Teach An Agent To Drive A Car In A Virtual Environment

Anant Vignesh Mahadhevan Rakesh Ramesh Dewang Shah

**Abstract-** It is highly evident that autonomous vehicles will be the future and it will be a prominent vehicle category in the next decade. For this to be a success, the vehicle should be safe, reliable and provide a comfortable user experience. Autonomous driving must have sophisticated negotiating skills while taking right, left turns and while pushing ahead in urban areas. Reinforcement learning is considered as the main domain for learning driving policy. We propose a reinforcement learning approach using deep Q-learning approach which will extract the maximum reward from a large state space. We use CARLA, an open-source simulator for autonomous driving research. The outcome of this experiment is to resemble a real-life environment where the agent tries to overcome the obstacles using the data from the virtual sensors attached to the agent.

**Introduction-** A self-driving car, also known as an autonomous car, driverless car,or a robotic car is a vehicle that is capable of sensing its environment and moving safely with little or no human input. Self-driving cars combine a variety of sensors to perceive their surroundings such as radar, lidar, sonar, GPS, odometry and inertial measurement units. Today, a driverless car provides partly automated features such as keeping the car within its lane, speed controls or emergency braking.

Currently, all the leading automobile manufacturers in the world are performing some kind of research and development involving autonomous driving or more fancily called as the “Auto-Pilot” in cars and trucks. Most of the development of the “Auto-Pilot” feature in automobiles is trained by combining multiple methodologies to create a functioning autonomous vehicle faster, in order to keep up with the rapidly changing market trends. This approach makes the process of creating driverless cars more complex and expensive. In recent work, an end-to-end supervised learning approach that maps the front-facing camera images of a car to steering angles gives expert data. However, they require a large amount of data from different possible driving scenarios in order to give a good approximation of the policy. Still, they might fail when facing scenarios that are very different from the ones in the training data.

The biggest disadvantage of completely automating the process of training a vehicle in a virtual environment is the absence of an environment that mimics the unpredictive nature of the real-world road scenarios. Even though some open source environments such as TORCS exist, they are merely racing track environments that does not mimic road traffic scenarios.

In order to overcome the above-mentioned disadvantages of the existing methods, this paper proposes a completely automated training approach using reinforcement learning on an open-source environment called CARLA that mimics real-world road traffic scenarios much better than any other open-source environments. The main idea of CARLA is to have the environment as the server and the vehicles/agents as the client and train the vehicle to drive itself by using reinforcement learning algorithms. One of the major advantages of using CARLA is that its environment is highly customizable and can be used to recreate any kind of real-world scenarios such as climate, lighting, road conditions, traffic congestion, pedestrians and other possible obstacles.

**Literature Review-** In one of the papers[2], the authors use the DQN to control a simulated car via reinforcement learning. This paper takes a very similar approach to ours where they use a simulated environment to train an agent without a pre-fed dataset. But again this paper uses a Javascript racer game to train their model and completely avoids real-world traffic problems where there can be n number of factors affecting the driving pattern of the agent.

One of the most popular instances of autonomous systems is self-driving cars. The main objective of an autonomous car is to drive people around the streets without compromising human comfort and most importantly there should be no input from the user. The authors of [7] propose an approach to building a training policy in a simulated environment with Inverse Reinforcement Learning and deep Q-network to extract the rewards in the problem with large state space. In this paper, the author addresses the exploding state-space problem and builds upon the IRL based method using DQN.

In [8], the authors apply deep reinforcement learning to the problem of forming long term driving strategies. To solve the two major challenges that make autonomous driving different viz, ensuring functional safety and the Markov Decision Process model causing unpredictable behavior of other agents in a multi-agent scenario, the authors show the use of policy gradient iterations, decompose the problem into a composition of a Policy for Desires and introduce a hierarchical temporal abstraction that they call an “Option Graph”, thereby reducing sample complexity, while also playing a similar role to LSTM gating mechanisms used in supervised deep networks.

In [9], the authors create a “compressed” network encoding where network weight matrices are represented indirectly as a set of Fourier-type coefficients, to tasks that require very-large networks due to the high-dimensionality of their input space. This helped determine how to create large-scale models efficiently. Their environment, “TORCS”, is a 2-d environment, but since they use video images, and retain information from previous images in order to compute the velocity of the car, it helped to research using our sensor based approach. Their dimensionality reduction of the search space, a problem with the Monte Carlo method, helps us create an efficient model.

In this paper[1], the authors propose a novel realistic translation network to make model trained in virtual environment be workable in the real world. The proposed network can convert non-realistic virtual image input into a realistic one with similar scene structure. Given realistic frames as input, driving policy trained by reinforcement learning can nicely adapt to real world driving. Here the authors use a conventional RL solver Asynchronous Advantage Actor-Critic (A3C) to train the self driving vehicle, which has performed well on various machine learning tasks. This model performs better than the domain randomization method, which requires training in multiple environments to generalize.

In this paper[10], the authors propose a novel autonomous driving paradigm based on direct perception.This method maps an input image to a small number of key perception indicators that directly relate to the affordance of a road/traffic state for driving . Their representation provides a set of compact yet complete descriptions of the scene to enable a simple controller to drive autonomously. They train a deep Convolutional Neural Network using a recording from 12 hours of human driving in a video game and show that this model can work well to drive a car in a very diverse set of virtual environments. They also train a model for car distance estimation on the KITTI dataset. Results show that this approach direct perception approach can generalize well to real driving images and can perform well in both virtual and real environments.

In another paper[11], the authors adapt an actor-critic, model-free algorithm called Deep DPG (DDPG) based on the deterministic policy gradient that can operate over continuous action spaces. Using the same learning algorithm, network architecture and hyper-parameters, this algorithm robustly solves more than 20 simulated physics tasks, including classic problems such as cartpole swing-up, dexterous manipulation, legged locomotion and car driving. The authors further demonstrate that for many of these tasks, the algorithm can learn policies “end-to-end”: directly from raw pixel inputs. Interestingly, all of these experiments used substantially fewer steps of experience, a factor of 20 fewer steps than was used by DQN learning to find solutions in the Atari domain. This suggests that, given more simulation time, DDPG may solve even more difficult problems than those considered here.

In this paper[12], the author collected a large data set of highway data by using Camera, Lidar, Radar, and GPS consisting of 17 thousand image frames with vehicle bounding boxes and over 616 thousand image frames with lane annotations and apply deep learning and computer vision algorithms to problems such as car and lane detection. The authors then trained on this data using a CNN architecture capable of detecting all lanes and cars in a single forward pass. The authors show how existing convolutional neural networks (CNNs) can be used to perform lane and vehicle detection while running at frame rates required for a real-time system. Results show existing CNN algorithms are capable of good performance in highway lane and vehicle detection.

In this paper[5], the authors present an approach in which supervised learning is first used to estimate depths from single monocular images. The learning algorithm can be trained either on real camera images labeled with ground-truth distances to the closest obstacles, or on a training set consisting of synthetic graphics images. Reinforcement learning/policy search is then applied within a simulator that renders synthetic scenes. The authors model the RC car control problem as a Markov decision process (MDP). This learns a control policy that selects a steering direction as a function of the vision system’s output. The experiments with the graphical simulator show that model-based RL holds great promise even in settings involving complex environments and complex perception.

In this paper [4], the author proposes a learning from demonstration approach that allows the user to simply demonstrate the desired style by driving the car manually. The author modelled the individual style in terms of a cost function and use feature-based inverse reinforcement learning to find the model parameters that fit the observed style best. Once the model has been learned, it can be used to efficiently compute trajectories for the vehicle in autonomous mode. This approach is capable of learning cost functions and reproducing different driving styles using data from real drivers.

In [15], the author proposes an efficient strategy to navigate safely through unsignaled intersections. The author was able to learn policies that surpass the performance of a commonly-used heuristic approach in several metrics including task completion time and goal success rate and have limited ability to generalize. The author was able to develop a system that uses Deep Q-Networks for the specific problem of intersection handling and show that it is capable of learning exploratory behaviors to more fully understand the scene.

In this paper [16], the author proposes a method to leverage the predictions related with other cars from the environment during autonomous driving to plan more efficient and communicative behaviors. The author models these consequences by approximating the human as an optimal planner, with a reward function that we acquire through Inverse Reinforcement Learning. The author was able to use the robot which is indeed capable of eliciting desired changes in human state by planning using this dynamical system.

In [3], the author presents deep reinforcement learning autonomous navigation and obstacle avoidance of self driving cars, applied with deep Q-networks to a simulated car an urban environment. This approach uses camera sensor and laser sensor in front of the car. The author uses a fully-connected convolutional neural network and feed the neural network with 4 consecutive frames of the camera. The author was capable of driving the car autonomously by taking proper decisions in the environment.

In the paper[13], propose an inverse reinforcement learning approach using Deep Q-learning networks to extract the rewards in problems with large state space. The author evaluate the performance of this approach in a simulation based autonomous driving scenario.. The author uses Markov Decision Process. Th networks output were to steer left, right and not steer. The author was able to do a collision free motions after few learning rounds.

In [14], the authors, after successfully being able to play an Atari game, tried to use sensory inputs, and scale up their approach by extending to driving a car autonomously, in a 3d simulation environment. They compare CNN, RNN and Hybrid CNN-RNN based deep Q-learning models to train the agent to drive around a race track. They use a low-dimensional discrete state space to be able to stably control the car. Their work did not involve a real world scenario, like a street where there are for more rules and obstacles as opposed to a single car race track, where the agent’s speed is the most important factor.

In [17], the authors present a simple model inspired by cognitive systems, and train the RNN-based agent, using the backpropogation algorithm, since it can be used to train large neural networks efficiently. The authors give the agent a visual sensory component that compresses what it sees into a small representative code. It also has a memory component that makes predictions about future codes based on historical information, and a decision-making component that decides what actions to take based only on the representations created by its vision and memory components. The environment provides the agent with a high dimensional input observation at each time step. This input is usually a 2D image frame that is part of a video sequence. The role of the model is to learn an abstract, compressed representation of each observed input frame. The authors apply this model to train an agent to solve a car racing task, and a small VizDoom environment, mimicking the Doom environment.

In [18], the authors tackle the challenge of efficiently learning a mapping from pixels to an appropriate representation for control using only a sparse reward signal. They say that though deep convolutional encoders can learn good representations (upon which a policy can be trained), they require large amounts of training data. Therefore, they add an auxiliary task with an unsupervised objective to improve the sample efficiency. They perform this task of adding an autoencoder to model-free RL approaches with a focus on off-policy algorithms. They provide a demonstration that adding a simple auxiliary reconstruction loss to a model-free off-policy RL algorithm achieves comparable results to state-of-the-art model-based methods They also provide an understanding of the issues involved with combining autoencoders with model-free RL in the off-policy setting that guides the algorithm.

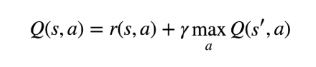
**Methodology and Design-**

**Techniques:**

We use a client and server approach and use the server-based python API for the interactions for the same. We add the clients in our server with this API and use the sensors for clients for interaction with the server environment.

**Deep Reinforcement Learning:**

Deep reinforcement learning (DRL) uses [deep learning](https://en.wikipedia.org/wiki/Deep_learning) and [reinforcement learning](https://en.wikipedia.org/wiki/Reinforcement_learning) principles in order to create efficient [algorithms](https://en.wikipedia.org/wiki/Algorithm) that can be applied on areas like [robotics](https://en.wikipedia.org/wiki/Robotics), [video games](https://en.wikipedia.org/wiki/Video_game), [finance](https://en.wikipedia.org/wiki/Finance) and [healthcare](https://en.wikipedia.org/wiki/Health_care). Implementing deep learning architecture ([deep neural networks](https://en.wikipedia.org/wiki/Deep_neural_networks) or etc.) with reinforcement learning algorithms ([Q-learning](https://en.wikipedia.org/wiki/Q-learning), actor critic or etc.), a powerful model (DRL) can be created that is capable to scale to previously unsolvable problems. That is because DRL usually uses raw sensor or image signals as input as can be seen in DQN for ATARI games and can receive the benefit of end-to-end reinforcement learning as well as that of [convolutional](https://en.wikipedia.org/wiki/Convolutional_neural_network) neural [network](https://en.wikipedia.org/wiki/Convolutional_neural_network)s.



The above equation states that the Q-value yielded from being at state s and performing action a is the immediate reward r(s,a) plus the highest Q-value possible from the next state s’. Gamma here is the discount factor which controls the contribution of rewards further in the future. Q-learning is a simple yet quite powerful algorithm to create a cheat sheet for our agent. This helps the agent figure out exactly which action to perform. In deep Q-learning, we use a neural network to approximate the Q-value function. The state is given as the input and the Q-value of all possible actions is generated as the output.

Given a transition < s, a, r, s’ >, the Q-table update rule in the previous algorithm must be replaced with the following:

1. Do a feedforward pass for the current state s to get predicted Q-values for all actions.
2. Do a feedforward pass for the next state s’ and calculate maximum overall network outputs max a’ Q(s’, a’).
3. Set Q-value target for action to r + γmax a’ Q(s’, a’) (use the max calculated in step 2). For all other actions, set the Q-value target to the same as originally returned from step 1, making the error 0 for those outputs.
4. Update the weights using backpropagation.

**Markov Decision Process:**

Our model uses the Markov Decision Process(MDP) which can be stated as,

**“Future is Independent of the past given the present”**

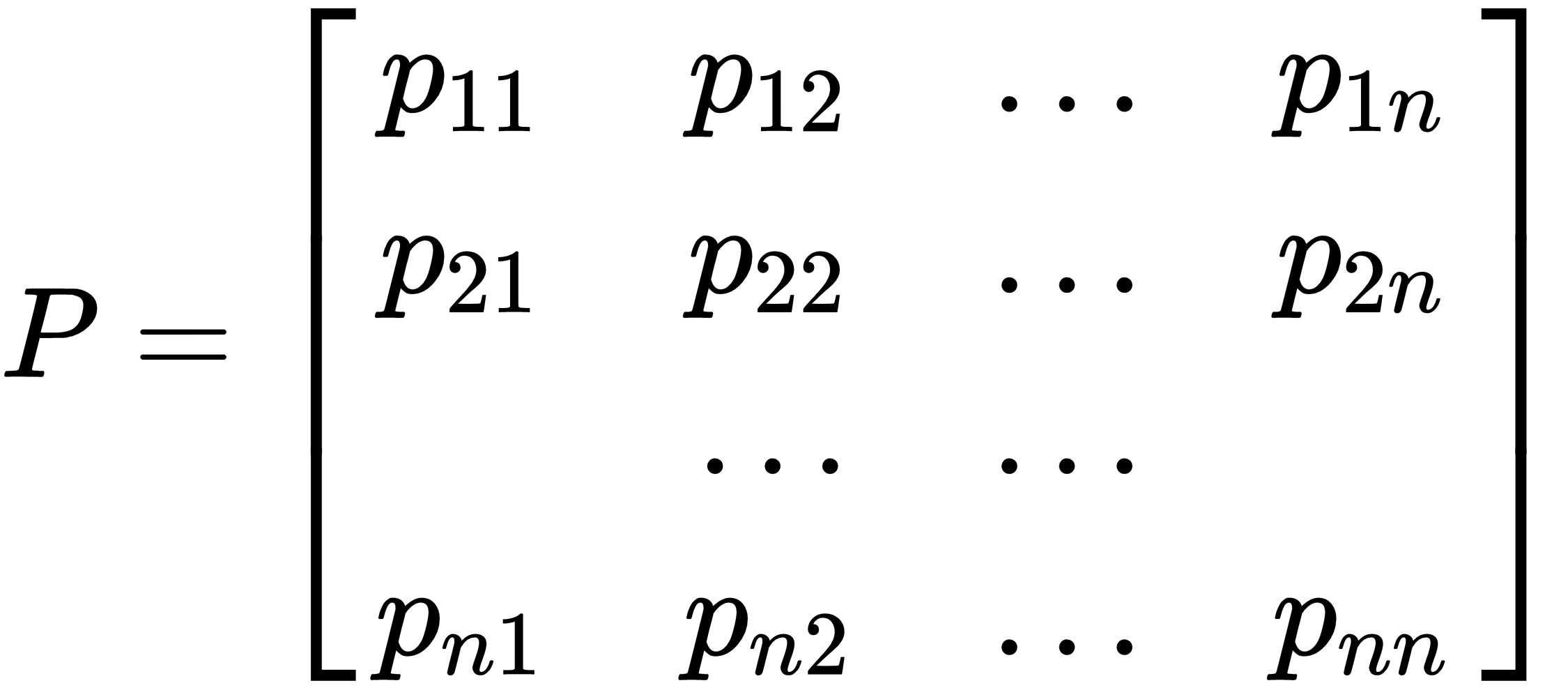
Mathematically the MDP can be represented as,

**P[ St+1 | St ] = P[ St+1 | S1, S2, …, St ]**

S[t] denotes the current state of the agent and s[t+1] denotes the next state. What this equation means is that the transition from state S[t] to S[t+1] is entirely independent of the past. So, the RHS of the Equation means the same as LHS if the system has a Markov Property. Intuitively meaning that our current state already captures the information of the past states. Similarly the state Transition Probability of MDP is represented and defined as, Markov State from S[t] to S[t+1] i.e. any other successor state , the state transition probability is given by:

***Pss’ =* P[ St+1 = s’ | St = s]**

We can formulate the State Transition probability into a State Transition probability matrix by,



Each row in the matrix represents the probability from moving from our original or starting state to any successor state. Sum of each row is equal to 1. Markov Process is the memory less random process i.e. a sequence of a random state S[1], S[2], …, S[n] with a Markov Property. So, it’s basically a sequence of states with the Markov Property. It can be defined using a set of states(S) and transition probability matrix (P).The dynamics of the environment can be fully defined using the States(S) and Transition Probability matrix(P).

In the case of our project, each and every frame captured by the RGB camera sensor is a state and the action to be taken is decided by the MDP. During the replay memory, a state in random is chosen initially and an action is performed on the state by the agent and a Q-value of that State-Action is generated and stored in the Q-table. This process continues for every state in the episode and the actions to be done on the current state is decided by the previous state and not the set of all previous states as every state is cumulative result of all its previous states.

**Fixed Target Model:**

In error calculation, target function is changed frequently with DNN. Unstable target function makes training difficult. So Target Network technique fixes parameters of target function and replaces them with the latest network every n steps. Why not use just use one network for both estimations? The issue is that at every step of training, the Q-network’s values shift, and if we are using a constantly shifting set of values to adjust our network values, then the value estimations can easily spiral out of control. The network can become destabilized by falling into feedback loops between the target and estimated Q-values. In order to mitigate that risk, the target network’s weights are fixed, and only periodically or slowly updated to the primary Q-networks values. In this way training can proceed in a more stable manner.

**Datasets:**

The collection of a dataset is different in our paper than in other approaches. We use CARLA to simulate the driving environment server, where we place the bunch of sensors that we can place upon the car to emulate real-life self-driving car sensors like LIDAR, cameras, accelerometers, and so on. These data sensors capture the environment around it, after which we pass the data to a function which pre-processes it. We also set the reward condition, which is whenever the agent doesn’t collide.

**Software Used:**

The main software that we are using is our virtual environment simulator CARLA. The main idea of CARLA is to have the environment (server) and then agents (clients). CARLA has three major concepts: Actors, Blueprints, and World. *Actor* is anything that plays a role in the simulation and can be moved around, examples of actors are vehicles, pedestrians, and sensors. Before spawning an actor you need to specify its attributes, and that's what *Blueprints* are for. The *World* represents the currently loaded map and contains the functions for converting a blueprint into a living actor, among others.

For the implementation of the project, we are using Python 3.7, Tensorflow 1.14.0 and CARLA 0.9.5. The environment that we are using to simulate has a working implementation of a car in the environment which has RGB sensor and the collision sensor in it. In our main script, we have a method that performs the necessary image preprocessing on the frames captured by the RGB sensor. Following the image preprocessing, our script then has a method that created the architecture, trains the agent and creates the model.

**Model Used:**

In order to train a model that teaches itself to drive a car in the CARLA environment, we have written a custom model that has 3 convolutional layer with 3 average pooling layers. For the purpose of processing the image data that we obtain from the RGB camera sensor attached to the car in our environment, we will be using the above model which is essentially a CNN model.We’ll be adding 3 neurons in the output layer, each one for left, right and straight movement prediction.

**State, Actions, Rewards:**

In terms of states of the agent in the CARLA environment, the states that the agent can be in are the actual RGB values that are received from the RGB sensor. Based on these state values, the agent takes one of the three actions, viz, go left, go right, and go straight. These actions in turn change the state of the agent in the environment frame by frame, and provide the agent rewards based on the action taken, i.e. +1 for each frame driving > 50 KMH, -1 for each fram driving < 50 KMH, and -200 for a collision, where the episode prematurely finishes and starts over.

**Training, analysis and evaluation:**

In this part, we combine all the above discussed techniques. With DQNs, instead of a Q Table to look up values, we have a model that we inference (make predictions from), and rather than updating the Q table, we fit (train) our model.

As we engage in the environment, we will do a .predict() to figure out our next move (or move randomly). When we do a .predict(), we will get the 3 float values, which are our Q values that map to actions. We will then do an argmax on these, like we would with our Q Table's values. We will then "update" our network by doing a .fit() based on updated Q values. When we do this, we will actually be fitting for all 3 Q values, even though we intend to just "update" one.

Our model is starting off as random, and it's being updated every single step, per every single episode. What ensues here are massive fluctuations that are confusing to our model. Therefore, we develop a seperate model called the *target\_model()* which will inherit the weights from the original model every *n* episodes, this the model that we use to determine what the future Q values.

Additionally, we also uses the replay\_memory which is another way that we attempt to keep some sanity in a model that is getting trained every single step of an episode. We still have the issue of training/fitting a model on one sample of data. This is still a problem with neural networks. Thus, we're instead going to maintain a sort of "memory" for our agent. In our case, we'll remember 5000 previous actions, and then we will fit our model on a random selection of these previous 5000 actions. This helps to "smooth out" some of the fluctuations that we'd otherwise be seeing.

We are training a model in a windows environment which has i7 6th gen processor, 16GB DDR4 RAM and we’ll be using CUDA to enable GPU programming on the Nvidia 1660Ti GPU. We would be training the model for 12,000 epochs with 64x3 CNN model.

After the training we will evaluate the results using the metrics such as loss, accuracy, reward average and epsilon by plotting them into graph with the usage of Tensorboard. We would be using Mean Squared Error for loss function and we would be using Adam optimiser with learning rate of 1e-4.

For analysing how the trained model will be performing, we will be developing script which will have the trained model and an agent will be spawned in the CARLA environment. From this, we can see how the agent maneuvers the obstacles in real time environment.

**Experiments:**

Initially when we started this project we decided to keep the problem simple by limiting the actions of the agent to just one of three total actions: Turn Left, Turn Right, Go straight. First we went with the Xception model, a model in Keras with71 hidden layers. As mentioned earlier, the reward conditions were set to the following:

* +1 for each frame driving > 50KMH
* -1 for each frame driving < 50KMH
* -200 for a collision and episode is over

Since we are using the DQN model, it is superficial to expect high accuracy as we might expect from other methods like supervised learning or imitation learning. Of course a model with good accuracy is important and we want something decent. It is important to remember that, since we are using DQN model our agent is taking a large number of just plain random actions, and is slowly learning the Q values. The neural network is part of the operation, but a highly accurate neural network, especially in the beginning would be a red flag.

Another important thing that we had to keep in mind is the frame rate of the simulation. It is very important that we maintain at least 10+ FPS during the time when the model is training. Due to the computational constraints of this project, the maximum we could achieve was 13 to 14 FPS and an average of 11 FPS. When it comes to something like driving a car, the higher the FPS the better. We aimed to average around 11FPS, controlling it by how many agents we decided to spawn. If you can barely run even just 1 agent, then you don't have much ability to tinker here.

Keeping all these factors in mind, we decided to train the agent with the Xception model. The agent was spawned in the CARLA environment in random locations for every episode. The reason why we did not go about spawning the agent in the same location for every episode because in our replay memory, we are taking batches of episodes with random states (Here states are every image captured by the RGB camera sensor) and spawning the car at the same location will feed the episodes with the same set of states for every episode. This will make the agent train with the same set of states for initial set of episodes and yield us very good Q-values for only those states. The agent will not be able to perform well with those calculated Q-values in other states.

After creating the script with Xception model, the first thing we decided was to run the model for 1000 episodes just to see how the results were turning to be. One thing we noticed was we were not getting decent results that we expected to get after 1000 long episodes. Of course we cannot expect excellent agent that runs well, but in our case the agent was not even doing anything other than just circle around in the same spot. This was because of our Xception model. Even though Xception is a good model to train self driving cars, it is an extremely complex model with 71 hidden layers. This increased our computational requirements which we did not have access to. The only way we can get decent results was to change our model to something much more simpler. For this purpose, we wrote our own neural network that had 4 hidden layer CNN. In our case but simpler seems better as there are less parameters to learn. For fully-supervised learning, we thought more parameters worked well because everything was a "ground truth." For reinforcement learning, we decided it's just too hard for the AI to learn to drive a car by trying to train 10s of millions of weights.

Our first experimentation during the training phase was to train the agent in an environment with dynamic weather and dynamic traffic conditions with 100 other agents in the environment. These other agents comprised of other vehicles of various sizes ranging from motorbikes, cars, and trucks. The agent also comprised of pedestrians who tend to cross the road at random moments. Dynamic weather conditions will alter the lighting conditions and weather conditions randomly over time from a beautiful dry sunny day to really wet rainy day with dull lighting. Although this is the apt environmental conditions that mimics the real world scenarios, we faced several problems during the training phase of the agent. The first main hindrance was that since we are spawning our agent at random locations in the environment, the agent/car might land right on top of another agent in the environment and end the episode even before the agent starts to learn the environment. And since there are lots of dynamic agents and conditions in the environment, a model that trained for around 12,000 episodes could not even make the agent drive in a straight road without colliding with another agent in the environment. This is because in the course of 12,000 episodes, the agent just fails numerous times even before starting to learn and adapt to the environment.

Our next set of experiments involved removing dynamic traffic from the environment and just keep the dynamic whether so that we can eliminate the problem of the episodes failing even before it started. This method of training the agent yielded much better results than the previous experiment. As mentioned earlier dynamic whether conditions will alter the lighting conditions and weather conditions randomly over time. Episodes that were trained during good lighting conditions became better over time but for the case of agents that were trained during dim and overcast conditions failed sooner because the RGB input that we were getting from the camera sensor were different than the bright images from the former conditions and hence, required extra image processing in order to give reasonable results. This requirement of extra image processing was an overhead for us as our computational capabilities were limited and hence an agent that was trained for 12,000 episodes did not give results that were decent enough.

Finally, we decided to just keep the environment in normal bright conditions without any dynamic traffic to make things simple and reduce the computational complexity. When the agent was trained for 12,000 episodes in these conditions, we could see a decent enough model that was running very well on straight roads and also made decent left and right turn in the road when it was required occasionally. The overall reason why we had to cut corners to get a decent enough model to drive a car was the computational constraints we had.

**Results:**

We trained the agent multiple times using the same model with checkpoints. The first time we trained the agent for 650 episodes. Then we reload from the checkpoint and continue to train to 3100 episodes. Finally, we complete the training at 12,000 episodes. Following are the results we obtained in terms of accuracy, epsilon value, loss, the maximum, minimum, and the average reward, plotted against the total number of episodes:

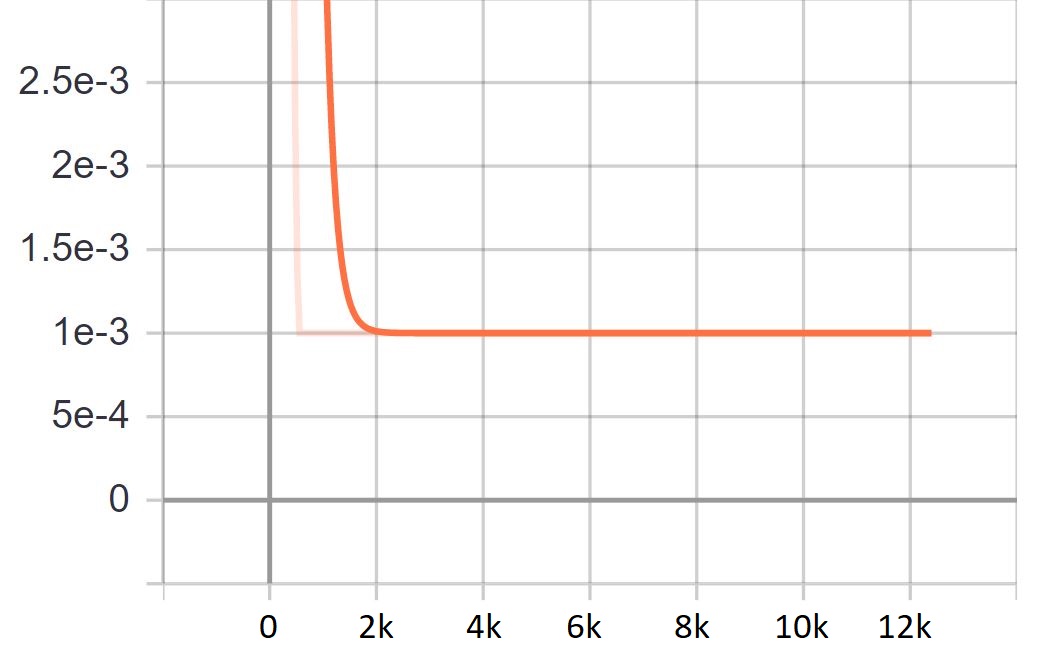
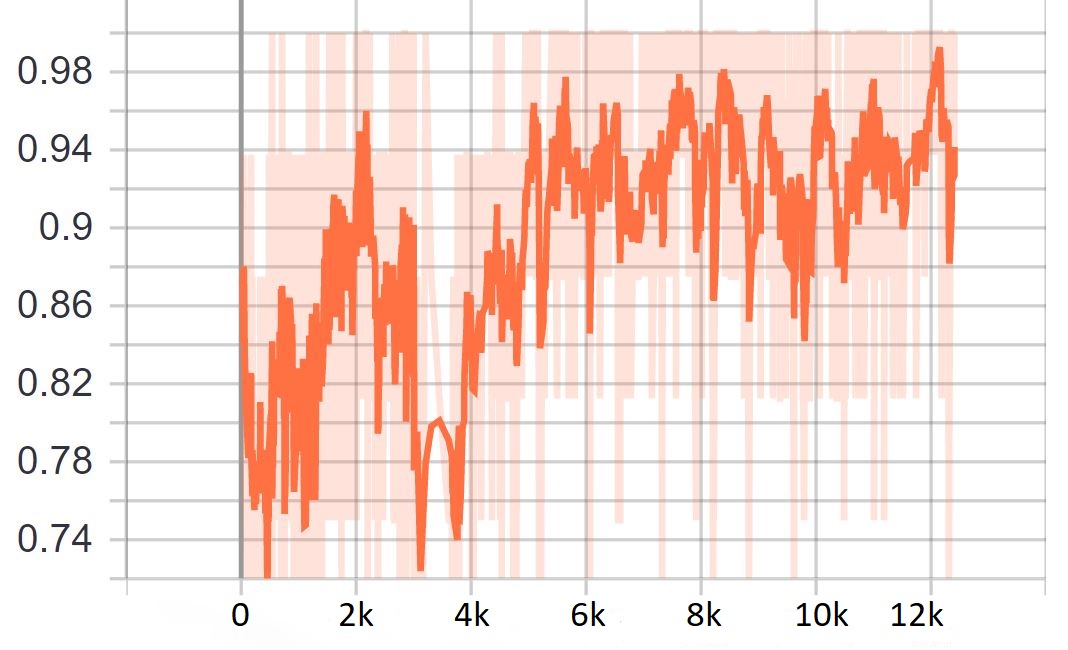


Figure 1: Accuracy Figure 2: Epsilon

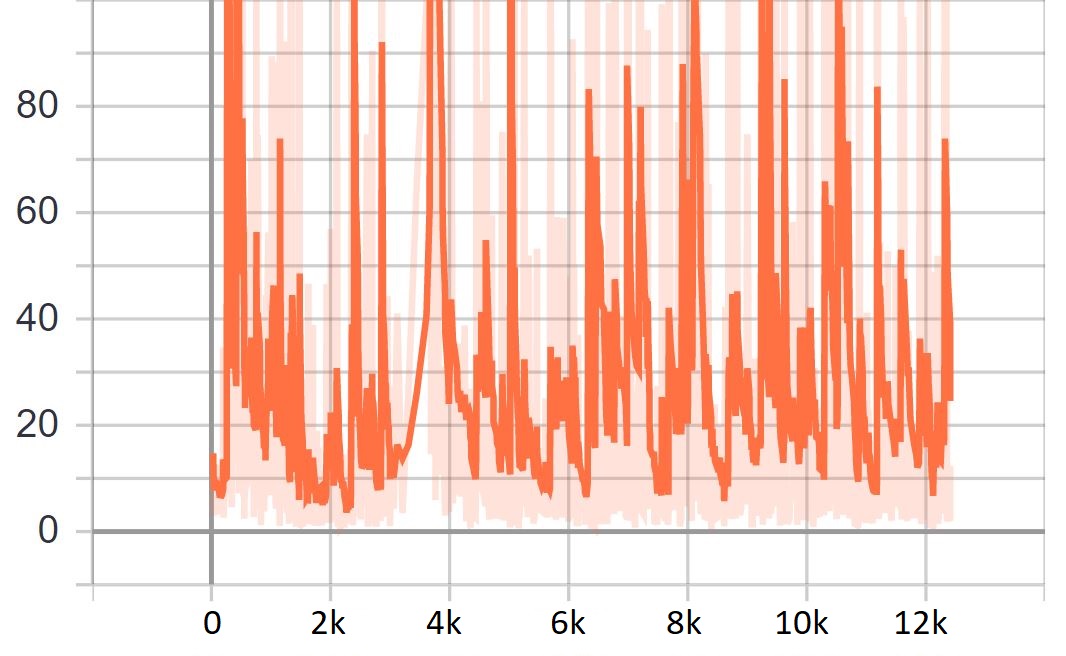
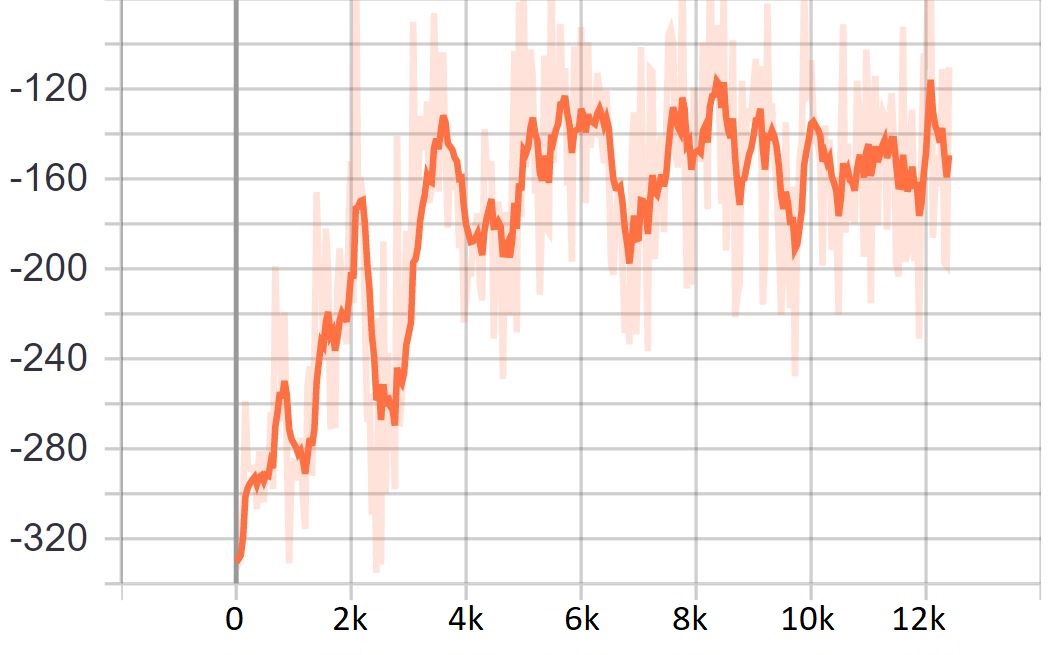
 

Figure 3: Loss Figure 4: Reward\_average

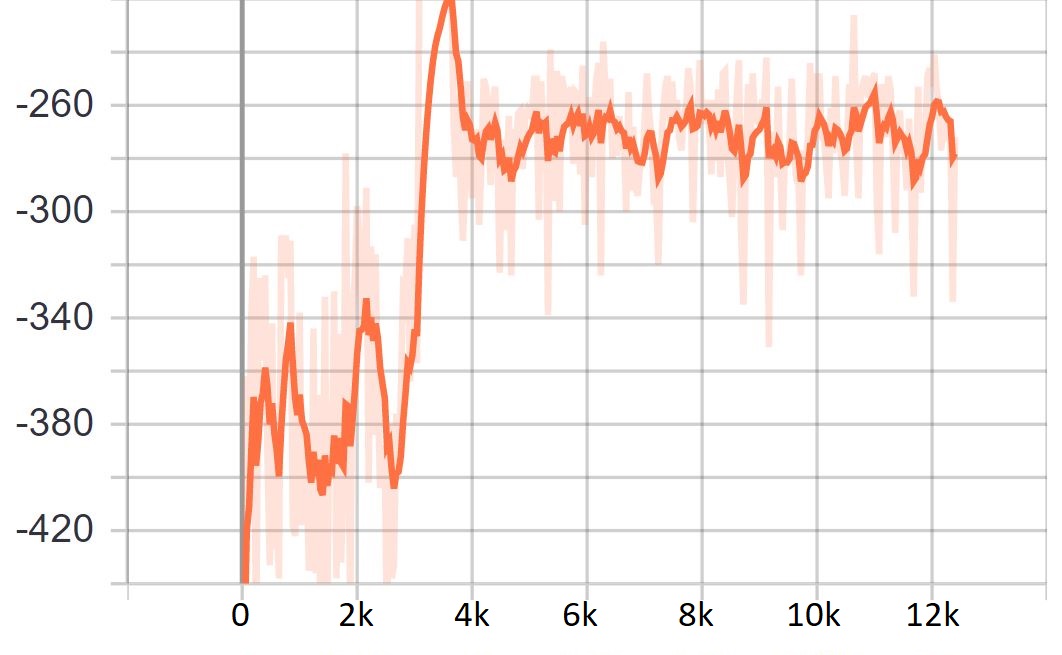
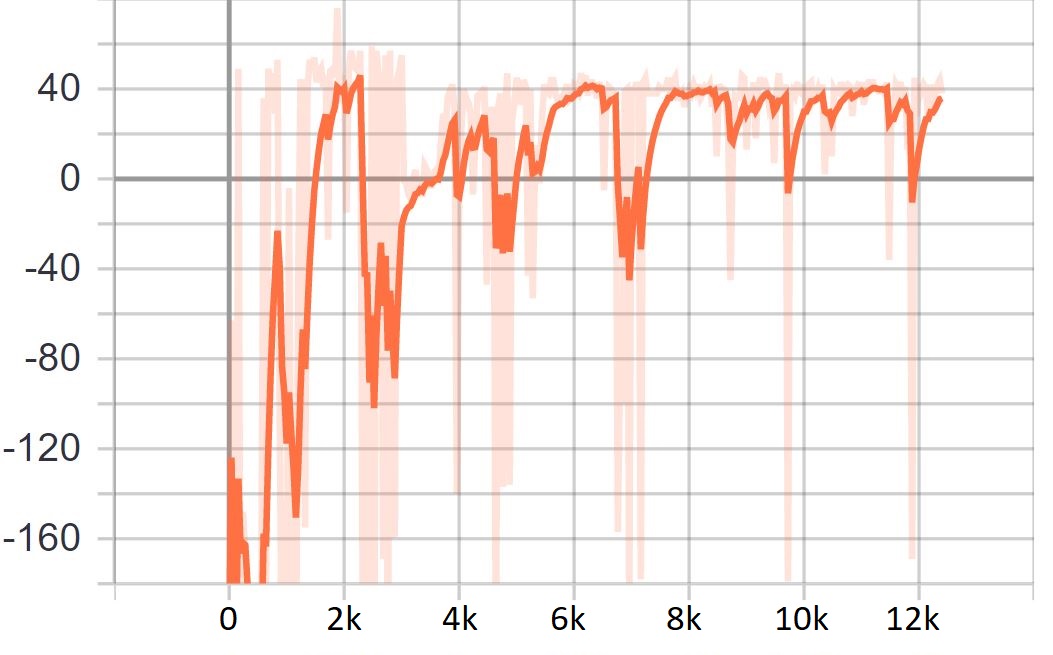


Figure 5: Reward\_max Reward 6: Reward\_min

From these graphs, we see that the accuracy for our model on an average was around a good 85%.Next, we see that the epsilon, which shows our attempts to explore some more, helps sometimes, other times not really. In loss rate, we see that there were no explosions.

For rewards, We can see that the max reward over time does trend up until the 12,000 episodes, and may have continued back up given more time. The minimum reward (ie: the worst an agent did), didn't seem to change drastically trend-wise, which isn't too shocking. Sometimes the car is just dropped into an unfortunate setting. Finally, we can see that the reward average improved slowly, which meant that overall, the model did improve, and this model turned out to be our best performing model.

**Discussion:**

When we used the Xception model, the first thing that we discovered was that both loss and q values were exploding. Reinforcement learning is quite a bit different from supervised learning, mainly in the fact that supervised learning is pure ground-truth (or at least that's the expectation). All imagery we feed it, and the labels are meant to be 100% accurate. With reinforcement learning, this isn't really the case. We're fitting a model and we're also fitting for these Q values. It's far more complex operation going on, and things are going to be a bit more "fuzzy" to the model. That is the reason why we shifted from the Xception model, to the simpler model. With this, we get a good accuracy of over 85%

In the end, it really was the case that more time was required. All checkpoints considered, collectively, this agent trained for over 40 hours alone with this model, under the basic circumstances no dynamic weather and traffic, since when considering traffic, the episodes failed directly when even a single vehicle from the traffic was spawned in an incorrect position, and when considering dynamic weather, the kernel failed most of the time, because of the complexity in the images provided by the RGB sensor.

**Future Research**:

In terms of future research, based on the computing power available, we would like to like to train the agent using the different models and parameters, for even more episodes, and compare the results of those models with each other, along with comparing the results at different episodic check points. We would also like to increase the time taken for each episode, and retrain the model based on that, and check for the results. Our current computational capacity prevents us from taking those steps.

In terms of dynamic weather, we would like to devise and apply some image processing algorithms to brighten and improve the image quality, to help the agent make better choices based on the improved and more accurate values of the data. We would then like to select the best image processing algorithm which gives us the best results. We would also like to devise an algorithm that could eradicate the problem of incorrect spawning of traffic vehicles, and then perform training the agent on this environment. We also aim to add more actions for the agent to impact the environment, like reverse, and brake.

**References-**

[1]Pan, You, Wang & Lu - *Virtual To Real Reinforcement Learning*, arXiv (2017)  
[2]Shai Shalev-Shwartz, Shaked Shammah, Amnon Shashua - *Safe, Multi-Agent, Reinforcement Learning for Autonomous Driving,* arXiv (2016)  
[3] Sahand Sharifzadeh, Ioannis Chiotellis, Rudolph Triebel, Daniel Cremers - *Learning to Drive using Inverse Reinforcement Learning ANd DQN,* arXiv (2017)  
[4]Markus Kuderer, Shilpa Gulati, Wolfram Burgard - *Learning Driving Styles for Autonomous Vehicles from Demonstration,* IEEE (2015)  
[5]Jeff Michels, Ashutosh Saxena, Andrew Y. Ng *- High Speed Obstacle Avoidance using Monocular Vision and Reinforcement Learning, ACM (2005)*[6]Ahmad El Sallab, Mohammed Abdou, Etienne Perot & Senthil Yogamani *- Deep Reinforcement Learning Framework for Autonomous Driving,* arXiv (2017)  
[7] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun -*CARLA: An Open Urban Driving Simulator*.  
[8]April Yu, Raphael Palefsky-Smith, Rishi Bedi- *Deep Reinforcement Learning for Simulated Autonomous Vehicle Control*  
[9]Jan Koutník, Giuseppe Cuccu, Jürgen Schmidhuber, Faustino Gomez- *Evolving Large-Scale Neural Networks for Vision-Based Reinforcement Learning*[10]Chenyi Chen, Ari Seff, Alain Kornhauser, Jianxiong Xiao- *DeepDriving: Learning Affordance for Direct Perception in Autonomous Driving*[11]Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver & Daan Wierstra- *Continuous Control With Deep Reinforcement Learning*[12]Brody Huval, Tao Wang, Sameep Tandon, Jeff Kiske, Will Song, Joel Pazhayampallil, Mykhaylo Andriluka, Pranav Rajpurkar, Toki Migimatsu, Royce Cheng-Yue, Fernando Mujica, Adam Coates, Andrew Y. Ng - *An Empirical Evaluation of Deep Learning on Highway Driving*[13] Abdur R. Fayjie, Sabir Hossain, Doukhi Oualid, and Deok-Jin Lee- *Driverless Car: Autonomous Driving Using Deep Reinforcement Learning In Urban Environment.* [14]Matt Vitelli, Aran Nayebi, *CARMA: A Deep Reinforcement Learning Approach to Autonomous Driving.*[15]David Isele1,Reza Rahimi,Akansel Cosgun,Kaushik Subramanian and Kikuo Fujimura- *Navigating Occluded Intersections with Autonomous Vehicles using Deep Reinforcement Learning.*[16] Dorsa Sadigh, Shankar Sastry, Sanjit A. Seshia, and Anca D. Dragan- *Planning for Autonomous Cars that Leverage Effects on Human Actions*

[17]David Ha, Jurgen Schmidhuber, *World Models*

[18]Denis Yarats, Amy Zhang, Ilya Kostrikov, Brandon Amos, Joelle Pineau, Rob Fergus, *Improving Sample Efficiency in Model-free Reinforcement learning from Images*