Don't Crash: Autonomous Driving Agent

Abhilash Karpurapu
CS Department, Viterbi
University of Southern California
Los Angeles
karpurap@usc.edu

Haripriya Dharmala
CS Department, Viterbi
University of Southern California
Los Angeles
dharmala@usc.edu

Naga Sanketh Vysyaraju
CS Department, Viterbi
University of Southern California
Los Angeles
vysyaraj@usc.edu

Jaimin Patel
CS Department, Viterbi
University of Southern California
Los Angeles
jaiminpa@usc.edu

Abhivineet Veeraghanta
CS Department, Viterbi
University of Southern California
Los Angeles
veeragha@usc.edu

Shreya Kunchakuri
CS Department, Viterbi
University of Southern California
Los Angeles
kunchaku@usc.edu

***Abstract*—In this paper, we introduce an agent for autonomous driving in a semi-realistic simulation environment, Microsoft AirSim. We outline past work in the field and explore the latest machine learning methods for processing images and controlling the agent. Frameworks in computer vision such as Mask-RCNN, YOLO and U-Net are implemented. Agents using imitation learning through behavioral cloning and reinforcement learning with Deep Q Networks are applied in the environment.A successful implementation of imitation learning is presented for driving straight and keeping in the central lane. Deep Q Networks are used to facilitate driving in 4 separate environment where it optimizes for both following roads and avoiding obstacles.**

***Index Terms*—Autonomous Driving, Imitation Learning, Behavioral Cloning, Deep Q Learning, Self-driving Agent, Object Detection, Semantic Segmentation, Computer Vision**

I. Introduction

The goal of this project is to train an agent that learns to autonomously drive a car. This is an important area ofresearch being studied by many companies, universities, and online communities. The real-world implications of this emerging field are huge, as approximately 1.35 million people die in road crashes each year [1]. Autonomous driving can make roads safer, helping to reduce the number of avoidable accidents by minimizing human error in driving.It will allow those with disabilities and senior citizens to travel independently and safely. It can also reduce the cost of transportation, by removing reliability on human drivers.

Simulation is a key area in self-driving research, and using realistic environments helps algorithm research and development where customized test situations can be forced without the risk of real-world consequences. We have been using Microsoft AirSim [2] for our project, a simulation environment created for autonomous driving research so that multiple objectives can be achieved. Some objectives implemented are: following roads, longest distance and time without crashing, driving with moving pedestrians. We want to build on AirSim's framework, training Computer Vision algorithms for perception and creating agents based on reinforcement learning and imitation learning to perceive and act in these environments.

II. Autonomous Driving Pipeline

The process of achieving autonomous driving includes many different tasks, which can be divided into either perceiving and understanding the scene, or planning and making decisions. We adapt the general autonomous driving pipeline [3] to explain these tasks in further detail in the context of our project.

Fig. 1. Autonomous Driving Pipeline

a) Sense: The first step includes using the car's equipment and sensors to perceive the agent's environment. The car contains cameras in different positions which allowit to identify the road, lanes and objects around it. LiDAR and RADAR sensors can aid with perceiving depth and identifying how fast an object may be moving. They are also especially useful during poor weather conditions or in thedark, where the camera visibility may be impaired.

b) Computer Vision (CV): Once the data has been collected from the car's sensors, the next step is to use itto help the agent make sense of its surroundings. Images taken by cameras can help us accomplish tasks such as lane finding, estimating road curvatures, detecting traffic lights, and recognizing obstacles through object detection and classification. We can use a number of CV algorithms such as RCNNs, Single Shot Detectors and YOLO networks to perform image segmentation and object recognition.

c) Localization and Mapping: The agent must next identify how to figure out its position in the world. This involves mapping the environment and then localizing the current position of the vehicle within the map. This is important because it allows the agent to move around different locations within the environment and also understand whatthe world around it looks like.

d) Path Planning and Policy: The next step after localization is to create a trajectory to travel from one point in the environment to another. In this phase, the agent canalso observe the objects around it and decide what actions they may take, thereby formulating its own action in response. We can test the agent against objectives such as, how long it takes to get from one location to another and how far it can travel within a certain time period.

e) Control: Finally, the agent must navigate the movement of the car by understanding what controls must be applied to achieve the motion it desires. Control is necessary to decide actions such as the speed of the vehicle, steering angle, acceleration and braking.

In any autonomous vehicle project each of these tasks has to be handled. We ensure that our project accounts forall of these tasks by using the simulation environment tools combined with our implementing for these tasks.

III. Related Work

A. Computer Vision

Object recognition is a task which has significantly advanced in recent years due to the rise of neural networks. This has important applications in autonomous vehicles as the car must accurately recognize objects near the it in order forit to take actions. Extending the traditional CNNs, R-CNNs extract a set of object candidate boxes that are rescaled and passed to an ImageNet model, which is further qualified bya linear SVM classifier [4]. This architecture saw a lot of improvement on its predecessor but has an obvious drawback of calculating redundant feature computations in overlapped boxes. This drawback is taken care of by the Fast R-CNN [5] model resulting in faster predictions with comparable accuracy. Although the architectures make attempts to improve accuracy and time to predict, they don't scale as viable solutions for the task of autonomous driving because they are two-stage detectors. These methods conduct the first stage of region proposal generation, followed by the second stage of object classification and bounding box regression. A One-Stage Detector such as You Only Look Once (YOLO) [6] can handle object classification and bounding box regression concurrently without a region proposal stage resulting in faster predictions.

Semantic segmentation is also important for autonomous vehicles. We look at U-Net, a CNN which was originally designed for biomedical image segmentation [7] to train networks with less data than typical CNNs. They use data augmentation effectively and show that the quality of segmentation is high using fewer training samples. The task of segmentation can be directly translated to self driving for identifying surfaces such as roads.

B. Imitation Learning

Imitation Learning (IL) is a supervised machine learning technique in which actions are recorded and a neural network learns to copy these actions. Behavioral cloning is a subset of IL where an agent's behaviour is replicated by a neural network [8]. Using this method, researchers have imitated many behaviors including that of an ant in a unity simulation and a 2d car on a hill.

NVIDIA created an end-to-end learning model using CNNs [10] 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 record what the vehicle sees whenever the steering angle is changed, that also helps to learn and imitate the driver. The CNN isthen trained on these images and steering inputs and when provided with an image can reproduce the optimal action.

C. Reinforcement Learning

Reinforcement Learning (RL) is a subset of machine learning algorithms whereby a computer program learns from experience to improve its performance at a specified task. Machine learning algorithms are often classified under one of three broad categories: supervised learning, unsupervised learning and reinforcement learning. In the RL paradigm, an autonomous agent learns to improve its performance at an assigned task by interacting with its environment. RL requires an environment where state-action pairs can be recovered while modeling dynamics of the vehicle state, environmentas well as the stochasticity in the movement and actions of the environment and agent respectively.

One of the main challenges in RL is managing the trade-off between exploration and exploitation. To maximize the rewards it receives, an agent must exploit its knowledge by selecting actions which are known to result in high rewards. On the other hand, to discover such beneficial actions, it hasto take the risk of trying new actions which may lead to higher/lower rewards than the current best-valued actions for each system state. In other words, the learning agent has to exploit what it already knows in order to obtain rewards, but it also has to explore the unknown in order to make better action selections in the future.

A novel deep reinforcement learning model, Deep Q Networks (DQN), combining RL with deep learning. In summary, the goal of the network is to select a set of actions that maximize cumulative future rewards. Recently, deep

learning and RL based local planners started to emerge as an alternative. Fully convolutional 3D neural networks can generate future paths from sensory inputs such as LiDAR point clouds. Planning a safe path in occluded intersections was achieved in a simulation environment using deep RL. In this, we have explored different ways and approaches of RL.

1) Monte Carlo and Temporal Difference Methods: In the Monte Carlo method, the agent calculates the maximum expected future reward by collecting all the rewards from all states at the end of the game when it goes to any particular terminal state or when it reaches the maximum steps threshold per game. Whereas in Temporal Difference methods, the agent updates the maximum expected future reward at each step. That is, it will update the estimated value at every state by using the information from the experience it had aftertaking an action on that state [11].

2) Model Based and Model Free: In model based the agent tries to understand the environment and create a model of it, it tries to understand and build the transition function that tells the amount of reward the agent gets for different transitions between states and also tries to learn the reward function from the experiences it had through the process of game playing. The agent uses the model as a reference and takes the next steps accordingly. In model free technique, the agent tries to learn the policy from the interactions it had in the game at different states. We could use multiple algorithms like policy gradient and Q-Learning for the agent to learn the policy directly without construction of a sample model [11].

After exploring all the above different methods and approaches to RL, we have narrowed down to a temporaldifference method, Q-Learning associated with neural network(DQN) for this problem.

IV. 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, trade offs between installation requirements, the complexity and features offered were important in 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 challenges. We opted to explore open-world autonomous driving simulators such as Carla, Voyage Deepdrive, Microsoft AirSim, Duckietown, and Udacity's Self-Driving Simulator. After exploring all these environments, we narrowed down to AirSim. More detail on these environments can be found in the Engineering Design Document (EDD).

Microsoft AirSim was the environment we selected, it is an open-source simulator using 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 metadata such as speed, steering angle, position and acceleration. A Python API is also available to control the car autonomously with template environments and algorithms. This API is where we implement our agent for control of the car.

V. Methods

A. Object Detection

1) Mask-RCNN: Mask R-CNN [12] is a progressively improved technique, from R-CNN, Fast R-CNN, Faster R-CNN to Mask-RCNN, which uses either of the backbone architectures to create feature maps over the entire image, ResNets with Faster R-CNN and Feature Pyramid Network (FPN) also with Faster R-CNN, where the latter is observedto perform better.

The whole architecture can be broken down into two segments, bounding-box recognition (classification and regression) and mask prediction, which is applied to every region of interest. The region of interests are also passed toits pool layer and align layers where the images are resized, bi-linear interpolation is applied to get the precise dimensions, the output is later sent to the backbone architecture. Mask- RCNN also has a special layer called Region Proposal Network which scans the FPN and generates two outputs, an anchor class, and bounding box specifications. Mask R-CNN makes pixel-wise dense predictions using masks for all the objects that are predicted in the image. The images in Figure
2 depict the masks created over cars and pedestrians in AirSim environment.

2) You Only Look Once (YOLO): For an autonomous 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 pedestrianto cross. However, if the object is a parked car, the agent is better off driving the car around the parked obstacle.

This is a complex task and can take some time. The YOLO CNN [6] works well for the purpose of object recognition in real-time. It can look at an image only once and recognize objects with high speed and accuracy. The way this works is that the input image is split into a gridof cells. For every object in the image, there is a grid cell

Fig. 2. Image Segmentation

within which the center of the object is located, that grid cell is responsible for the prediction of the object. Each grid cell then predicts bounding boxes by using the (x,y) coordinate of the box center, the height, width, and confidence.

The YOLO v3 architecture [6] uses Darknet-53 as the backbone and includes 53 layers. This architecture also improves from previous versions of YOLO by solving the vanishing gradient problem through skip connections. Vanishing gradient occurs in deep neural networks when the change in the gradient is so small that the weights are no longer updated and the training effectively stops. With skip connections, the output of one layer is propagated to the next few layers instead of just one. Then during backpropagation, the value of the gradient is maintained in these previous layers. This is because we are not multiplying the gradient with a fractional value at each and every layer, thereby preventing the gradient from becoming very small as we approach the earlier layers.

The grid cell also outputs the class probabilities to classify an object once it is detected. This probability is given as $P(\frac{Class(i)}{})$, i.e. given that an object is present, it gives us the probability that it belongs to class i.

Along with object recognition, we have also created a metric to perceive relative distance from the camera. We calculate this closeness as (1), 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.

$$c = k\frac{(Area)}{(Distance)} \quad (1)$$

To avoid obtaining a very large closeness value for boxes with large areas, we multiply with a constant k which lies between (0,1). So we have chosen the value of k to be 0.1 to

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

Fig. 3. YOLO V3 Architecture

normalize the value of closeness.

This information can be useful for the car to determine which object is more likely to be collided into. We use this YOLO model to modify the reward function of our RL agent to consider closeness to neighboring obstacles so the agent minimizes its chances of crashing.

Figure 4 depicts the bounding boxes and classification done by the YOLO v3 model. The lines in green are the perpendiculars which help determine how close an object isto the car.

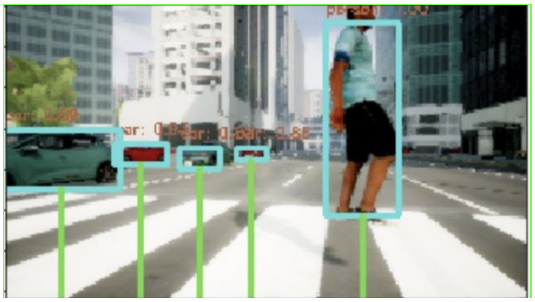

Fig. 4. Classification with Bounding Boxes - YOLO V3

B. Semantic Segmentation

We investigate how we can tackle the task of road segmentation with U-Net [7]. U-Net is named due to its typical representation in architectural diagrams in the shape of a U. There are two paths to the U-Net, contraction and expansion. The first (contraction) utilizes an encoder which is a stack

of convolutional and max pooling layers and the second (expansion) is an expanding path decoder which enables precise localization with transposed convolutions. It is an end-to-end

fully convolutional network (FCN). This network architectureis detailed in figure 5.

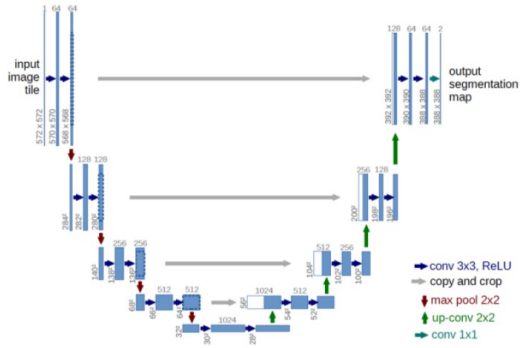

Fig. 5. U-Net architecture

We used a pre-trained U-Net model for road segmentation on a dataset of Carla images on Kaggle [9]. We replaced the last fully connected layer of the network and freezing the rest of the layers by using Keras, we were able to usewhat it had already learnt and retrain the network on 60 images that we hand labelled from AirSim. We retrained the network for 100 epochs with a batch size of 4 and used early stopping to ensure we are not overfitting our small training data sample. For a comparison between the original pre-trained network performance and that after transfer learning, refer to the semantic segmentation section in the EDD.

C. Imitation Learning

IL will mimic the player's driving patterns using a supervised model. The player's data is recorded from the AirSim environment and the corresponding images paired with actions are stored for a stack of 5000 images. We have built two models with different architectures and each modelis explained in detail below:

1) Steering Angle Prediction Model (Model 1):

- Pre-processing: The image is resized, and the pixel values in them are normalized between 0 and 1. The brightness level is also set at 0.4 for the entire image.
- CNN Model in figure 6. The network is trained for 500 epochs, batch size 32, with a learning rate of 0.0001, using a Nadam optimizer with a beta value of 0.9.
- Predictions: The predicted values from the model are approximated to +1,-1 and 0. These values are used to control the agent through the python API, for the specific snapshot image taken when the agent interacts with the environment.

2) Steering Classification (Model 2): The alternative method we implemented has a 3 class softmax objective function to classify steering input into distinct categories. We trained another CNN to classify steering input, this did

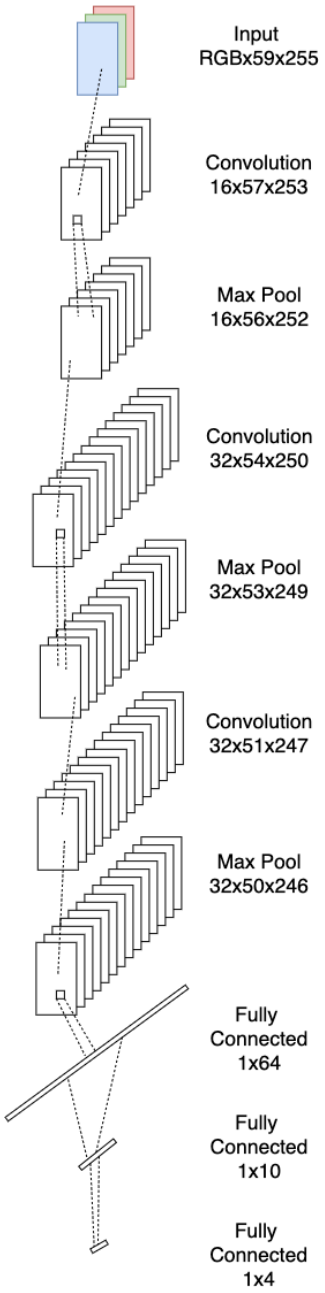

Fig. 6. Model 1: Steering Angle Prediction Regression Model

not work on the first try. An unforeseen problem was that the task of keeping in lane gave us an unbalanced dataset of 2000 images that has over 10 times more data for straightinput compared to left or right steering.

Our initial effort resulted in the CNN, shown in figure 7 finding a local minimum, predicting straight for every example. Although achieved a high accuracy of over 0.9, the car did not make any turns in the environment. We investigated ways to fix this problem, we could either imposea greater penalty for incorrectly predicting left or right with anasymmetric loss function or we can undersample the number of straight examples so we have a balanced dataset. We opted for the latter and used data augmentation, normal procedures of flipping around axes, rotating, translating and scaling would all randomly generate brightness samples as shown in figure8 to increase the amount of training data from 900 to 9000 images.

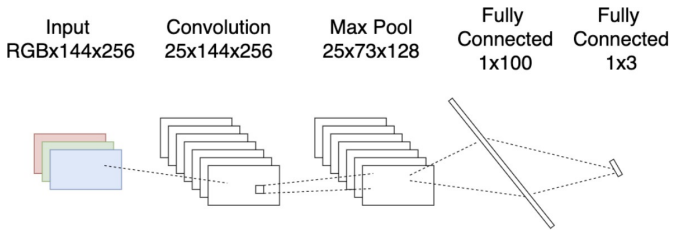

Fig. 7. Steering Angle Classification Architecture

After running this model, we found that it was able to keep in lane when the road surface had two distinct lane lines. When the agent crossed intersections, the learnt pattern on the road was obscured and the agent frequently went out of lane and crashed. In order to correct this problem we collected more training data from the dataset we had originally undersampled from by running the simulation with human input. When the CNN was trained with this extra data, it was able to go over intersections consistently.

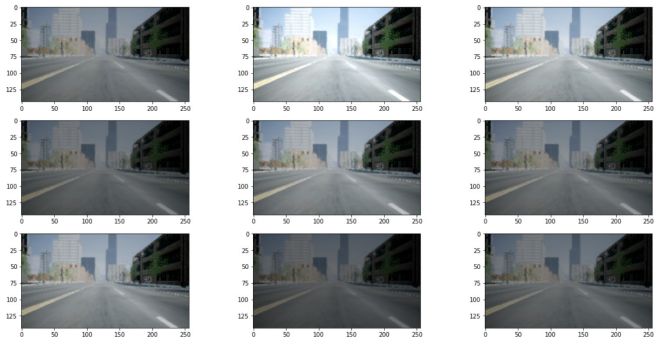

Fig. 8. Brightness Augmentation

D. Reinforcement Learning

Deep RL is used to train a car to navigate a 3D space in realistic virtual environments autonomously. 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. This means the car is given high rewards for good driving and and given punishments for bad driving in a well constrained reward function the car should exhibit the behavior of good driving. We adopt DQN Mnih [13], a deep RL method, originally from DeepMind's experiments which achieved superior results to human performance at Atari games. As we are working on AirSim's environment, it contains a vision based baseline RL agent using DQN to control the car.

There are many ways to update Q-values in the Q-learning algorithm. In our scenario, building an agent for an autonomous driving car, where the input data with respect to each state of the environment is neither simple nor limited.We take input data in the form of a set of frames as eachstate can be described by an image (frame) from the camera at any particular point of time in the simulation. Therefore, using value-iteration to update the Q-values is an expensive process due to size and form of the input state. This leads usto the option where we update Q-values using deep learningto develop a DQN. The steps of the algorithm are as follows:

- In the first step, we initialize the replay memory andset its capacity. We now expand upon the concepts of experience replay and replay memory which are used during the training process of DQN. During the training process, we store the agent's experiences at each timestep and store in a memory called replay memory. At each time step, we store a tuple of values, which includes the current state of the environment, the action that was taken in that state, and the next state of the environment along with reward given to the agent for this action. This gives us a summary of the agent's experience atthat time step. Since we cannot store all the experiencesat every time step we set a size N. Every time a new experience is observed we update our replay memory by replacing this with the oldest experience in the memory. So we only store the last N experiences. The process of gaining experiences and later sampling these from replay memory for training is called experience replay.

- Next, we initialize the weights of the neural network similar to a regular one that is used to approximate a function. And we start the game (also called an episode) by initializing its starting state.

- At each time step in the episode, we select an action and execute it in a simulator. There are two ways to choose an action at any time step, we can either take a random action or take the best step considering the

information we have from the environment. In the initial time steps, we don't have much information about the environment so it is okay to choose random action whereas in the later stages we have more information about the environment compared to the initial stages soit is not preferable to take random actions. Since it is a little ambiguous as to which action is optimal at eachstep, we employ *Epsilon greedy strategy*.

- Here, Epsilon is a floating-point value that ranges from0 to 1. We initialize it to some value within its range by restricting it with a min and max value. Then we choosea random number which also lies between (0,1). If the chosen value is less than the epsilon value, then we choose a random action (*exploration*), if it is more than

the epsilon value then we credit the present information and choose an action based on that (*exploitation*). As we run the simulation multiple times or pass through

multiple episodes for training, we update the epsilon value using the epsilon decay rate. Initially, the epsilon value is close to 1, so the percentage of random actions will be higher than calculated actions. As the training goes on, the epsilon decays, the percentage of random actions decreases, and the percentage of informed actions increases. This is called exploration versus exploitation.

- Now, the chosen action in the previous step is executed and then we observe the next state along with the reward and store the experience in replay memory.

- In the next step, we sample a random batch from the replay memory, preprocess it, and feed it to the neural network also called a policy network as it updates the Q-values (2) to find the optimal policy. We should note that we take a random sample and not a list of sequential experiences to avoid the high correlation between the experiences which will lead to very inefficient learning. This is also one of the main reasons for using replay memory.

learned value

old value (2)

Where,
α - Learning rate
γ - Discount rate
S - Current State
S^1 - New State

cropping, scaling, etc depending on the input data weuse. Then we give this to the network, the input is a state in the form a stack of frames instead of a singleone because it gives context to the environment whenthe objects or the agent in it is moving.

- Then for each state-input, the network will evaluate and give out the Q-values for all possible actions on that state.As in a normal deep learning network, we calculate the target Q-value for that state using the Bellman equation
(3) and get the loss. We do a second pass to the policy network on the next state to get Q-value for all the possible actions of that state and then get the max-valueto calculate the target Q-value.

$$Q(s,a) = E\left[R_{t+1} + \gamma \max q_*(s^1, a^1)\right] \quad (3)$$

- After obtaining the loss value, we use gradient descent and back propagation to update the weights and reduce the loss, we repeat the process for each time step of an episode until we reach a final state. In this way, we train the network with a set number of episodes before getting the final matrix of Q-values for the self-driving agent [14].

For the autonomous agent provided by AirSim, the neural architecture follows that provided by Mnih et al [15]. It contains 3 convolutional layers and 3 max pooling layers and2 dense layers with rectified linear unit (Relu) activation to output the Q values for each possible action as show in in figure 9. The network takes the images from the car cameraas input and transforms them into states that it can then compute Q values for. In AirSim's baseline Deep Q learning agent, the rewards are calculated on the basis of speed and distance from set paths. If the car exceeds the maximum speed limit or strays from the center, it receives a negative reward.

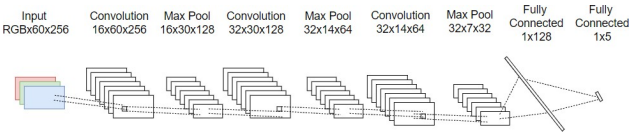

Fig. 9. AirSim DQN CNN architecture

1) DQN Neighborhood Environment Agent: Training a

R - Reward received
a - Action taken
a^1 - Action on new state

- Before feeding the data to the network we preprocess,if the data is image data, we use gray scale conversion,

deep RL model from scratch is a resource intensive task which requires powerful computational resources and time, especially when dealing with an environment with thousands of possible states and actions. We decided to create a baseline for a RL agent by applying transfer learning to the model established by Microsoft's Autonomous Driving Cookbook. The network

model is a three layer Convolutional Neural Network that is closely modeled after Google DeepMind's Deep Q-Learning 13. The reward function is very simple and takes into accountthe position of the car with respect to the center of the nearest road. It is designed to make the car learn to drive on the road instead of veering off-lane, a task which is fundamentalto autonomous driving. It obtains the car position from the AirSim API and awards the vehicle a higher reward for driving within a threshold distance from the center of the road and penalizes it for colliding into obstacles. The distance reward is calculated by using an exponential function which considers the product of the distance from the road and the decay rate, which determines the rate at which the reward decays for the distance. For this reason, it is important to note that all the rewards are positive and range between [0,1].

$$distanceReward = e^{-(distance \ . \ decayRate)} \quad (4)$$

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 take turns to avoid crashing into a dead end. Figure 10 shows the loss decreasing during training. 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.

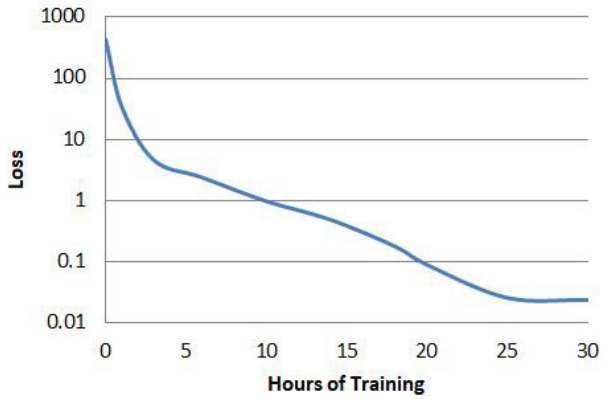

Fig. 10. Change in loss of model while training

2) 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 IL 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. The model we implemented was to drive round the roundabout without crashing. For this objective, Microsoft Cognitive Toolkit (CNTK) was used, transforming coordinates to polar coordinates with the center of the roundabout as the origin. We built a reward function which would allow the agent to navigate the roundabout, this reward function was made upof 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 and if the car goes below a minimum speed, it would incur a penalty and the car would be reset.
- The angle between the car direction and tangent of the roundabout at this point which rewards the agent if it is steering in the correct direction.

The epsilon greedy decay strategy was optimized, initially using a linearly decreasing function to move from a fully exploring agent to a fully exploiting agent. Employing an exponentially decreasing strategy produced more satisfactory results. The reason we suspect this was the case as when the agent is purely exploring, the mean result is driving straight as there is an equal probability of turning left or right. Thisby nature of the roundabout curve means the model learns simply to turn left regardless of input state. As the model reduces the proportion of exploration actions, it begins to turn left (to go round the roundabout). The model would eventually turn left too far and reset because it would stray too far from the circumference. The exponentially decaying strategy slows the rate of learning the longer the algorithmhas been running, so the agent is exploring for more iterations when it is already turning too far left. A replay buffer was also used in this method so the model could sample the last 50000 iterations and select a minibatch to relearn from previous states it had seen.

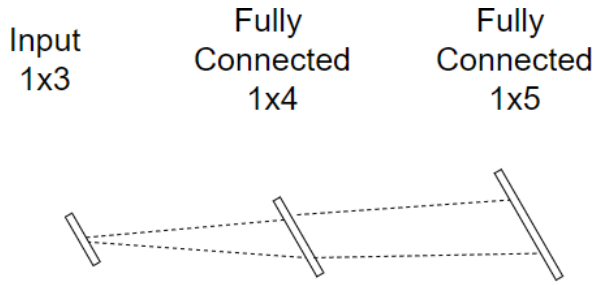

Fig. 11. DQN roundabout CNN architecture

A very simple architecture was used for this model consisting of 1 hidden layer in 3 dense layers. This was used as the model inputs were 3 floating point numbers rather than an image and the output was steering actions with different

accelerations. This allowed us to iterate training quickly and tune hyperparameters to achieve a working model.

Using this strategy the model we trained after 2 hours, 60000 iterations and 453 episodes, the agent could navigate the roundabout reasonably well without being reset. The driving however was quite jerky, this issue could be dealtwith by applying a reward for smooth driving.

3) 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 landscapewith cone objects placed at various locations. The second was another boxed landscape, instead of cones, with 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 [16] and OpenAI Baselines [17]. Gym is a toolkit available for developing and comparing RL algorithms. The reward function of this was purely based on distance with a penalty applied for crashing into any objects. We used a multi-layer perceptron architecture provided by the Deep q agent in OpenAI Baselines with 64 hidden layers, the model architectures can be seen in Figure 12. After two hours of training in the static environment, the car seemed to learn to avoid objects quite well from the depth perspective inputs. Initially our model kept on looping in circles effectively an optimal solution which allows it to infinitely drive with no collision by steering one way. With this result unsatisfactory, we imposed a penalty for the same repeated action more than 10 times in a row. The car stopped driving in circles and did learn to avoid the obstacles well.

The next experiment involved training the model in the environment with moving pedestrians and applying the YOLOcloseness metric on each side of the image as an input. 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, additional camera views and more hyperparameter tuning, perhaps we wouldsee further progress with this objective.

The models used and cumulative reward function is shown in figures 12 and 13. As we can see for the static environment the reward function seems to be increasing with iteration

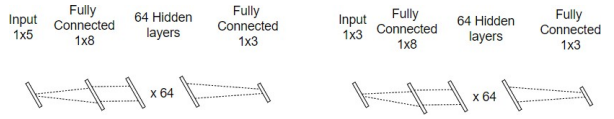

Fig. 12. DQN multi layer perceptron architectures for static and pedestrian models

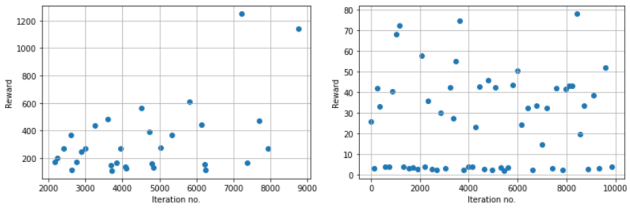

Fig. 13. Cumulative reward for static and pedestrian environments

however this effect is not clear in the pedestrian environment.

E. Imitation Learning Vs Reinforcement Learning

Both IL and RL are methods which can be applied for self driving objectives. In RL, the agent learns online while acting through trial and error. It tries different actions to see which one yields the best rewards and uses that as a basis to improve its policy. In IL, the agent learns from demonstration in a supervised way. Although there are no rewards, it learns from the recorded input of someone driving and creates a policy to imitate those actions. In this sense, a RL agent could theoretically achieve optimal results with the task. An IL agentmay not always be optimal, it can only become as good as the human which teaches it the task. It can also emulate different styles of driving depending on who it is learning from. One advantage of IL is that the learning can be done offline as CNN can be trained by we did in a Python notebook, RL however must be done in an environment where the state and actions are to be simulated. We suspect that to achieve full autonomous driving, there are aspects of both the methods which will be useful for an agent.

VI. Results

We have successfully implemented computer vision, IL and RL in our simulation environment, AirSim. Object detection was performed, comparing two computer vision techniques, Mask R-CNN and YOLO with YOLO performing better to provide bounding boxes identifying cars and pedestrians. Using a pre-trained U-Net architecture, transfer learning allowed us to produce a reasonable segmentation mask of the road in front of the car. Building on successive IL models, both using behavioral cloning, we implemented our first objective, getting the car to drive straight in the middle lane. Additional tasks were tackled using RL, specifically DQN with varying architectures allowing the agent to learn to navigate for different objectives such as driving in a neighborhood, around a roundabout and avoiding obstacles.

To see more results and demonstration of the outputs from models described in this paper refer to the Methods sectionin the EDD.

VII. 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 which allow control in different environments.

IL 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 which our model was trained on consistedof 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 paper wouldbe a method capable of achieving lane control.

DQN was used for a 4 objectives with variable levels of success. The performance depended on factors such as model architecture, hyperparameters, simulation parameters, training times to name a few. While our results were encouraging, the performance of all of our DQN efforts fall short of human level behavior. To improve this effort and to use more robust approaches, more powerful GPUs are required with machines which allow training for multiple days. This would have allowed us to go beyond coordinate based approaches which assume GPS or LiDAR data with set paths mapped out in advance.

Future work to build on the foundations laid in this paper would be to integrate multiple objectives into one agent. Ifthe agent could detect which scenario applies at a particular time, it would deploy one of the trained models to achievethis 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. In addition, path planning, an important part of the autonomous driving pipeline could be tackled with path planning algorithms in conjunction with the model switching approach.

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. Road Traffic Injuries and Deaths—A Global Problem, Centers for Disease Control and Prevention (CDC), URL : https://www.cdc.gov/injury/features/global-road-safety/index.html
2. Microsoft AirSim, URL: https://github.com/microsoft/AirSim
3. B, R., Sobh, I., Talpaert, V., Mannion, P., Al Sallab, A., Yogamani, S. and Pérez, P., 2020. Deep Reinforcement Learning for Autonomous Driving: A Survey
4. Zhengxia Zou, Zhenwei Shi, Yuhong Guo, Jieping Ye, Object Detection in 20 Years: A Survey, Computer Vision and Pattern Recognition 2019
5. Ross Girshick, Fast R-CNN, Computer Vision and Pattern Recognition 2015
6. Jiwoong Choi, Dayoung Chun, Hyun Kim, Hyuk-Jae Lee, Gaussian YOLOv3: An Accurate and Fast Object Detector Using Localization Uncertainty for Autonomous Driving, Computer Vision and Pattern Recognition 2019
7. Olaf Ronneberger,Philipp Fischer,Thomas Brox, U-Net: Convolutional Networks for Biomedical Image Segmentation, Medical Image Comput-ing and Computer-Assisted Intervention – MICCAI 2015.
8. Farax Torabi, Garrett Warnell, Peter Stone, Behavioral Cloning from Observation. International Joint Conference on Artificial Intelligence 2018.
9. Kaggle Segmentation Dataset, URL : https://www.kaggle.com/kumaresanmanickavelu/lyft-udacity-challenge
10. Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, End to End Learning for Self-Driving Car, NVIDIA Corporation
11. Thomas Simonini, An Introduction to Reinforcement Learning URL: https://www.freecodecamp.org/news/an-introduction-to-reinforcement-learning-4339519de419/
12. Kaiming He, Georgia Gkioxari, Piotr Dollár, Ross Girshick, Mask R-CNN, International Conference on Computer Vision 2017
13. Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller, Playing Atari with Deep Reinforcement Learning, NIPS Deep Learning Workshop 2013
14. Reinforcement Learning - Goal Oriented Intelligence byDeeplizard, URL : https://deeplizard.com/learn/playlist/PLZbbT5os2xoWNVdDudn51XM8lOuZ_Njv
15. Mnih, V., Kavukcuoglu, K., Silver, D. and Rusu, A, Human-level control through deep reinforcement learning 2020
16. OpenAI Gym URL : https://gym.openai.com/
17. OpenAI Baselines URL : https://github.com/openai/baselines