Deep Learning For Autonomous Driving in Grand Theft Auto IV

Hyunjun Choi
Viterbi School of Engineering
University of Southern California
Los Angeles, California, USA
choi797@usc.edu

Manpreet Singh
Viterbi School of Engineering
University of Southern California
Los Angeles, California, USA
msingh60@usc.edu

Naicih Liou
Viterbi School of Engineering
University of Southern California
Los Angeles, California, USA
naicihli@usc.edu

Raveena Kshatriya
Viterbi School of Engineering
University of Southern California
Los Angeles, California, USA
kshatriy@usc.edu

Ritika Chaudhary
Viterbi School of Engineering
University of Southern California
Los Angeles, California, USA
ritikach@usc.edu

Suyash Kanungo
Viterbi School of Engineering
University of Southern California
Los Angeles, California, USA
skanungo@usc.edu

***Abstract*—This paper sheds light on our attempt to create an artificially intelligent bot capable of playing as a driver in GTA IV and navigate through the environment, avoid hitting other cars, pedestrians, and road obstacles and reach a destination as safely as possible. We explored several neural network models for image classification, like AlexNet [1], InceptionV3 [2], Xception [3] and ResNet50 [4] and trained those on 200K game play image data. The results so far are quite encouraging and we observed that using virtual game environments like GTA is an effective way to create new data for training and testing of autonomous driving agents.**

***Index Terms*—Autonomous Vehicles, CNN, Deep Learning**

I. Introduction

Games as an educational tool provide opportunities for testing out deep learning algorithms, like most games are used today to test AI controlled bots. Grand Driving Auto is an effort to create an artificially intelligent agent which can drive a vehicle in a simulated environment. Near perfect simulation of the real world can be obtained by using simulators and games like GTA, for the purpose of this project we chose the game Grand Theft Auto IV (PC) as our environment and the aim of the agent is to navigate through the city to reach a destination as safely as possible. This project will serve as a proof of concept for research in the field creating and training autonomous vehicle agents in a secured & simulated environment before testing these out in the field. The overall idea is to build a bot that can theoretically drive on the virtual roads as close to real driving as it can get. This paper goes through the life cycle of the project in detail, the methods we followed and talks about the results, analysis & limitations of the project along with the scope of possible future work in this field.

Problem Definition

The purpose of this paper is to describe the process of building a bot that plays as a driver in Grand Theft Auto IV (GTA-IV) and navigate through the environment to reach a destination as safely as possible. The bot will try to avoid hitting other cars, pedestrians, and other road obstacles. Various attributes of the game are described in the following sections.

A. Grand Theft Auto Games

Grand Theft Auto (GTA) is a series of action adventure games developed and maintained by Rockstar games. Gameplay focuses on an open world where the player can complete missions to progress an overall story, as well as engage in various side activities. Most of the gameplay revolves around driving, shooting missions, and stealing cars. The games in the Grand Theft Auto series are set in fictional locales modelled after real-life cities, at various points in time from the early 1960s to the 2010s. The original game's map encompassed three cities—Liberty City (based on New York City), San Andreas (based on San Francisco) and Vice City (based on Miami).

GTA IV takes place in the fictional Liberty City, based off of New York City and the surrounding areas including Brooklyn, Manhattan, The Bronx and New Jersey. The game comprises shooting missions, driving missions, and story-based missions. It is an open world game, allowing the player to drive to any point in the game map at any time. The driving environment is complex and realistic, involving heavy traffic, pedestrians, lampposts, city landmarks and bodies of water. The game recreates several famous New York City streets and landmarks and is a realistic representation of the city.

Driving in the game is controlled using standard WASD keys, which will be the labels our model is trying to predict:

- W: Forward
- A: Left
- D: Right
- WA: Accelerate Left
- WD: Accelerate Right
- S: Brake/Reverse
- SA: Reverse Left

- SD: Reverse Right
- NK: No Key

For our model, we added this “No Key” class to give the possibility of pressing no key. We will train our model to drive using the first-person view of the street.

B. Purpose

Artificially intelligent bots in a game can be built by coding rules that impart game intelligence. Most of the tasks involve driving around the cities using the map for navigation. Hence, the motivation behind the project. We want to create an artificially intelligent bot which can drive on its own by making it learn from humans driving around the game.

Exploring this requires us to play GTA and drive around town in order to collect data of human players ahead of training the bot. While playing we record our in-game actions and decisions which in turn will allow us to train an end-to-end Deep Learning bot without hard coding the rules.

C. Goal

With growing AI in every sphere and bots being created to replicate every human action with the same or better performance level, the urge to do the same for the game GTA arose. The intent is to create an artificial intelligent bot which can master driving within the time limit without going off the road and hitting obstacles. The bot should replicate the actions performed by a driver using keypress simulation or even perform better than a human would do. Performing better would mean reaching a destination in minimum time without hitting any obstacle. The initial objective is for the bot to correctly identify lanes on the road and correctly choose actions depending upon the map to follow (like when to take a turn, stop, or move right or left). With humans playing, there is always a chance of making mistakes in a driving game. The desire is to reduce those cases since the chance of human error is out of the picture. The GDA bot should eliminate the situations where humans make mistakes, thus increasing the accuracy of actions and the efficiency with which a destination is reached.

II. Related Works

Research on Autonomous cars have existed long back, but was seriously considered in the 1980s with the ALVINN project where neural network was used to map input images directly [5].

Muller [6], proposed a Convolutional Neural Networks based off road driving robot, DAVE, that mapped images to steering angles. Huval [7], described a CNN system that detects vehicles and lane markings for highway driving showing that CNNs have promise in autonomous driving. Video game data was used to augment real datasets to provide coverage for scenarios that were difficult to find data for in the real world [8].

Car manufacturers and researchers have been working on autonomous driving for years and significant progress has been made. Although firms like Google are teaching their software by physically driving millions of miles in the real world, they also train their algorithms using pre-recorded footage of traffic. Computers need hundreds of thousands of labelled images, showing how to drive, to make them expert vehicle drivers [7]. Several researchers have used games like GTA and other simulated environments, Martinez built up a complex convolutional neural network to make a self driving bot in GTA V which distance to cars/objects ahead, lane markings, and driving angle in the game [9].

A. Convolutional Neural Network (CNN)

Convolutional neural network (CNN) is a class of deep learning neural networks. CNNs represent a huge breakthrough in image recognition. They’re most commonly used to analyze visual imagery and are frequently working behind the scenes in image classification. They can be found at the core of everything from Facebook’s photo tagging to self-driving cars. They’re working hard behind the scenes in everything from healthcare to security.

CNNs are distinguished from traditional neural networks in that they are built up of convolutional layers, which are a set of learned kernel filters (weights) that are convolved with regions of an input image to generate a feature map. This feature map is essentially a summary of features detected in the input. In a CNN, the output of each convolutional layer is passed to the next layer, and at each layer higher level features of the input image are learned. This allows for learning complex spatial relationships and features in images, images go through a series of convolutional layers with filters (Kernels), Pooling, and end in fully connected layers seen in traditional neural networks to produce output for image classification or other tasks. In the case of image classification, a softmax or sigmoid activation function is applied to the output of the fully connected layers to generate a probability distribution [10].

A CNN convolves learned features with input data and uses convolutional layers. This means that this type of network is ideal for processing images. Compared to other image classification algorithms, CNNs actually use very little preprocessing. This means that they can learn the filters that have to be hand-made in other algorithms. CNNs can be used in tons of applications from image and video recognition, image classification, and recommender systems to natural language processing and medical image analysis [11].

B. Image Classification

The goal of image classification is to assign an input image to one of a predefined set of classes based on what is featured in the image. Oftentimes image classification is used when there is one main object featured in the image, but can also be used when multiple objects are featured. One of the biggest challenges in image classification is that the subjects can appear in different positions in the image, be of different colors, not be completely in the frame, and other variabilities. The goal is to be able to learn the main distinguishing features of the objects we are aiming to classify, so that the objects can be correctly classified no matter what variabilities might occur. Convolutional Neural Networks are typically used for this task, we have used state-of-the-art Convolutional Neural Networks like Alexnet, InceptionXception and Resnet which have performed extremely well in the ImageNet Challenge.

C. *Object Detection*

Object detection [12] is a computer vision application where the goal is to identify and locate numerous objects within an image or video. Object detection has been widely used for face detection, vehicle detection, pedestrian counting, web images, security systems and driver-less cars. Object detection draws bounding boxes around these detected items, thus allowing us to locate and perform actions with those objects. In object detection, the tasks of image classification and localization are combined. The breakthrough and rapid adoption of deep learning brought into existence modern and highly accurate object detection algorithms and methods such as R-CNN, Fast-RCNN, Faster-RCNN [13].

D. *LSTM Models*

Long Short-Term Memory (LSTM) networks are used to classify and make predictions based on time series data. It has feedback connections which enables it to process entire sequences of data, like in a video. They are also relatively insensitive to gap duration as there can be gaps between important events in real life scenarios of different time duration. This provides LSTMs an advantage over other RNN layers and other sequence learning methods. [14]

III. Environments and Data

A. *Environments*

To create a self driving bot we needed a simulation environment that provide us ample of real world scenarios for training data and testing purposes, Games like GTA IV provide us with very large open world map, and very good good game physics, while requiring minimum computational resources thus proving to be the best choice for our simulation environment.

environment for its open world and GPS feature for navigation. The game also has significantly lower computational requirements than other versions of GTA allowing us to utilize computational power on our local machines for data collection, training and testing purposes.

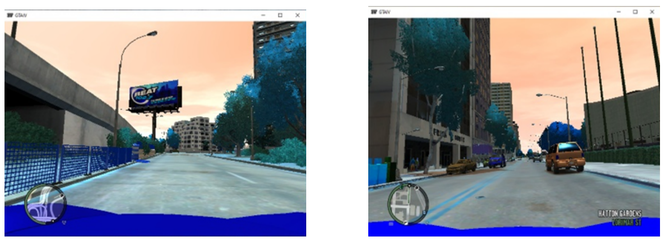
the whole map of the game had to be unlocked and for better data collection certain Game mods were used.

a) Lighting Mod: The lighting in the game is poor compared to other GTA games. Most of the game is very dark, and in some areas it can be difficult for even a human player to see the lanes on the road. To make it easier for the model to learn that it needs to stay within lanes, we install a mod to improve lighting in the game i.e. G4BP Lighting Overhaul 1.2

b) Trainer Mod: A trainer mod was used, which allows us to freeze time and weather, spawn cars, and set other player protections. Using this we can control the environment of the game as much as possible. Mod settings used to generate the majority of the training data include preventing the car from being chased by the police or getting damaged, freezing time at 12 pm and weather at extra sunny.

B. *Data Collection*

Data collection is using a script that continuously captures frames from the game within a 800x600 window along with the key presses of the human player while driving, taken as labels, these are the 9 data classes: W, A, S, D, WA, WD, SA, SD, and NK. 500 such samples are stored as a (image, key) tuple in a numpy file. So far we've collected 200,000 RGB images for training the models. No external data is used.



(a) training image from random driving data, "W" (accelerate) class

(b) training image from waypoint driving data, "WD" (accelerate right) class

Fig. 1: Training images collected by driving in the game

C. *Data Preprocessing*

1) Data Resize: Complex CNN architectures being used in the project were computationally expensive on local machines, requiring high GPU memory. For efficient memory utilisation the images were resized to 400x300 to use as input for our training models.

2) Data Augmentation: The raw data collected from the game frames, was largely imbalanced having images from the W and NK class, much more than other classes. To fix this problem without much effort on human player's part, we implemented Data Augmentation where images from the minority classes were duplicated by transforming the hue, saturation, and construct. The augmented data was 2 times the amount of the raw data.

Other data augmentation techniques like flipping, clipping, and rotation, were not applicable in our case since these conditions do not occur naturally in the real-world driving.

IV. Methodology

Our initial approach is to train a Convolutional Neural Network classifier suited for object detection to predict what the correct keypress is, given a screenshot from the game while driving.

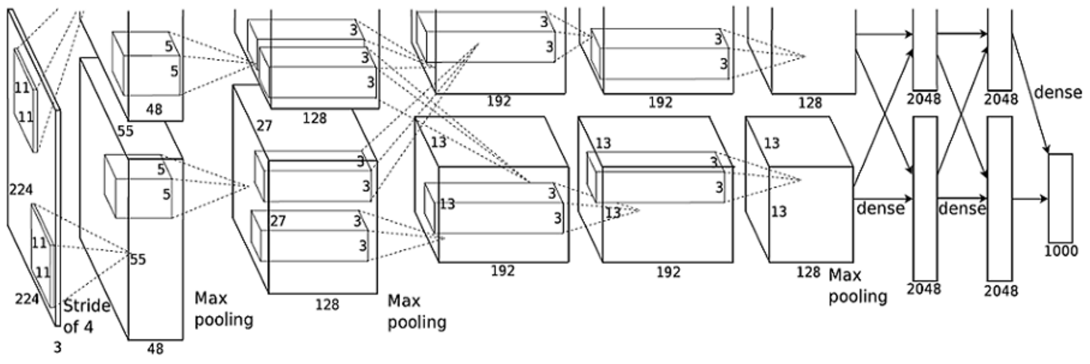


Fig. 2: Architecture of AlexNet

A. *Model Architecture*

The Model training is done in batches, to use the available memory efficiently and not load the complete dataset which could exhaust the memory. We use a Script that loads 2500 image samples at a time and creates batches from them. Training hyperparameters: Number of epochs, Batch size, Learning Rate, Learning Rate decay rate, Optimizer to use, Number of files to train on (number of training samples)

Training is carried out on several Models using both pre-trained models using ImageNet weights and training them from scratch. On pre-trained models only the fully connected layers are trained according to our specific task, while ImageNet training weights are used for the other convolutional layers.

Models we used are as followings:

- CNN - Baseline model with 3 convolution layers + 2 full connected layers
- AlexNet
- AlexNetV2 - AlexNet + 2 fully connected layers
- InceptionV3
- Xception
- ResNet50
- Time Dependent Models
  - LSTM
  - ConvLSTM

1) AlexNet: The Architecture consists of eight layers with learnable parameters: five convolutional layers and three fully connected layers. AlexNet uses Rectified Linear Units(ReLU) instead of the tanh function as the activation function in all layers which provides an advantage in training time. The activation function used in the output layer is Softmax. It uses two Dropout layers and a total number of 62.3 million parameters. "Fig. 2", Depicts the Architecture of AlexNet.

2) AlexNetV2: This model follows the standard AlexNet architecture described above, but adds 2 more fully connected layers with dropout after each layer to the end of the network. The purpose of this is to simply increase the complexity of the standard AlexNet architecture, after observing inadequate performance with the standard AlexNet [15].

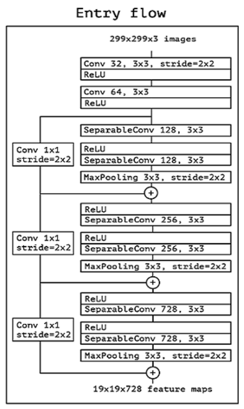
3) InceptionV3: Inception V3 is a pre-trained convolutional neural network model that is 48 layers deep and uses transfer learning. It is trained on more than a million images from the ImageNet database and can classify images into 1000 object categories. As a result of training, the network has learned rich feature representations for a wide range of images. The model extracts general features from input images of size 299-by-299 and classifies them based on those features. While building a new model to classify our data we can reuse the feature extraction part and just retrain the classification part with our data.

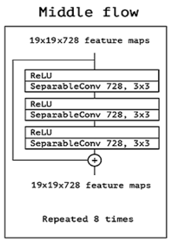
4) Xception: Xception [3] offers an architecture that is made of Depthwise Separable Convolution blocks + Max-pooling all linked with Shortcuts between Convolution blocks as in ResNet implementations. The data first goes through the entry flow, then through the middle flow which is repeated eight times, and finally through the exit flow. All the Convolution and Separable Convolution layers are followed by batch normalization. The specificity of XCeption is that the Depthwise Convolution is not followed by a Pointwise Convolution instead the order is reversed.

"Fig. 3", shows the data first goes through the entry flow, then through the middle flow which is repeated eight times, and finally through the exit flow.

5) ResNet: The ResNet-50 has four stages, all ResNet architectures perform the initial convolution and max-pooling using 7×7 and 3×3 kernel sizes, then the four stages in the network start, each stage with 3 Residual blocks containing multiple layers each. The size of kernels used to perform the convolution operation in all 3 layers of the block of each stage are shown in Figure 11. As we progress from one stage to another, the number of filters is doubled and the feature map size is reduced to half. Finally there is an average pool and end it with a fully connected layer containing 1000 nodes and at the end a softmax function so this gives us a total of 50 layers. The ResNet-50 has over 23 million trainable parameters. "Fig. 4", Detailed architectures for various Residual Nets

6) Object Detection: Object detection in a driving scenario helps to track various objects such as pedestrians, vehicles,





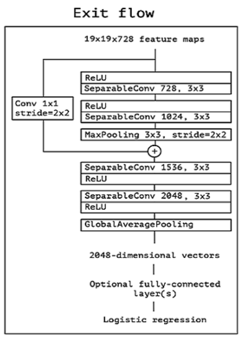
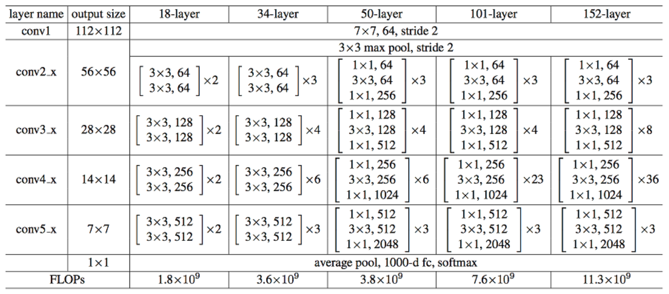


Fig. 3: The Xception architecture

lanes, and trees. This technique is vital in physical movement of the car and to track other moving objects such as the traffic and pedestrians, and make decisions based on the obstacle present in front.

Faster RCNN with InceptionResnetV2 [16], a pretrained model from Tensorflow object detection API, was used to perform Object Detection on the collected game frames, and the extracted features were passed on to time dependent models like LSTM, to predict the correct keypress based on detected obstacles and previous timesteps.



layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix}3\times3, 64\\3\times3, 64\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 64\\3\times3, 64\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix}3\times3, 128\\3\times3, 128\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 128\\3\times3, 128\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix}3\times3, 256\\3\times3, 256\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 256\\3\times3, 256\end{bmatrix}\times6$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times6$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times23$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix}3\times3, 512\\3\times3, 512\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 512\\3\times3, 512\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Fig. 4: ResNet Architectures for ImageNet

7) LSTM: A common LSTM unit is composed of a cell, an input gate, an output gate and a forget gate. LSTM is a Recurrent Neural Network Architecture that carries information as cell state and the flow of information is controlled by the gates throughout the processing. [14] This model consists of two LSTM layers with dropout and two fully connected layers. Each training sample is a sequence of 60 feature vectors outputted by the object detection model. Label is the keypress associated with the final feature vector (final screenshot). This model outputs the keypress.

8) ConvLSTM: A ConvLSTM model is a variant of a Recurrent Neural Network with convolutional operations in its LSTM cells for spatio-temporal prediction (which leads to convolutional structures present in the input-to-state and state-to-state transitions). The ConvLSTM determines the future state of a certain cell in the grid by the inputs and past states of its local neighbors. Unlike traditional LSTM models, it can capture underlying spatial features which makes it well suited for image classification tasks. [17] We experiment with this model in addition to our Object Detection + LSTM model, as this eliminates the need for an object detection model but still incorporates time dependencies. The model architecture we chose consists of three Convolutional LSTM layers with batch normalization and two fully connected layers. The model takes 50 sequential screenshots as a training sample for input and the label is the keypress of the last screenshot in the sequence. The output of the model is the predicted keypress, taking into account the environment in the last 50 frames.

B. Model Testing

Model testing is the process where the performance of a fully trained model is evaluated on a testing environment. Models are tested by a script that takes in the game frames as input and and passes the pre-processed images as input to the the trained neural network models. The output of these models is the predicted keypress for the screenshot. The predicted keypress is then simulated via a Python script to facilitate driving the car while the game is running, this involves explicit checks for behaviors that we expect our model to follow.

1) Biasing Model Predictions: During model testing, we observe many models moving erratically by trying to turn or brake too frequently, rather than moving forward most of the time. To mitigate this effect, we apply a final weight layer to the model's probability predictions. By applying lower weights to turning and braking keys compared to forward keys, we can ensure that a model only turns when it is very confident in that prediction. The application of this weight layer also helps bias the model's prediction distribution towards a distribution more similar to a human driver who goes straight most of the time and occasionally turns or brakes. These weights are tuned manually for each model while testing in the game. In the Xception model we see significant improvements in the model's control and speed, and much smoother driving overall when these weights are used.

V. Experimental Results

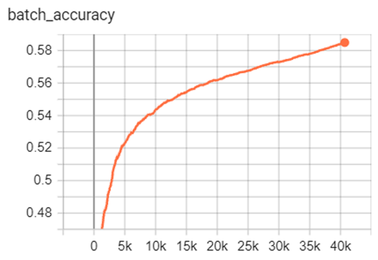
We recorded and documented the results of training various different models in the form of accuracy and loss graphs using TensorBoard while running the training in batches. Further, to benchmark the performance of various algorithms our plan is to use some qualitative judgements while observing the driving pattern of the respective models. These patterns may be number of collisions with pedestrians, number of collisions with other vehicles, number of collisions with objects on and around the road, time needed to reach a destination etc.

A. Training Results

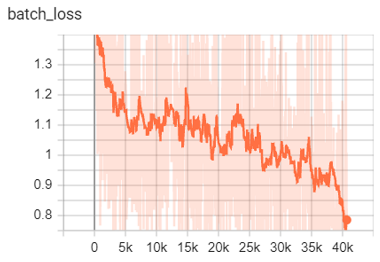
Summarized below are our results of training various different models against their accuracy and loss over the epochs. Each of these observation is based on training a model on batch size of 32 (exception one, ResNet50 trained on a batch size of 16) and a learning rate (lr) of 0.0001.

1) AlexNet: AlexNet was one of initial models that we started training after setting up the game environment. After training for 10 epochs, this model plateaus quickly at a validation accuracy of 44.84% and at a loss of 1.412. We discontinued training this model and started looking for more complex versions that can give us better results.

2) AlexNetV2: AlexNet training accuracy and loss plateaued early, so subsequently we switched to AlexNetV2, which has two fully connected layers added to standard AlexNet model. This was done to increase the complexity of the model and increase the likelihood the network could model the function for this task. After training it for 14 epochs we reached a training accuracy of 58.5% and 0.8 final training loss. See Fig. 5 for AlexNetV2 training results.



(a) Accuracy



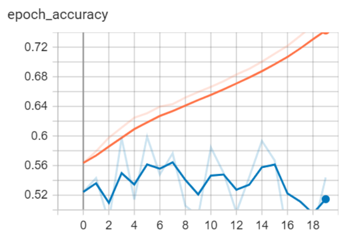
(b) Loss

Fig. 5: Training results for AlexNetV2

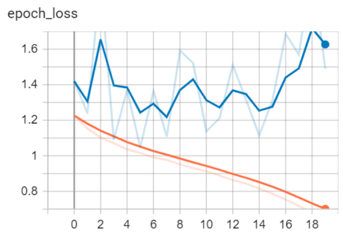
3) InceptionV3: We trained the IncpetionV3 model in parallel with AlexNetV2. The InceptionV3 model performed significantly better than our previous AlexNet models, achieving a validation accuracy of 59.77% and a final validation loss of 1.277. See Fig. 6 for InceptionV3 training results.

4) Xception: The Xception model achieved similar results to the InceptionV3 models in terms of training metrics, but we observe significantly better performance while testing in game. After training the model for a total of 13 epochs we were able to achieve a validation accuracy of 59% and a validation loss of 1.233. See Fig. 7.

5) ResNet50: ResNet50 is one of the more advanced models that we experimented with. This model is deeper than Xception and thus requires more training. As of now for a batchsize of 16, this model trained for 20 epochs and gave an accuracy of 62.35%. In the future we plan to train this model further.

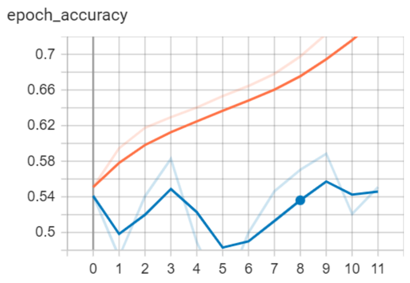


(a) Accuracy

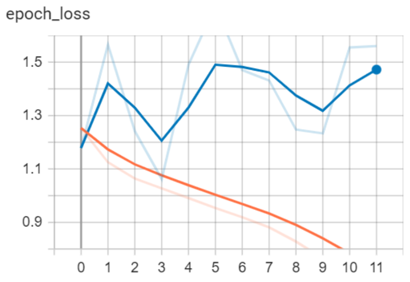


(b) Loss

Fig. 6: Training and validation results for InceptionV3



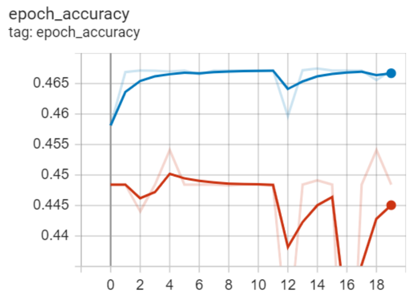
(a) Accuracy



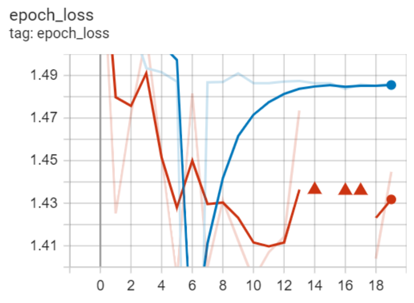
(b) Loss

Fig. 7: Training and validation results for Xception

6) LSTM: We used an LSTM model to incorporate temporal dependencies when predicting the correct keypress. Our LSTM model was composed of four LSTM layers and each was followed by one dropout layer. The training results for 18 epochs is shown in 8. We observe poor performance in terms of validation accuracy and loss. We were able to achieve a max validation accuracy of 48.4% and min validation loss of 1.394. We see that the LSTM model combined with object detection feature vectors did not work well for this task. The main reason is that for the self-driving task, the position of road line is important for driving decisions. However, the object detection model only detects positions of obstacles, so our model receives no information about where the road is. Therefore keypresses within the training data corresponding to attempts to stay on the road or within a lane are not understood by the model. The other reason is that our data is imbalanced, which makes the model hard to learn when the environment is more complicate.



(a) Accuracy



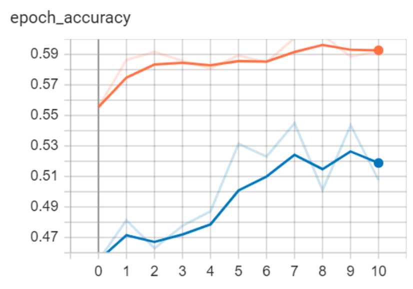
(b) Loss

Fig. 8: Training and validation results for LSTM model after 20 epochs. The orange line shows the training result, and the blue line shows the validation result.

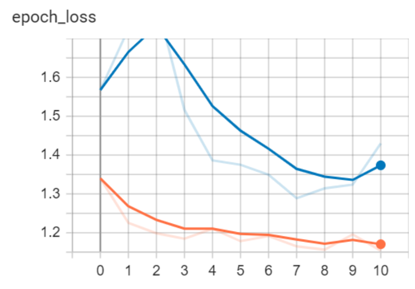
7) ConvLSTM: In the ConvLSTM model, we observe a much more steady increase in validation accuracy throughout training than we have seen in previous models. Validation loss also seems to steadily decrease for the last 9 epochs, until we see an increase at the 10th epoch and overfitting begins. We stop model training at this point. We achieve a max validation accuracy of 54.5% and a min validation loss of 1.324. Figure 9 shows the training results of this model.

For the in-game evaluation, we define the following metrics for more quantitative judgements for performance of the model. We test each model in the same environment (same duration, destination, lighting, route, car, etc.) and evaluate it based on these defined metrics.

- Off road – Number of times the car goes onto the sidewalk
- Back on road – Number of times it comes back on the road after going off-road
- Collisions – Number of collisions with other vehicles or walls
- Recovery after collision – Number of times the car recovers from a collision and continues to drive around without human intervention
- Random reverses – Number of random reverses



(a) Accuracy



(b) Loss

Fig. 9: Training and validation results for ConvLSTM model after 20 epochs.

- Random U-turns – Number of random U-turns when there is no waypoint
- Random turns – Number of random turns while following a waypoint
- Destination reached – The model reaches the destination using a waypoint

B. *In-Game Model Comparison Summary*

- All models are prone to go off road at the current training levels. However, Xception returns to road the most when compared to all the other models.
- AlexNet and AlexNetV2 have worst performance in terms of number of collisions.
- Most trained models did not make any reverses randomly, however Xception, after recently being trained to reverse out of accidents, did make some random reverses.
- ResNet model would make the most number of random turns out of all models.
- No models have been able to navigate to a destination yet following the directions on the on-screen map. This might be because the map is too small on the screenshots.

C. *Summary of Results*

After all the experiments, we observe top training validation results with the InceptionV3 and Xception model which tells us that we need a model with a large number of parameters. We also observe that the improved version of our first model AlexnetV2 comes next in the list of top training validation models. It even outperforms our temporal models with time-frame screenshots. This makes the LSTM and ConvLSTM the worse performing models out of all these and we can

Metrics for Evaluation	**Models**					
	Xception Regular	***Xception Waypoint***	***ResNet Regular***	***ResNet Waypoint***	***AlexNet V2 Regular***	***Alexnet Waypoint***
off road	5	2	5	4	6	6
Back on road	4	1	3	2	4	5
Collisions	5	8	10	12	14	10
Recover from collisions	3	3	4	2	6	4
Random Reverses	2	0	0	0	0	0
Random U Turn	1	1	4	1	1	2
Random Turns	1	2	0	2	0	1

TABLE I: Model Evaluation Results

Models	**Validation Accuracy**	**Validation Loss**
AlexNet	44.84%	1.483
AlexNetV2	55.80%	1.426
InceptionV3	59.32%	1.277
Xception	59%	1.233
Object Detection and LSTM	46.72%	1.535
ConvLSTM	54.50%	1.324

TABLE II: Model performances comparison.

improve these models by incorporating the road position as a feature in the model. Additionally, we found that InceptionV3 and Xception also both perform better during testing in game as compared to all other models. They both learn to avoid some obstacles. However, Xception slightly outperforms InceptionV3 as it has better control over driving and tries to stay within the lane. This performance difference might be due to more efficient use of parameters in the Xception model as they both have the same number of parameters. The Xception model also has better control because of an additional biasing weight layer we added to the model's predictions.

On the other hand we compared the temporal object detection model and we observed the model recognises objects but not the road. It turned out that we need features of the position of the road as well for the model to identify the path as currently the model drives in any direction. In future we can add the road position feature along with other objects.

Finally we also experimented with a larger image size of 480x360 as none of our models is able to follow GPS. We hypothesized that this might be because the map showing the directions is too small at the current image size. But we observed the model performance decreases with larger image size.We see high validation accuracy with the AlexNet model, but when testing in game we notice very poor performance. This is likely because of the resolution distortion that occurs when images are scaled, making it more difficult to delineate objects on a screenshot.

VI. Limitations

The methodologies and techniques used in this project are quite promising, however this system has it's own set of limitations. Broadly these limitations can be categorized as following:

A. Environment related limitations

The limitations of the virtual environment are one of the key impeding factors in development of an accurate model. GTA being a commercial game, although provides a great simulation environment, is not intended for academic and research purposes. This poses certain challenges, for e.g. the speed of the car can not be determined and can not be used by our models for training. Not being able to identify the speed of the car affects the driving intelligence of the model greatly especially around turns & corners.

B. Data related limitations

These are the limitations directly associated with the data collection and data cleanliness. The collected data is unbalanced and often times when collecting frames some unclean data sticks around. This negatively affects the model accuracy while testing. For e.g. a data set collected for driving the car during the day will make the agent biased towards driving in clear broad day light.

C. Resources related limitations

These limitations are attributed to what resources / technology can we use and how much we can spend. With large amounts of data, comes the problem of storing and transferring the data in a safe and fast method. The storage space requirements are above 90GB and transfer times are low. Further more, computing these large data sets require GPUs with very large memories. For e.g. to train the Xception, it required a GPU with memory above 40GB to run batches of size 32. Decreasing the batch size slows down the training significantly and increasing the size requires large resources. These GPU enabled instances on the cloud are expensive which therefore limits our ability to try and test the models properly.

VII. Conclusion

From the current results and analysis of the models, it can be safely concluded that this project serves as a good proof of concept for autonomous vehicle testing in simulated environments. Although there are some limitations to what can be done at the present moment, the project itself has wide scope for expansion and further research.

VIII. Future Work

In the future we would like to tap the potential this project has to become an educational product for the companies that are trying to step into autonomous vehicle industry. There are startups like TuSimple, which are trying to build industrial level AI to drive vehicles based off of a similar approach of training models in a simulated environment initially. We will try to reach out to other people and find out solutions to each of the limitations we faced throughout, one by one and try to bring the project closer to reality where our trained model could be used in any any environment and be able to navigate around with little to no new training at all.

References

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[2] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2818–2826, 2016.

[3] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1251–1258, 2017.

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[5] Dean A Pomerleau. Alvinn: An autonomous land vehicle in a neural network. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA ARTIFICIAL INTELLIGENCE AND PSYCHOLOGY …, 1989.

[6] Urs Muller, Jan Ben, Eric Cosatto, Beat Flepp, and Yann L Cun. Off-road obstacle avoidance through end-to-end learning. In *Advances in neural information processing systems*, pages 739–746. Citeseer, 2006.

[7] Brody Huval, Tao Wang, Sameep Tandon, Jeff Kiske, Will Song, Joel Pazhayampallil, Mykhaylo Andriluka, Pranav Rajpurkar, Toki Migimatsu, Royce Cheng-Yue, et al. An empirical evaluation of deep learning on highway driving. *arXiv preprint arXiv:1504.01716*, 2015.

[8] Why Video Games. Play and learn: Using video games to train computer vision models.

[9] Mark Martinez, Chawin Sitawarin, Kevin Finch, Lennart Meincke, Alex Yablonski, and Alain Kornhauser. Beyond grand theft auto v for training, testing and enhancing deep learning in self driving cars. *arXiv preprint arXiv:1712.01397*, 2017.

[10] Yann LeCun, Patrick Haffner, Léon Bottou, and Yoshua Bengio. Object recognition with gradient-based learning. In *Shape, contour and grouping in computer vision*, pages 319–345. Springer, 1999.

[11] Jason Brownlee. How do convolutional layers work in deep learning neural networks? *Machine Learning Mastery*, 2020.

[12] Zhong-Qiu Zhao, Peng Zheng, Shou-tao Xu, and Xindong Wu. Object detection with deep learning: A review. *IEEE transactions on neural networks and learning systems*, 30(11):3212–3232, 2019.

[13] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems*, 28:91–99, 2015.

[14] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[15] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

[16] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-first AAAI conference on artificial intelligence*, 2017.

[17] Xingjian Shi, Zhourong Chen, Hao Wang, D Yeung, W Wong, and WC Woo. Convolutional lstm network: A machine learning approach for precipitation nowcasting. arxiv 2015. *arXiv preprint arXiv:1506.04214*.