
NVIDIA created an End-to-End learning model using CNNs [9] to map pixels from a single front-facing camera to steering commands. They set up a Data-Acquisition car which has three

cameras installed: left, right, and center. These cameras work in tandem with the steering wheel. The cameras are able to record what the vehicle sees whenever the steering angle is changed, which helps the agent learn and imitate the driver.

In recent times, simulators have allowed more researchers to experiment with different kinds of models to perform autonomous driving, such as Koutnik et al [10] who built an agent to drive in a TORCS environment. In 2020, Sushrut et al [11] were able to use model-free Deep Q Networks to allow multiple agents to drive autonomously while communicating with each other. For a more in-depth overview of prior work refer to the 'Related Work' section in the technical document.

Methods

Computer Vision

The initial step in creating a self-driving car is ensuring that it can perceive its environment. This is a vast area of research in the autonomous driving industry with companies utilizing cameras, lidar, radar, and satellite maps as inputs to make decisions. For a car to determine the next actions, it must be aware of current surroundings and be able to predict changes.

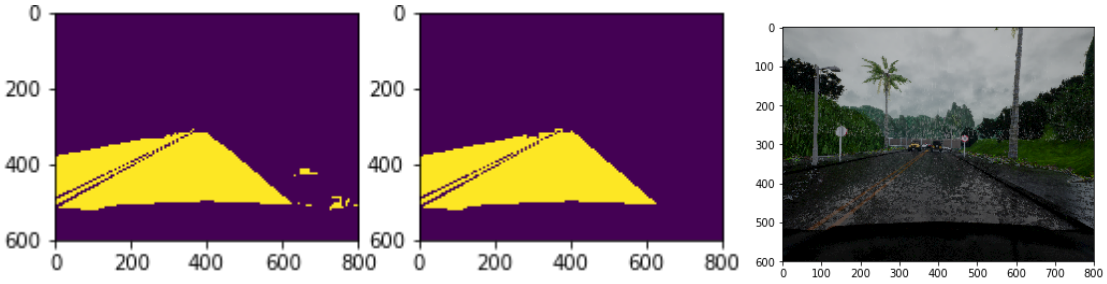
There are many problems an autonomous vehicle faces in everyday driving that Computer Vision can help solve. The image data is rapidly processed and actions are taken in real-time in a dynamic environment. Convolutional neural networks (CNN) provide an excellent framework for tackling these challenges.

Semantic Segmentation

A car must be able to separate surfaces and objects from one another. This allows the car to transform a live stream of pictures into a segmentation where objects and surfaces can be identified. A key challenge is road detection, determining what surfaces are drivable (asphalt) and what are not driveable (sidewalks, other vehicles, etc.), this plays a key role in the positioning of a vehicle. In our simulation starting with a segmentation, producing a mask of the road in front of the car ensures that the vehicle takes actions to stay on the road.

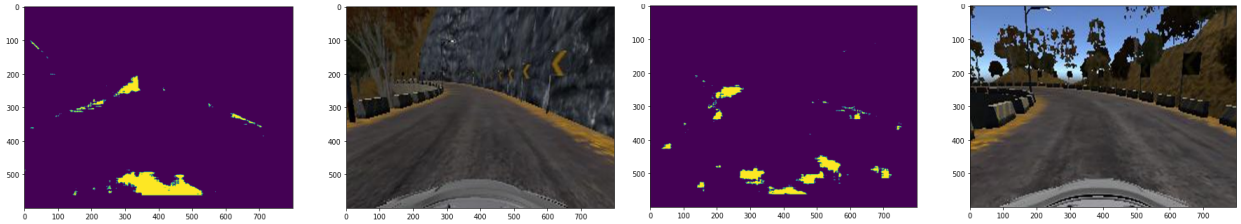
It is treated as a supervised learning problem using CNNs with the input as the image and the output as a labeled mask with boolean, one for each pixel. There are many approaches to this task including Mask R-CNN (see object detection section). AirSim provides a pre-built segmentation filter on the images. This segmentation, unfortunately, is not helpful for us as it distinguishes sky from land and building surfaces but it fails to separate road and sidewalk. There are many pre-trained networks that attempt to address this task [12], some on real images and some in simulation. We leverage a pre-trained network generated from Carla [3]. A Kaggle user has uploaded a labeled dataset of images [13] with associated masks distinguishing specific

surfaces and a pre-trained network for this task As seen in *Figure 1*. They train a U-net, a modified fully connected CNN architecture, originally developed for biomedical image segmentation with little training data [14]. Here is an example of the pre-trained network working on their dataset.

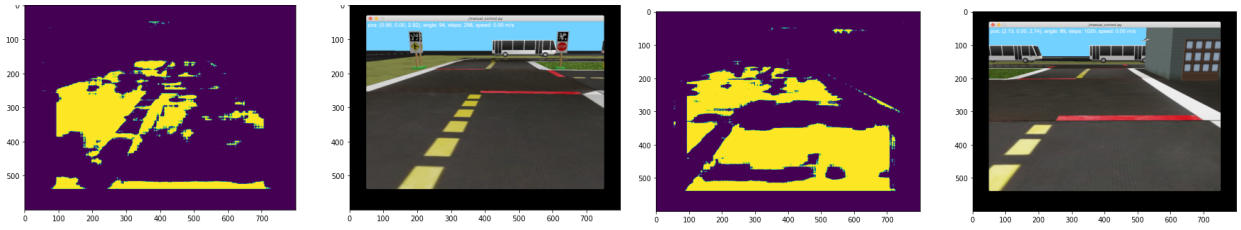


a) Prediction *b) Ground Truth* *C) Input Image*

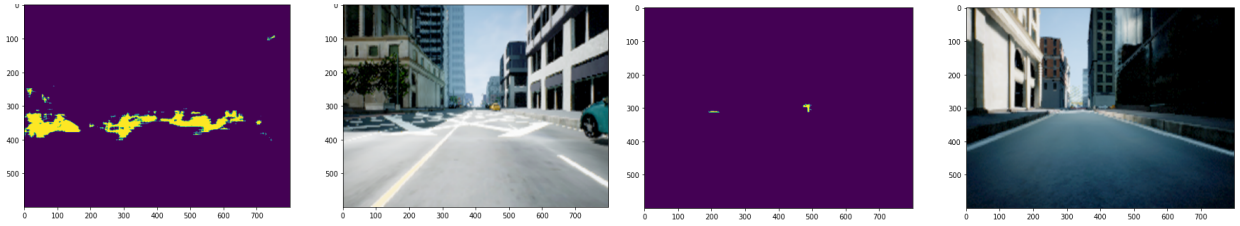
Figure 1: Carla environment road mask segmentation (The dataset the model was trained on) We applied this pre-trained U-net CNN to images from the simulation environments, the results are shown below.



a) Udacity Self Driving Simulator



b) Duckietown



c) AirSim

Figure 2: Pre-trained U-net on our simulation environments

As seen here,the CNN network does not create a good segmentation of roads across environments. The best performance is in Duckietown, where the road is recognized. Unfortunately, the model fails to provide a usable segmentation on AirSim images, the environment we settled on.

Generating labeled training data for the road surface with AirSim is a difficult task requiring a manually drawn mask to segment the road from other surfaces. Rather than producing our own labeled dataset with thousands of images and road surface masks, we utilize the pre-trained U-net CNN for our project. To implement transfer learning, we manually labeled 60 images with an online tool, Labelbox [15] giving road segment masks similar to the labeled Carla dataset. This procedure outputs a mask of 1s where the road segment is present.

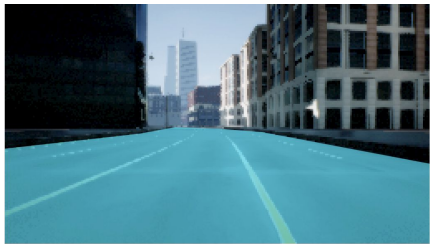


Figure 3: Labelbox segmentation mapped in blue on AirSim image

Next, the AirSim images and produced masks are scaled to 600 x 800 pixels (required input size), and the output layer of the pre-trained network was replaced in Keras. The rest of the network was frozen and the network was retrained for 10 epochs with a batch size of 2. The resulting network produced segmented masks of the road as shown below.

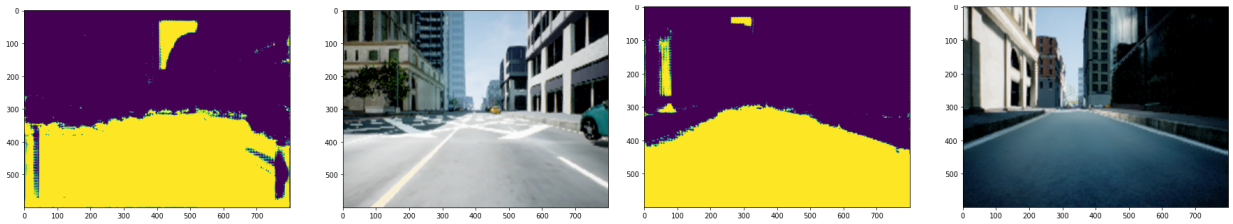


Figure 4: U-net CNN after transfer learning on AirSim images

As we can see, this CNN performs much better after applying transfer learning. Although not quite perfect, the mask segments parts of the sky and buildings as road, this output is still useful in distinguishing the shape of the road from the image. This will be an important input for our reinforcement learning algorithm.

Lane Detection

Lane detection is another important task in autonomous driving which is tackled through computer vision. It is important for a car to stay in lane when cruising so it will not cross paths with other vehicles. Identifying the lane lines is necessary for a car to do this.

Our experiments with lane detection are performed using 3 models.

1. A model based on Canny filter
2. Transfer Learning using VGG-16 Model [16]
3. Mask R-CNN Model

All the above models were only able to detect the lanes in a few pictures but the rest of the predictions were erroneous. This happened due to certain limitations as stated below.

The Canny filter model was built by converting the image into grayscale, then converting into gaussian blur, canny filter and a region of interest is selected from the image. But the brightness levels were not uniform among the images and the dashed lines are not clearly demarcated. only certain images were identified with lanes for a universal value of threshold.

The CNN model was also built using a VGG-16 architecture pre-trained weights but the predictions were not accurate.

The Mask R-CNN Model is built by using transfer learning on images trained with real world pictures using the same backbone architecture but the transfer learning model failed in predictions owing to the pixelated images of the environment.

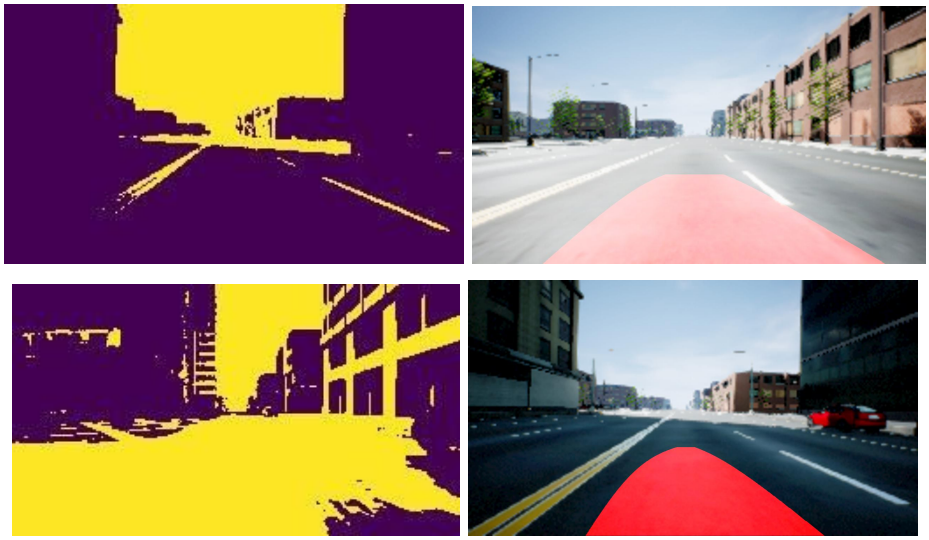


Figure 5: *Lane detection through VGG-16 and Mask R-CNN*

Object Detection

Mask R-CNN

Mask R-CNN is an instance segmentation technique [17], which locates each pixel of an object instead of a bounding box. It classifies the image to the class it belongs to and makes dense predictions inferring labels for each pixel in the classified object of the image for the whole input. It provides not only the classes but the spatial location of the classes that are predicted.

We took snapshots of the environment at certain frequencies to generate masks depicting various objects using Mask R-CNN. This will later be trained by an imitation learning network or deep reinforcement learning network for predicting actions.

A few of the segmented images of the environment are shown below:

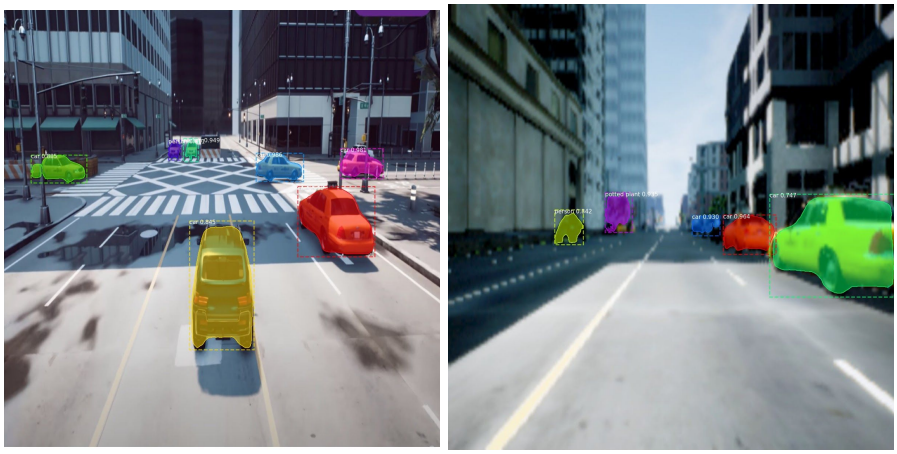


Figure 6: Mask R-CNN Object Detection

YOLO

For a self-driving car, it is important to not only consider where the objects are located relative to the car but also what the objects are. For example, if the car is driving in the direction of a pedestrian, it makes sense to stop and wait for the pedestrian to cross. However, if the object is a parked car, the agent is better off driving the car around the parked obstacle.

For our project, we decided to choose YOLO v3 [18] due to its high accuracy and speed. Since there are 53 layers in the architecture, it is computationally expensive and takes a long time to train even with a powerful GPU. For the purposes of our project, we used the pre-trained Darknet YOLO v3 model, which was trained on images in the COCO dataset. This network is able to identify various objects such as pedestrians and vehicles in the AirSim environment.

After performing object recognition, we created a metric to perceive the relative distance from the camera. We calculate this closeness as c = k(Area)/Distance, where distance is the length

of the perpendicular drawn from the bottom center of the bounding box. Since the City environment consists of a flat landscape, we can assume that the longer the perpendicular, the farther away the object is. We also take area into consideration, as objects that are closer have a larger area.

Since the area of closer objects is much greater than those farther away, the resulting closeness value can be very large for nearby obstacles. Hence, we multiply with a constant k which lies between (0,1). We have chosen the value of k to be 0.1, so as to normalize this value of closeness.

This closeness can be useful for the car to determine which objects are more likely to be collided into. We plan to modify the rewards function of our Reinforcement Learning agent to consider closeness to neighboring obstacles so the agent minimizes its chances of crashing and learns to navigate around pedestrians and cars.

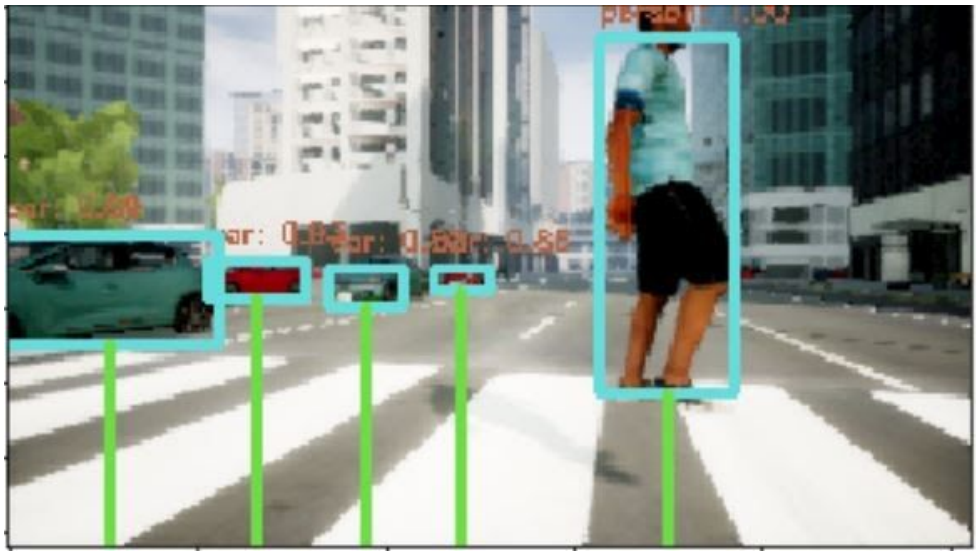


Figure 7: a) Depicts the bounding boxes (in blue) and classification done by the YOLO v3 model b)The lines in green are perpendiculars which help determine how close an object is to the car.

Mask R-CNN Vs YOLO

Mask R-CNN and YOLO are both popular object recognition algorithms, and for the purposes of our project, we decided to test both algorithms on images captured from AirSim's City Environment.

Mask R-CNN **YOLO**

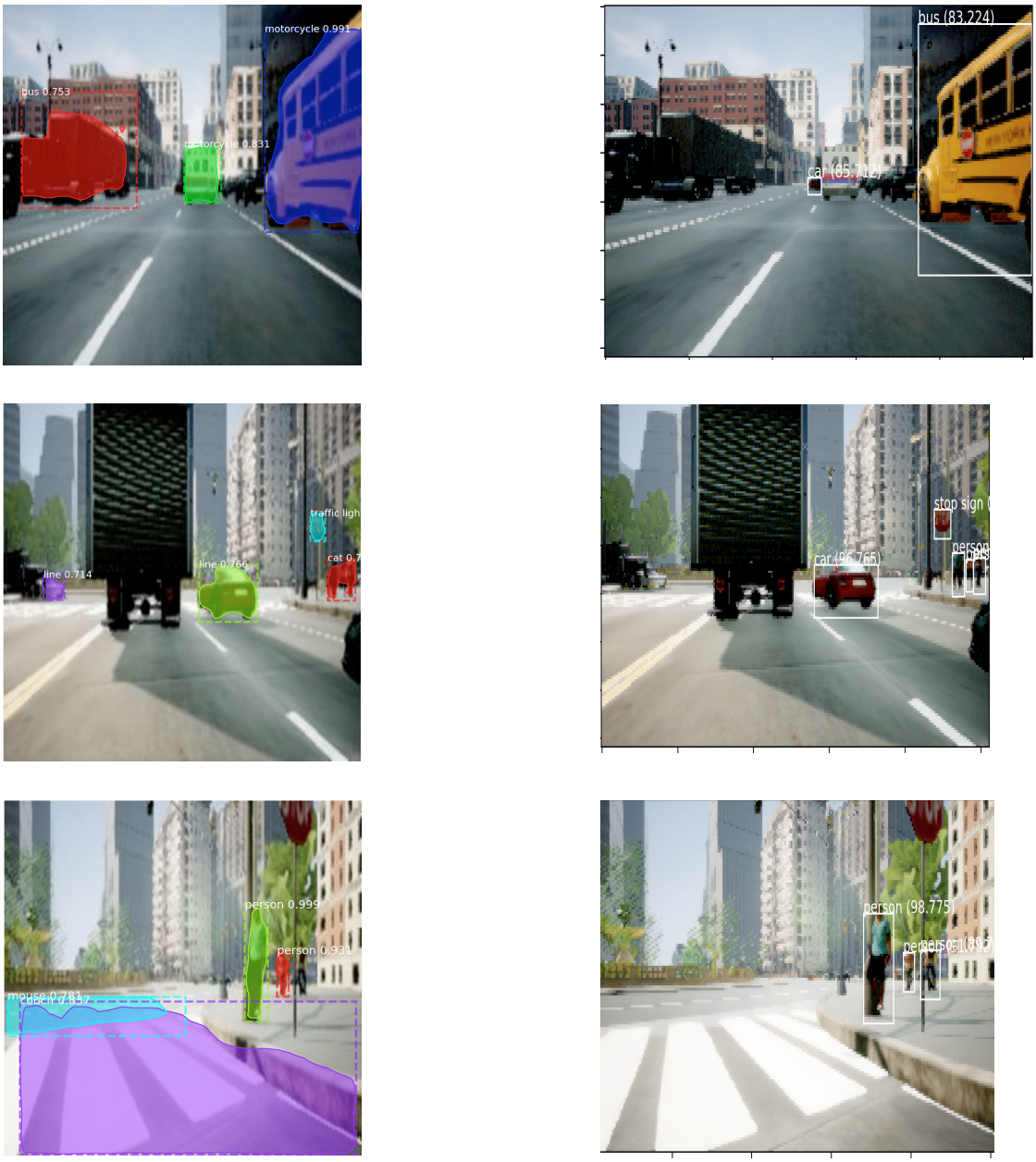


Figure 8: Mask R-CNN and YOLO applied to image logs in AirSim

From the above, we make the following observations of a random sample of five images. The Mask R-CNN prediction time is 20 seconds whereas YOLO performs better at 9 seconds. The bounding boxes are more regressed for certain images in YOLO than Mask R-CNN, and YOLO is far more accurate.

Metric	YOLO v3	Mask R-CNN
Speed	1.2s / image	29.3s / image

Table 1: YOLO v3 vs Mask R-CNN

Imitation learning

Our first attempt to train a self-driving agent to predict the best course of action was based on imitation learning. Imitation learning is a supervised machine learning technique in which a labeled training set is generated by a human controller giving inputs to the system. To create our dataset, we drove the agent around the AirSim City environment, recording images, and an action log. A CNN model is built for the agent to predict the actions by taking the snapshots of images as input in the environment.

To ensure we collected good clean data for imitation learning, we selected the simple task of driving straight in the central lane at a speed close to 7m/s ensuring the car did not drift out of the lane. The recording was stopped for any turn made as this would lead to inconsistencies for the CNN to learn.

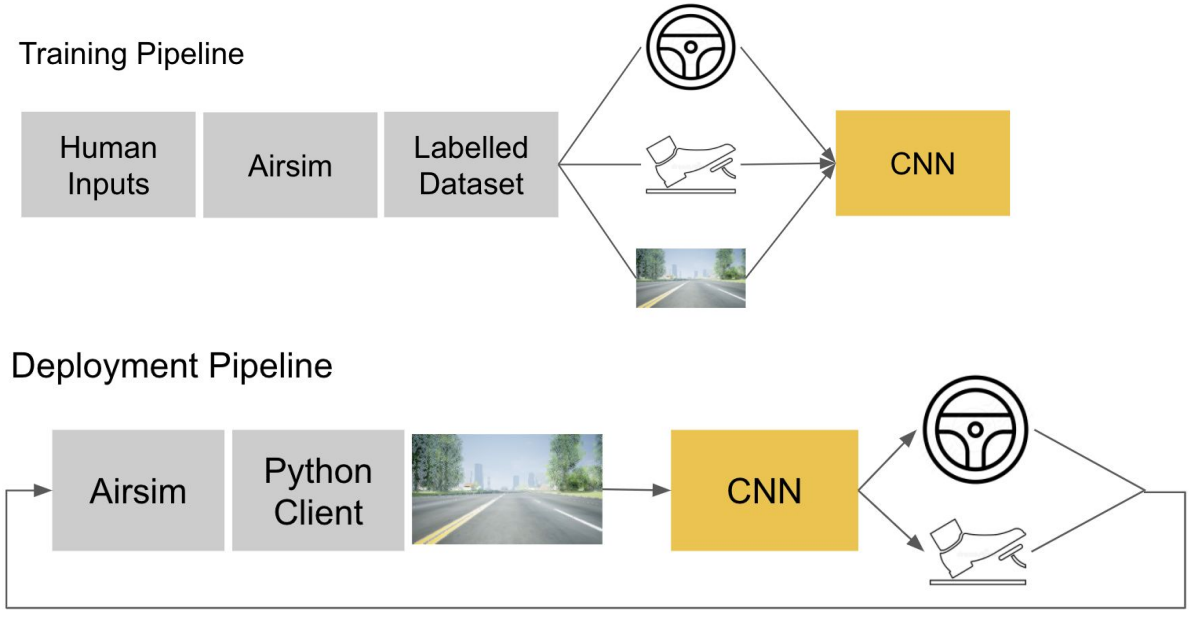


Figure 9: Imitation Learning training and deployment pipelines

Figure 9 shows the pipelines we use to collect our training data, train our CNN, and how it operates in deployment. In our first attempt at imitation learning, model 1, we used the AirSim recommended method, training a CNN for regression, to predict steering angle and acceleration from a given image input. The training dataset consisted of about 5000 images labeled with actions associated with each image. The actions considered for training were: Speed, Steering, Throttle, Brake. In *Figure 10*, the red line is the actual angle and the blue line is the predicted angle, depicting the actual, predicted, and error values.

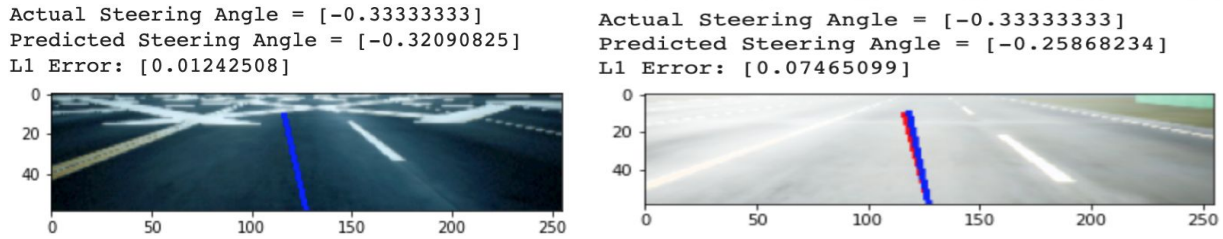


Figure 10: Imitation learning model 1, steering angle and acceleration prediction

Although appearing to accurately predict the steering angles, there is a major flaw with this methodology. Namely, the data we collected for steering is not optimized for a regression objective function. When we drive around in AirSim, as we are using the arrow keys to control the car, the only steering inputs we can use are left, right and straight. In Microsoft AirSim's own setup, instead of using a keyboard, they used a steering wheel giving analog steering angle input data. Essentially we were doing regression on a classification problem. Thus, seeing as our error is fairly low, we can conclude we are most likely overfitting the training data. The results of deploying this method did not have good results with the car frequently veering off course. The correct data collection for this method is to use a steering wheel or another analog controller where the exact steering angle can be recorded, unfortunately, we do not have the hardware required to do this.

The alternative method we implemented mitigates this flaw using a (3 class) softmax objective function to classify steering input into distinct categories. We trained another CNN to classify steering input, this did not work on the first try. An unforeseen problem was that the task of keeping in the lane gave us an unbalanced dataset of 2000 images containing over 10 times more data for straight input compared to left or right steering.

The trained model did not make any turns in the environment predicting straight for each image input. We detailed how we fixed this problem in the technical paper. The results were promising with the agent able to keep in lane consistently. *Figure 11* has the outputs from some images which we fed to our final trained CNN. As we can see the left image outputs a -1 telling the agent to turn left as it is on the right side of the lane, the right image does the same but in the opposite direction. The middle image is closer to the center of the lane and outputs a value of 0

meaning no turn. Thus in the AirSim City environment, it kept within the lane oscillating between the two road markings.

Figure 11: Imitation learning model 2, three class steering classification prediction

Reinforcement Learning:

For training the car to navigate autonomously in a 3D space which can either be a real world or realistic virtual environment, we use Deep Reinforcement Learning. Using this principle, the training of the car happens via rewards and punishments, the car is expected to learn to drive itself maximizing the reward. That means the car is given high rewards for good driving and given punishments for bad driving, in a well constrained reward function the car should exhibit the behavior of good driving.

AirSim's baseline reinforcement learning agent currently uses Deep Q-Learning (DQN), a temporal difference technique of reinforcement learning to build an agent for the autonomous driving car. The neural architecture for AirSim's DQN agent provided by Mnih et al [19] follows that the network takes the images from the car camera as input and transforms them into states that it can then compute Q values for. The output is a set of Q values computed for each possible action. In AirSim's baseline DQN agent, the rewards are calculated on the basis of whether the car stays within the maximum speed limit and how well it stays to the center of the lane. If the car exceeds the speed limit or strays from the center, it receives a negative reward.

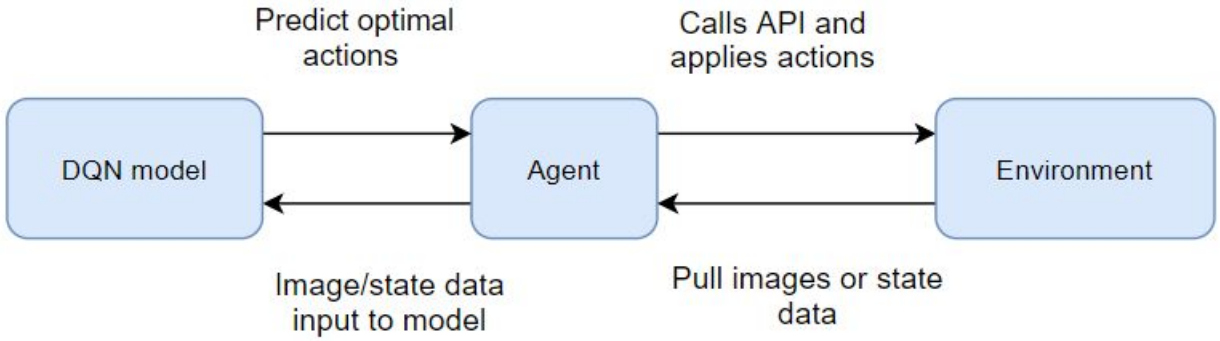


Figure 12: DQN Model interaction in environment

In the following sections we test various forms of DQN learning in pre-built and custom environments utilizing different machine learning frameworks in Python.

a)

b)

c)

d)

Figure 13: a) Neighborhood b) City c) Custom Static d) Custom Pedestrian

DQN Neighborhood Environment Agent

We chose to train this model on AirSim's neighborhood environment, a static environment with houses, streets, fences and parked vehicles. The road is flat with little to no inclination and they are relatively narrow considering it is meant to simulate a neighborhood. Since training a RL model from scratch would take approximately 5-6 days even with a powerful GPU, we decided to use Transfer Learning on top of Microsoft's baseline in Tensorflow. The vehicle passes the state as an image taken from its cameras to the neural network, and the network, in turn, responds to this image with the action that outputs the maximum Q-Value. Since the performance of the car depends mainly on the road and any obstacles it might crash into, we utilized the bottom half of the image, eliminating any irrelevant information like the sky or tops of buildings and trees. This made the training faster and less computationally expensive by reducing the number of parameters in the model.

The performance within the first few hours of training was suboptimal and the car kept crashing into nearby vehicles and houses. After approximately 25 hours of training, the model achieved a loss of 0.026 and the car learnt to drive smoothly on a straight road and able to take turns to avoid crashing into a dead end. With better computational resources, we would be able to train the car to perform better on curved roads and drive for a longer time without crashing.

DQN City Environment Agent

An important objective was observing the performance of our agent in an urban environment. To do this we used the AirSim City environment in which we had previously applied our imitation learning algorithm. This gave us a pre-built simulation environment with roads and obstacles. Unfortunately as this environment is 3rd party, we did not have access to the editor files and could not customize them. This left us with a set start point for the simulation which spawned the car next to a roundabout and so we decided to create an objective to drive around the roundabout.

The model we implemented for the roundabout objective used Microsoft Cognitive Toolkit (CNTK)[20] and translated the coordinates to polar coordinates with the center of the roundabout, the origin. We built a reward function which would allow the agent to navigate the roundabout, this reward function was made up of multiple factors.

- Minimum distance from the roundabout road to the car center. The car was penalized and reset for going over a threshold distance from the roundabout road center.
- The speed of the car was used, a faster speed increases the reward. If the car goes below a minimum speed, it would incur a penalty and the car would be reset.
- Angle between the car direction and tangent of the roundabout at that point.

One difficulty we experienced with training our initial models was that training times were exceptionally long, in the order of days when using vision based inputs for Q value prediction with CNNs. This is a result of running the heavy simulation which uses the GPU alongside the neural network training, which also requires GPU usage. To ameliorate this issue, we decided to reduce the size of our network, instead of using image inputs of 256 x 144 pixels, we replace the input state x and y coordinates, speed and distance from the roundabout circumference, and use fully connected networks with just 1 or two hidden layers. In this way the number of parameters to learn in the neural network was drastically decreased. We realize that although this state input is not representative of real life where a car receives image input, these states can be deduced from sensory input such as LiDAR, imu, gps, road and lane detection to get a coordinate based input.

Another advantage of using much smaller networks and reducing the training time is that hyperparameter tuning is much more efficient. Instead of waiting for days for our model to train, within half an hour we could see how the trained model performed. Some of the important hyperparameters to tune were: learning rate, hidden layer size, episodes between training, minibatch size, epsilon decay (exploration vs exploitation).

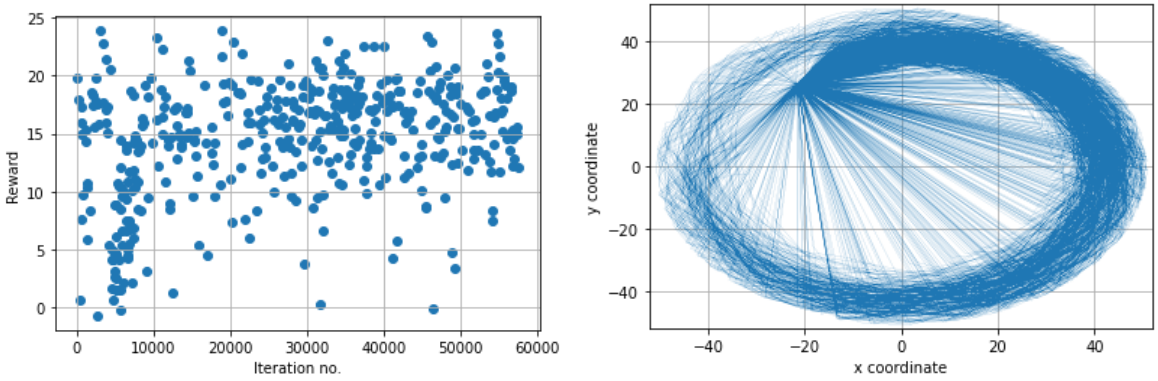


Figure 14 a) Reward function over time *b) Traced path of car*

Figure 14 a shows the reward for each episode. This increases to a maximum around 24. This case would be when the agent is close to the road center, driving above 10 m/s and angled close to the tangent. The average reward initially increased as the number of iterations increased then the agent would be driving round the roundabout. *Figure 14 b* shows the car's path throughout training from start point a -22, 25 going clockwise round the roundabout and being reset if it strays too far from the roundabout road.

Another attempt to use DQN learning in the City environment was to get the car to follow certain roads. Unfortunately we did not see satisfactory results with this methodology as the car struggled to change between road segments before being reset. Initially the car was facing the opposite direction from the road located at the origin and hard coding the car to turn around was

more difficult than planned as the simulator quite often ran at different speeds depending on how long it had been run for and how much the computer was otherwise being used which led to an inability to train for sustained periods of time. *Figure 15* shows the environment mapped out with the roads attempted highlighted in different colors.

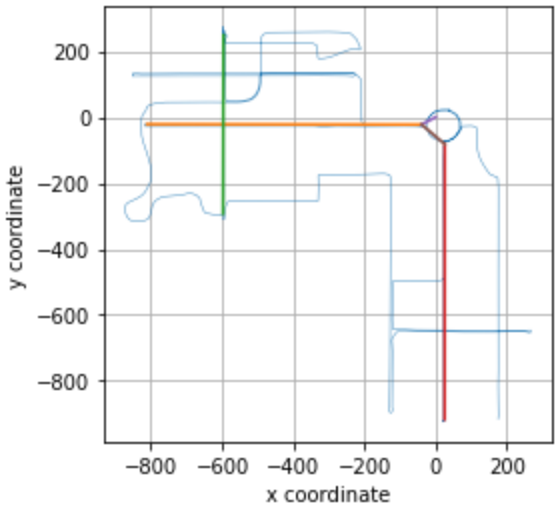


Figure 15: The city map plotted with roads selected for the reward function

DQN Custom Environment Agent

One experiment with RL involved creating custom environments and objectives where we could train our model. We used Unreal Editor 4.24 to create two environments, the first is a boxed landscape with cone objects placed at various locations. The second was another boxed landscape, instead of cones, we introduced moving pedestrians. The idea of these environments was to train an agent to navigate without crashing into the objects.

The models we trained used both OpenAI Gym and OpenAI's Baselines. Gym is a toolkit available for developing and comparing RL algorithms. It is an open-source library which gives access to a set of standardized environments. OpenAI Baselines is a set of high-quality implementations of RL algorithms which will make it easier to replicate, refine and identify new research ideas. This will help us to use the existing approach as a base and add new ideas and serve as a tool to compare both the approaches.

Instead of using the coordinate based approach, we used a vision based approach without the need for large CNNs. Using AirSim's depth perspective Image client, we were able to use a pseudo-LiDAR representation where a distance between car camera and object in each pixel is returned. By taking the minimum distance of objects in the left, center and right of the image, the car can sense if an object is in front of it and also how far away it is from the car.

The reward function of this was purely based on distance with a penalty applied for crashing into any object. After two hours of training the car had seemed to learn to avoid objects quite well from the depth perspective inputs. Initially our model kept on looping in circles, effectively an optimal policy that allows it to infinitely drive with no collision by steering one way or another. With this result we imposed a penalty for an action that repeats more than 10 times in a row. The car stopped driving in circles and did learn to avoid the obstacles well. The model was able to drive for 7 minutes without crashing, which equated to almost 3km.

In the dynamic environment with pedestrians, the pedestrians are set to walk about in random directions. The YOLO closeness metric as defined in the previous section under the name YOLO was used in conjunction with the depth perspective as input, the results can be seen in *Figure 16*. The closest pedestrian in each side of the image by this metric was returned in so that the car would learn to avoid close pedestrians.

Figure 16: Yolo closeness metric in the custom environment with pedestrians

The results for this experiment did not show the agent being able to avoid pedestrians. Quite often when the car is stationary after respawning, a pedestrian collides and the simulation is reset with very little distance covered, the agent cannot possibly learn anything from these false starts. Even after more than two hours of training, the pedestrians were hit quite frequently. The agent struggles to accurately predict the path of pedestrians and relative to the car, their motion is quite fast. Another issue with this environment and model is when a collision is caused by pedestrians walking into the side of the car. It is a near impossible task for the car to evade pedestrians given the only camera view used was forward facing. With a longer training time and more hyperparameter tuning, perhaps we would see further progress, yet we think it would be best to change the parameters of the simulation as well.

Conclusion and Future Targets

The main objective of the project was to train an autonomous driving agent in a simulation environment, tackling tasks faced by the autonomous driving community. This objective was

achieved through a variety of state of the art machine learning methods that allow control in different environments.

Imitation learning gave reasonable performance for driving in the middle lane. Although the agent oscillated between the lane lines, this is representative of the training data fed into the model. Given the dataset on which our model was trained consisted of an hour of simulation driving, as is true for many deep learning tasks, the addition of more training data would naturally improve the model. With access to large datasets of cars driving on the roads in real life, behavioral cloning using a model similar to that presented in this document would be a method capable of achieving lane control.

DQN was used for a variety of objectives with variable levels of success. The performance depended on factors such as model architecture, hyperparameters, simulation parameters and training times to name a few. While our results were encouraging, the performance of all of our DQN efforts fell short of human level behavior as the agent drove for 7 minutes and approximately 3km before colliding. To improve this effort and to use more robust approaches, more powerful GPUs are required which would allow us to leverage computer vision based CNNs to estimate Q values and also machines which allow training for multiple days. This would have allowed us to go beyond coordinate approaches which assume GPS or LiDAR data with set paths mapped out in advance.

One significant area of future work is integrating multiple objectives into one agent. If the agent could detect which scenario applies at a particular time, it would deploy one of the trained models to achieve this objective. For example if the agent following the middle lane came to a roundabout it could switch to the roundabout objective model and navigate that way until it exited. This is necessary to achieve full self driving.

It is important to note that we have only scratched the surface of the potential use of simulation in autonomous driving research. The number of problems encountered in driving that we can replicate is enormous and the scope of our project as it developed was exciting. We hope our solutions aptly reflect the real work done in the autonomous driving research community and that these approaches are transferable from simulation to the real world.

References:

[1] Centers for Disease Control and Prevention. *Global Road Safety*. [online] Available at: <https://www.cdc.gov/injury/features/global-road-safety/index.html> .

[2] Microsoft.github.io. *Home - AirSim*. [online] Available at: <https://microsoft.github.io/AirSim/>.

[3] C. Team, "CARLA", *CARLA Simulator*. [Online]. Available: https://carla.org/.

[4] *Deepdrive.voyage.auto*. [Online]. Available: https://deepdrive.voyage.auto/.

[5]"Duckietown - Learning Autonomy", *Duckietown*. [Online]. Available: https://www.duckietown.org/.

[6] "udacity/self-driving-car-sim", *GitHub*. [Online]. Available: https://github.com/udacity/self-driving-car-sim.

[7] Dean A Pomerleau. Alvinn: An autonomous land vehicle in a neural network. In Advances in neural information processing systems,1989.

[8] Urs Muller, Jan Ben, Eric Cosatto, Beat Flepp, and Yann L Cun. Offroad obstacle avoidance through end-to-end learning. In Advances in neural information processing systems, 2006.

[9] M. Bojarski, D. Testa, D. Dworakowski and B. Firner, "End to End Learning for Self-Driving Cars", 2016. Available: https://images.nvidia.com/content/tegra/automotive/images/2016/solutions/pdf/end-to-end-dl-using-px.pdf . [Accessed 13 October 2020].

[10] J. Koutnik, J. Schmidhuber, and F. Gomez. "Evolving Deep Unsupervised Convolutional Networks for Vision-Based Reinforcement Learning". Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation. 541-548.

[11] S. Bhalla, S. Subramanian and M. Crowley, "Deep Multi Agent Reinforcement Learning for Autonomous Driving", Proceedings from Canadian Conference on Artificial Intelligence, *Springer*, 2020.

[12] X. Liu, Z. Deng and Y. Yang, "Recent progress in semantic image segmentation", *Artificial Intelligence Review*, no. 52, 2017.

[13] "Semantic Segmentation for Self Driving Cars", *Kaggle.com*. [Online]. Available: https://www.kaggle.com/kumaresanmanickavelu/lyft-udacity-challenge.

[14] Z. Zhou, M. Siddiquee, N. Tajbakhsh and J. Liang, "UNet++: A Nested U-Net Architecture for Medical Image Segmentation", *Springer*, 2018.

[15] "Labelbox: The leading training data platform for data labeling", *Labelbox.com*. [Online]. Available: https://labelbox.com/.

[16] Simonyan, K., Zisserman, A, 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition Computer Vision and Pattern Recognition. [online] Available at: <https://arxiv.org/abs/1409.1556>

[17] He, K., Gkioxari, G., Dollár, P. and Girshick, R., 2020. Computer Vision and Pattern Recognition. [online] Available at: <https://arxiv.org/abs/1703.06870>

[18] Redmon, J. and Farhadi, A., 2018. YOLOv3: An Incremental Improvement. *Nature*, [online] Available at: <https://arxiv.org/pdf/1804.02767.pdf>

[19] Mnih, V., Kavukcuoglu, K., Silver, D. and Rusu, A., 2020. Human-level control through deep reinforcement learning. [online] Available at: <https://www-nature-com.libproxy2.usc.edu/articles/nature14236.pdf>

[20] "The Microsoft Cognitive Toolkit" Documentation. [online] Available at : https://docs.microsoft.com/en-us/cognitive-toolkit/