CITY, UNIVERSITY OF LONDON

MSC IN DATA SCIENCE

PROJECT REPORT

2020

Evaluation of self-driving cars using CNNs in the rain

Daniel Sikar

Supervised by: Artur Garcez

December 21, 2020

Declaration of Authorship

By submitting this work, I declare that this work is entirely my own except those parts duly identified and referenced in my submission. It complies with any specified word limits and the requirements and regulations detailed in the assessment instructions and any other relevant programme and module documentation. In submitting this work I acknowledge that I have read and understood the regulations and code regarding academic misconduct, including that relating to plagiarism, as specified in the Programme Handbook. I also acknowledge that this work will be subject to a variety of checks for academic misconduct.

Signed:

Daniel Sikar

Abstract

This project investigates the effect of rainy images on the performance of end to end convolutional neural networks (CNNs) applied to self-driving cars, using public domain real-world datasets, as well as data synthetically generated by the Unity 3D gaming engine.

CNNs have become state-of-the-art in supervised image classification tasks, and are used for regression tasks in both *pipelined* and *end-to-end* neural network architectures. the former relying on a fusion of image and sensor data the latter on images alone, to determine steering and movement output. However, CNNs applied to image classification have been shown to lack robustness when, given noisy images, in some cases a single pixel change, are known to output incorrect results that for humans would have been obviously the same as the denoised versions. As deep learning models, be it CNNs, Recurrent Neural Networks or Deep Reinforcement Learning Networks rely increasingly on computer vision for robotics in general, and for self-driving cars in particular, it is important to evaluate the effect of noisy data, such as images containing rain, used by such models. This study, as far as current literature searches show, is the first of its kind, and provides a novel testing environment that is able to subject self-driving CNN architectures to different rain levels, which may inform design decisions relating to safety in self-driving, as well as robotics relying on computer vision, as well as any Machine Learning/Artificial intelligence application relying on game engines for training and testing data, where the trained model is to be deployed in environments subject to image noise.

Keywords: autonomous vehicles, convolutional neural networks, rain noise

Contents

Declaration of Authorship i

Abstract ii

1 Introduction and Objectives 1

1.1 Background . . . 1

1.2 Aims and Objectives . . . 2

1.2.1 Beneficiaries . . . 2

1.3 Introduction to Methods and Work Plan . . . 2

1.3.1 Changes in Methods and Work Plan . . . 3

1.4 Structure of the Report . . . 3

2 Context 5

2.1 Related Work . . . 5

2.2 Self-driving systems . . . 6

2.3 Network Training . . . 7

3 Methods 10

3.1 Datasets . . . 10

3.2 Keras . . . 10

3.3 tcpflow . . . 10

3.4 Git . . . 11

3.5 SDSandbox and the Unity Game Engine . . . 11

3.6 SD Sandbox and Udacity Driving Simulator Comparison . . . 11

3.7 Adding rain to images . . . 12

3.8 Data Augmentation and Pre-Processing . . . 13

3.9 Identifying rainy images with Amazon Mechanical Turk . . . 14

3.10 NVIDIA *End to End Learning for Self-Driving Cars* Network Architecture . . . 15

3.11 Alternative CNN Models . . . 17
3.12 Training Environments . . . 17
3.13 Training . . . 17
3.14 Evaluation . . . 17
3.15 Creating Videos . . . 18
3.16 Steering data plots . . . 18

4 Results **19**
4.1 Outputs . . . 19
4.2 Datasets . . . 19
4.3 Locating rainy sections with Mechanical Turk . . . 20
4.4 Generating Synthetic Datasets . . . 21
4.4.1 Generated Track . . . 21
4.4.2 Generated Road . . . 23
4.5 Training self-driving models . . . 23
4.6 Testing self-driving models without rain . . . 25
4.7 Adding rain . . . 27
4.8 Evaluation of self-driving cars using CNNs in the rain . . . 28
4.8.1 Realtime predictions for best performing nvidia1 model . . . 29
4.8.2 Realtime predictions for best performing nvidia2 model . . . 31
4.8.3 Synthetic dataset predictions . . . 33
4.9 Modifications to original source codes . . . 37

5 Discussion **38**

6 Evaluation, Reflections and Conclusions **42**

Bibliography **44**

A RPMI Project Proposal Proposal **A1**

B Methods **B1**
B.1 Generating graphs . . . B1
B.2 Network architectures for future reference . . . B1
B.2.1 ResNet . . . B1

B.2.2 GoogleLeNet . . . B1
B.2.3 VGGNet . . . B2
B.3 Cleaning SDSandbox data . . . B2
B.4 Degradation . . . B2
B.5 Running the car simulator . . . B2
B.5.1 Installing the source code . . . B3
B.5.2 Running the Unity Hub application image . . . B3
B.5.3 Adding the SDSanbox project to Unity Hub . . . B3
B.5.4 Running the driving simulator . . . B3
B.5.5 Recording one lap of training data . . . B4
B.6 Network monitoring and debugging with tcpflow . . . B4
B.7 Generating Plots . . . B5
B.7.1 Steering angle comparison . . . B5
B.8 Datasets . . . B5
B.8.1 Ford AV Dataset . . . B5
B.8.2 Ford AV Dataset . . . B6
B.8.3 Kitti . . . B7
B.9 Automold . . . B7
B.10 Development Environments . . . B7
B.10.1 Intel DevCloud . . . B7
B.11 Unity . . . B8
B.12 Tensorflow and Keras . . . B9
B.12.1 Recording one loop around a track . . . B9
B.12.2 Running simulator predictions . . . B9
B.13 ROS . . . B10
B.14 Correspondence with supervisor . . . B10
B.14.1 Correspondence with Artur Garcez 1 . . . B10
B.14.2 Correspondence with Artur Garcez 2 . . . B10
B.14.3 Correspondence with Artur Garcez 3 . . . B11
B.15 Correspondence with authors . . . B12
B.15.1 Correspondence with Urs Muller 1 . . . B12
B.15.2 Correspondence with Urs Muller 2 . . . B13

B.15.3 Correspondence with Ankit Vora B13
B.15.4 Correspondence with Maxime Ellerbach B15

C Network architectures **C1**
C.1 NVIDIA baseline architecture C1
C.2 NVIDIA 0201207201157_ nvidia_ baseline.h5 . . . . . C3
C.3 NVIDIA1 architecture C4
C.4 NVIDIA2 Architecture C6
C.5 Alexnet C7
C.6 Training and testing networks C9

D Deliverables **D1**

E Results **E1**
E.1 Ford AV Dataset steering angles E1
E.1.1 Ford AV Dataset E1
E.2 Audi E2
E.3 Unity3D changing sky hue E4
E.4 Udacity E4
E.5 Training and Testing Log E5
E.5.1 Run 3 . . . . . E5
E.5.2 Run 4 - E5
E.5.3 Run 5 - 20201102081239 . . . . . E5
E.5.4 Run 5 - 20201102090041_nvidia2 . . . . . E6
E.5.5 Run 6 - 20201102094552_nvidia1 . . . . . E6
E.5.6 Run 7 - 20201102090041_nvidia2.h5 . . . . . E6
E.5.7 Run 8 - 20201102134802_nvidia2.h5 . . . . . E6
E.5.8 Run 9 - 20201102210514_nvidia2.h5 . . . . . E7
E.5.9 Run 10 - 20201103211330_nvidia2.h5 . . . . . E8
E.5.10 Run 11 . . . . . E9
E.5.11 Run 12 . . . . . E9
E.5.12 Run 13 . . . . . E9
E.5.13 Run 14 . . . . . E10

E.5.14 Run 15 . . . . . E10
E.5.15 Run 16 . . . . . E11
E.5.16 Run 17 . . . . . E11
E.5.17 Run 18 . . . . . E12
E.5.18 Run 19 . . . . . E12
E.5.19 Run 20 . . . . . E12
E.5.20 Run 21 . . . . . E13
E.5.21 Run 22 . . . . . E13
E.5.22 Run 23 . . . . . E13
E.5.23 Run 24 . . . . . E13
E.5.24 Run 25 . . . . . E13
E.5.25 Run 26 . . . . . E14
E.5.26 Run 26 . . . . . E14
E.5.27 Run 27 . . . . . E14
E.5.28 Run 28 . . . . . E14
E.5.29 Run 29 . . . . . E14
E.5.30 Run 30 . . . . . E15
E.5.31 Run 31 . . . . . E15
E.5.32 Run 32 - 20201117154210_nvidia_baseline.h5 . . . . . E15
E.5.33 Run 33 . . . . . E15
E.5.34 Run 34 - 20201117162326_nvidia_baseline.h5 . . . . . E16
E.5.35 Run 35 - 20201120124421_nvidia_baseline.h5 . . . . . E16
E.5.36 Run 36 - 20201120171015_ sanity.h5 . . . . . E17
E.5.37 Run 37 - 20201120184912_sanity.h5 . . . . . E21
E.5.38 Run 38 - 20201123162643_ sanity.h5 . . . . . E22
E.5.39 Run 39 - 20201123162643_sanity.h5 . . . . . E25
E.5.40 Run 40 - 20201124032017_ nvidia2.h5 . . . . . E26
E.5.41 Run 41 - 20201120171015_ sanity.h5 . . . . . E28
E.5.42 Run 42 - 20201120171015_ sanity.h5 . . . . . E28
E.5.43 Run 43 - Record triple video E29
E.5.44 Run 44 - Intensity x4 . . . . . E30
E.5.45 Run 45 - Intensity x8 . . . . . E30

E.5.46 Run 46 - 20201203164029_ nvidia1.h5 . . . . . E30
E.5.47 Run 47 - 20201206171648_ nvidia1.h5 . . . . . E31
E.5.48 Run 48 - 20201206211122_ nvidia1.h5 . . . . . E32
E.5.49 Run 49 - 20201207091932_ nvidia1.h5 . . . . . E32
E.5.50 Run 50 - 20201207111940_ nvidia2.h5 . . . . . E33
E.5.51 Run 51 - 20201207124146_ nvidia2.h5 . . . . . E35
E.5.52 Run 52 - 20201207132429_ nvidia2.h5 . . . . . E36
E.5.53 Run 53 - 20201207133600_ nvidia2.h5 . . . . . E37
E.5.54 Run 54 - 20201207141017_ nvidia2.h5 . . . . . E38
E.5.55 Run 55 - 20201207142129_ nvidia2.h5 . . . . . E41
E.5.56 Run 56 - 20201207170938_ nvidia2.h5 . . . . . E43
E.5.57 Run 57 - 20201207175205_ nvidia2.h5 . . . . . E44
E.5.58 Run 58 - 20201207184329_ nvidia2.h5 . . . . . E48
E.5.59 Run 58 - 20201207185525_ nvidia2.h5 . . . . . E48
E.5.60 Run 60 - 20201207190447_ nvidia2.h5 . . . . . E49
E.5.61 Run 61 - 20201207192309_ nvidia2.h5 . . . . . E49
E.5.62 Run62 - 20201207192948_ nvidia2.h5 . . . . . E51
E.5.63 Run 63 - 20201207193607_ nvidia2.h5 . . . . . E52
E.5.64 Run 64 - 20201207194331_ nvidia2.h5 . . . . . E53
E.5.65 Run 65 - 20201207194331_ nvidia2.h5 . . . . . E53
E.5.66 Run 66 - 20201207193607_ nvidia2.h5 . . . . . E53
E.5.67 Run 67 - 20201207195804_ nvidia2.h5 . . . . . E54
E.5.68 Run 68 - 20201207201157_ nvidia_ baseline.h5 . . . . . E55
E.5.69 Run 69 - 20201102090041_ nvidia2.h5 . . . . . E56
E.5.70 Run 70 - 20201207203451_ nvidia2.h5 . . . . . E58
E.5.71 Run 71 - E60
E.5.72 Run 72 - 20201207091932_ nvidia1.h5 . . . . . E60
E.5.73 Run 73 - 20201207091932_ nvidia1.h5 . . . . . E61
E.5.74 Run 74 - 20201207091932_ nvidia1.h5 . . . . . E61
E.5.75 Run 75 - 20201207091932_ nvidia1.h5 . . . . . E62
E.5.76 Run 76 - 20201207091932_ nvidia1.h5 . . . . . E62
E.5.77 Run 77 - 20201207091932_ nvidia1.h5 . . . . . E63

E.5.78 Run 78 - 20201207091932_ nvidia1.h5 . . . . . E63
E.5.79 Run 79 - 20201207091932_ nvidia1.h5 . . . . . E63
E.5.80 Run 80 - 20201207091932_ nvidia1.h5 . . . . . E64
E.5.81 Run 81 - 20201207091932_ nvidia1.h5 . . . . . E64
E.5.82 Run 82 - 20201207192948_ nvidia2.h5 . . . . . E65
E.5.83 Run 83 - 20201207192948_ nvidia2.h5 . . . . . E65
E.5.84 Run 84 - 20201207192948_ nvidia2.h5 . . . . . E66
E.5.85 Run 85 - 20201207192948_ nvidia2.h5 . . . . . E66
E.5.86 Run 86 - 20201207192948_ nvidia2.h5 . . . . . E67
E.5.87 Run 87 - 20201207192948_ nvidia2.h5 . . . . . E67
E.5.88 Run 88 - 20201207192948_ nvidia2.h5 . . . . . E68
E.5.89 Run 89 - 20201207192948_ nvidia2.h5 . . . . . E68
E.5.90 Run 90 - 20201207192948_ nvidia2.h5 . . . . . E69
E.5.91 Run 91 - 20201207192948_ nvidia2.h5 . . . . . E70
E.5.92 Run 92 - 20201209001926_ nvidia1.h5 . . . . . E71
E.5.93 Run 93 - 20201209221402_ nvidia1.h5 . . . . . E72
E.5.94 Run 94 - 20201207192948_ nvidia2.h5 . . . . . E74
E.5.95 Run 95 - 20201207192948_ nvidia2.h5 . . . . . E75
E.5.96 Run 96 - Two models on Generated Road E76
E.5.97 Run 97 - Four models on Generated Track E76
E.5.98 Run 98 - 20201121090912_ nvidia_ baseline.h5 . . . . . E78
E.6 Outputs E78
E.6.1 Models E78
E.6.2 YouTube videos E80
E.6.3 tcpflow logs E82
E.7 Download links E83

F Source Code Listing **F1**

1 Introduction and Objectives

1.1 Background

The development of self-driving cars, a particular case of autonomous vehicle (AV), is motivated by a number of goals, of a practical, safety, public interest and economic nature. From a practical perspective, the goal is "to transport people from one place to another without any help from a driver" (Lin, Yoon, and Kim, 2020). From a public health perspective, to transform the current approach to automotive safety from reducing injuries after collisions to complete collision prevention (Fleetwood, 2017). From a public interest and economic perspective, AV fleets allow for new shared autonomous mobility business models (Riggs and Beiker, 2019) though shared autonomous electric vehicle (SAEVs) fleets (Loeb and Kockelman, 2019). Shared Autonomous Vehicles (SAVs) have gained significant public interest as a possible less expensive, safer and more efficient version of today's transportation networking companies (TNCs) and taxis.

This perceived superiority to human drivers is attributed to high-performance computing that allows AVs to process, learn from and adjust their guidance systems according to changes in external conditions at much faster rates than the typical human driver (West, 2016).

The self-driving system for AVs can be distinguished by two approaches: the multi-sensor pipeline approach (Grigorescu et al., 2020, Yurtsever et al., 2020) relying on data from radar, sonar, lidar and images, and the end to end approach (Bojarski et al., 2016), relying on images alone.

Although CNNs have been successfully used to solve problems applied to Computer Vision, the robustness of such architectures has been increasingly scrutinised. Su, Vargas, and Sakurai, 2019 proposed an algorithm where a deep neural network, when presented with a images (in this case images from the Krizhevsky, Nair, and Hinton, 2009 dataset) modified with a single pixel change, is capable of "fooling the network", such that an incorrect label is predicted with a high confidence value.

Zhang et al., 2017 debated the need to rethink generalization, by demonstrating how traditional benchmarking approaches fail to explain why large neural networks generalize well in practice. By randomizing target labels, the experiments show that state-of-the-art convolutional neural networks for image classification trained with SGD (stochastic gradient descent) are large enough to fit a random labelling of the training data. This is achieved with a simple two-layer neural network, which presents a "perfect finite sample expressivity" once the number of parameters is greater than the number of data points as often is the case with CNNs. This

poses a challenge for self-driving cars, specially if relying only on images. Since testing real life scenarios is not practical in this study, two options are examined: using public self-driving datasets containing images labelled with steering angles, and using game engines (Cowan and Kapralos, 2014), which are capable of generating labeled datasets and realistic environments where the autonomous vehicle may be tested. The research question is *how do images containing rain-like patterns affect self-driving CNNs trained to output steering predictions*?

1.2 Aims and Objectives

The main aim of this project is to to answer the research question, by evaluating how reliable are self driving cars using end to end convolutional neural network architectures to predict steering given images with rain-like noisy patters. The significance of this type of network is it relies only on raw pixel values, and no additional sensor inputs. To reach this aim, the project objectives are:

- Determine a game engine capable of supporting the experiments
- Select and create appropriate datasets
- Determine software packages able to augment and add rain-like patterns to digital images
- Train self-driving end to end CNN models
- Establish a workflow to modify images to be presented to the network during the running (driving) phase
- Determine a metric for evaluating how well a self-driving CNN is performing with respect to a steering ground truth

1.2.1 Beneficiaries

Autonomous vehicles and robotics increasingly rely on computer vision and in some cases convolutional neural networks. The issue of reliability in computer vision based systems when faced with noisy data, in the case of this study, images containing rain, is one that affects any CNN based system, not only autonomous vehicles, hence, any findings in this work could be beneficial to help, practitioners inform design decisions, and industry approval bodies surveying benchmarking methodologies. The project, once reaching the aims, provides an open framework which may benefit any further research.

1.3 Introduction to Methods and Work Plan

The work plan is to survey the literature for related work and current trends in self-driving CNNs. Followed by researching required tools, datasets and data augmentation techniques.

Next task is to create a development environment for training and testing. Then research existing CNN self-driving models that can be replicated and modified. Models must also be evaluated qualitatively by using the game engine, observing the self-driving vehicle around a simulated track and noting any oversterring and crashes. Once quantitative and qualitative analyses are completed, the process should generated figures required to complete the report.

1.3.1 Changes in Methods and Work Plan

The original work breakdown structure described in appendix A was revised as shown in Figure 1.1.

	Task	Month	June					July				August					September				October				November					December		
		Week number	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52
1	Project Proposal																															
2	Reference tools	Start 22.06.2020																														
3	Literature search																															
4	Literature review																															
5	Toolset selection	Start 13.07.2020																														
6	Dataset selection																															
7	Dataset augmentation																															
8	Replicate existing model	Start 17.08.2020																														
9	Create alternative models																															
10	Evaluation metrics	Start 19.10.2020																														
11	Evaluate models																															
12	Draft report	Start 22.06.2020																														
13	Complete report	Start 16.11.2020																														
14	First round review																															
15	Second round review																															
	Contingency																															
16	Submit final report	Start 21.12.2020																														

FIGURE 1.1: Retrospectively revised project work plan.

1.4 Structure of the Report

The remaining report is structured as follows:

- Chapter 2 provides the critical context, project motivation, methods used, literature survey and analysis. It outlines the current state of research with autonomous vehicles deep neural networks, including ConvNets.
- Chapter 3 contextualises the broader context of self-driving cars and the technologies pushing research further
- Chapter 4 describes results of training networks, together with metrics and video links showing the simulated car models self-driving
- Chapter 5 contains a discussion on what objectives were met, what objective was not met and suggested modifications to meet all objectives.
- Chapter 6 evaluates and reflects on the project, limitations, contributions and ares for further work.

- Appendix A contains the RPMI project proposal
- Appendix B details methods used
- Appendix C contains network architectures used
- Appendix D presents tables containing *deliverables* first put forward to project supervisor
- Appendix E contains training logs and notes on gathering results
- Appendix F contains listings for original and modified (with attribution) codes used in this project
- This document in pdf format and a readme.txt file are submitted in the project submission area. readme.txt file lists SharePoint links for trained models, dataset, source code for training, testing, driving simulator and source code audit files.

2 Context

2.1 Related Work

Adverse weather affecting sensor data including cameras used by AVs was surveyed by Zang et al., 2019. Fu et al., 2017 Cord and Gimonet, 2014, determining that video-camera based ADASs (Advanced driver assistance systems) are becoming ever present in the automotive sector. The trade off being good performance in good weather *versus* bad performance in bad weather, particularly in the rain. The suggested approach is to detect unfocused raindrops on the vehicle's windscreen, using in-vehicle camera images. Then analyse raindrop photo metric properties and rely on image processing to highlight the rain drops. This technique is claimed to improve ADAS behaviour in the rain.

Yoneda et al., 2019 acknowledge the possibility of object recognition performance degradation due to image quality degradation, and suggest careful positioning of camera inside the vehicle within windscreen wiper range. It is also acknowledged that this may not be a possibility if the camera is place on the side or outside of the vehicle to capture images in every direction, the raindrops then being inevitable, concluding that when camera installation is performed, raindrops and a mechanism for removal are required. Research on rain-drop recognition in the captured image is reported.

Kurihata et al., 2005 suggest a method for weather recognition with images being captured from in-vehicle camera able to use a subspace method to asses rain by detecting rain drops on the windshield. *Eigendrops* represent the a form of PCA (principal component analysis), extracting from components from raindrop images in the learning stage. Then template matching is used to detect rain drops. The authors claim the method showed good results with video sequences containing raindrops and promising detection results for rainfall judgement

Webster and Breckon, 2015 state image containing raindrops are prone to distortion as rain has a significant negative performance impact in a wide range of sensing applications based on visual sensing and used in all weather conditions including surveillance and vehicle autonomy. The problem is framed as "robust raindrop detection" as a means of identifying the potential for performance degradation in affected image regions. Raindrop detection is colour video is performed using "extended feature descriptor comprising localised shape".

Garg and Nayar, 2007 state rain produces complex visual effects such as sharp intensity changes, that can severely compromise performance of outdoor vision systems, providing a comprehensive analysis of visual effects of rain and various related factors such as camera parameters, rain properties, brightness.

Related work was therefore found studying the effect of rain on sensors specifically, applicable to multi-sensor fusion models (see next section). All literature surveyed frame the problem in terms being solved with a manual feature extraction solution, in other works, no mention is made of CNNs. No literature related to the effect of rain on CNNs was found.

2.2 Self-driving systems

The "Dave" system, LeCun et al., 2004, was a proof-of-concept, model scale vehicle that used end-to-end learning for obstacle avoidance, based solely on two onboard cameras equipped with analogue video transmitter and a remote computer to process the image data and send back steering control signals via radio. On the remote computer, YUV images (Maller, 2020) from left and right cameras were presented to a network trained to output steering angles. Hours of binocular video images, labelled with a steering angle provided by a human driver, who avoided any obstacles found to be in path, were used to train the network.

Figure 2.1 shows a diagram representing a CNN predicting a steering angle from a single front-facing camera. The NVIDIA PilotNet (Bojarski et al., 2016) was able to learn to drive on local roads with or without road markings, and also operate in areas with unclear markings such as parking lots. Like Dave, this is another example of end-to-end learning, with no intermediate criteria such as lane detection. The authors state such criteria is devised solely to simplify human understanding, but does not necessary translate into system performance. Features are instead extracted automatically by convolutional layers on the network, the fully connected layers then performing the classification or regression task as needed.

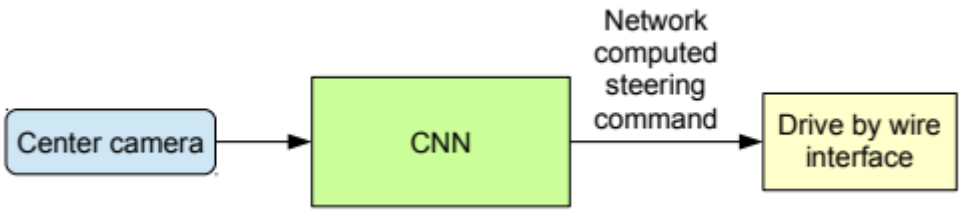

FIGURE 2.1: Diagram of a trained network with a single centre camera input and a steering command output

One weakness of convolutional neural networks is the perceived lack of robustness. Figure 2.2 shows examples of a *one-pixel attack* (Su, Vargas, and Sakurai, 2019). The attackers did have a deep knowledge of the network architecture being exploited, still in this classification task setting it is an undesirable outcome.

Another scheme used for self-driving cars is the multisensor (Grigorescu et al., 2020 and Yurtsever et al., 2020) approach shown in Figure 2.3, where a number of number of input configurations can potentially be used, with of without deep learning, that is with the help of a multilayer neural network with a least one hidden layer. The approach may use classical "rules based" programming to determine steering and motion.

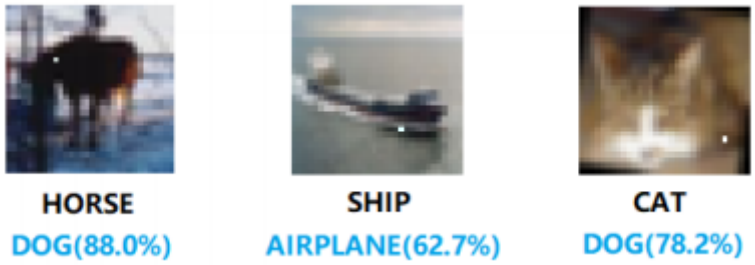

FIGURE 2.2: One-pixel attacks created with the algorithm proposed by Su, Vargas, and Sakurai, 2019 that successfully fooled three types of deep neural networks trained on CIFAR-10 (Krizhevsky, Nair, and Hinton, 2009) dataset. The modified pixel is white, the original class labels are black while the target class labels and the corresponding confidence are blue

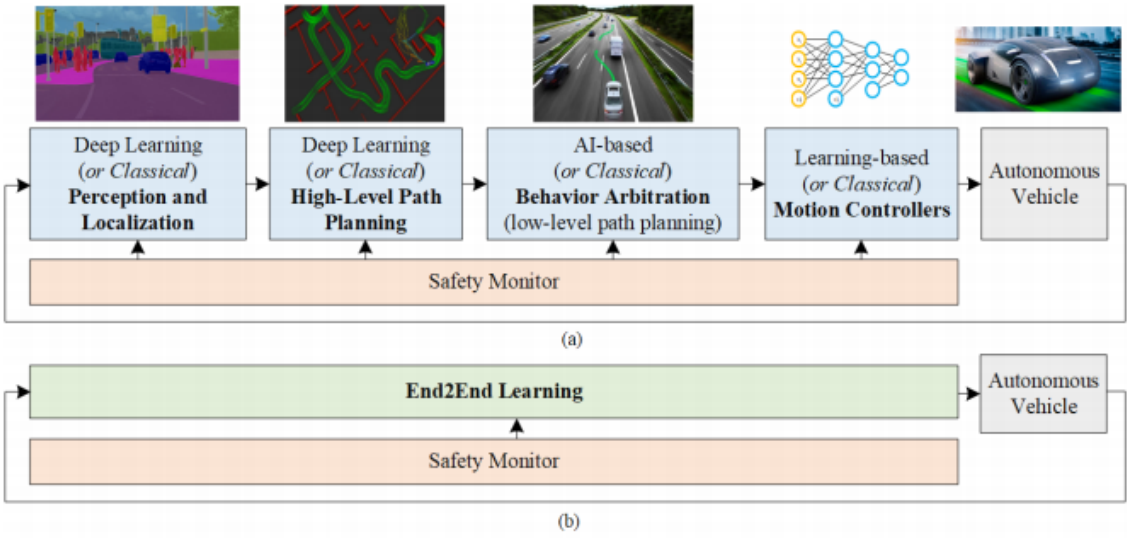

FIGURE 2.3: Diagrams showing the multisensor pipeline approach (a) and the end to end approach (b) as described by Grigorescu et al., 2020

The introduction of any AI-controlled hardware into public spaces also raises a number of societal issues like safety, liability, privacy, cybersecurity, and industry (Taeihagh and Lim, 2018). Other issues include the impact on demand for a human workforce, wages and employment (Acemoglu and Restrepo, 2018).

2.3 Network Training

The properties of multilayer feedforward neural networks as universal function approximators has been studied extensively, especially in the 1980's. Hornik, Stinchcombe, White, et al., 1989 rigorously established that standard multilayer feedforward networks with as few as one hidden layer can approximate specific functions provided sufficiently many hidden units are available, though the issue of how many are required is not addressed. Cybenko, 1989 demonstrated analytically that a feedforward neural network with one hidden layer and any continuous sigmoidal nonlinearity can approximate any function with arbitrary precision provided

that no constraints are placed on the number of nodes or the size of the weights. As such, failures in applications can be attributed to inadequate learning, inadequate numbers of hidden units, or the presence of a stochastic rather than a deterministic relation between input and target. Training will therefore fail due to inadequate learning or hidden unit number choices, or the existence of a stochastic, non-deterministic, relation between input and target.

The goal is to find optimal parameters (weights and biases) and biases for a arbitrary size multilayer, or deep network, because as the depth or width increases, so do the number of parameters.

Changes in network parameter distribution during training lead to a changes in network activation distributions, this is known as Internal Covariate Shift (Ioffe and Szegedy, 2015; Tellaeche and Maurtua, 2014). LeCun et al., 1998; Wiesler and Ney, 2011, suggest a long known method to address the issue by linearly transforming input values to have zero mean and unit variance. To achieve this, the pixel values (maximum value 255) are divided by 255 and have 0.5 subtracted, such that transformed values will have "whitened" zero mean and unit variance. This is done repeatedly throughout the training. By fixing the distribution of the inputs, training speed is expected to improve. Networks trained which such inputs must also be presented with whitened inputs at inference (prediction) time.

The inputs are affected by weights and biases in all the previous layers, changes in the inputs during training will lead to saturation, also known as vanishing gradients (weights and biases becoming very large or small), the effect increasing with network depth (deep networks) . Several schemes have been used to address this saturated regime: adequate parameter initialization (Glorot and Bengio, 2010, Saxe, McClelland, and Ganguli, 2013), small learning rates and Rectified Linear Units (Nair and Hinton, 2010).

The large search space for optimal network parameters can be addressed by a number of optimizers for example error backpropagation (Rumelhart, Hinton, and Williams, 1986) with Stochastic Gradient Descent that can lead to better results in large datasets where there is repetition. LeCun et al., 2012 suggests that examples (inputs) should be shuffled to attain maximum information content but points out this will not apply to data with outliers.

Since more data is better for network training, data augmentation can be used to increase the amount of data available to train the model (Perez and Wang, 2017). This is achieved by drawing additional data from the training data itself. It can help to make the network generalise and avoid overfitting, that is, doing a good job in predicting examples that have been seen but not doing so well with unseen examples. "Hyperparameter", a non-trainable parameter, choices can also impact accuracy. Kernel size, stride padding, activation and loss functions, and optimizers are hyperparameters than can be adjusted for best network performance.

A typical multi-class classifier CNN will have a softmax activation function in the final layer and a cross-entropy loss function. a typical regression model will have a linear activation function in the final layer and a mean squared error loss function. To transform a deep classification model into a deep regression model, the number of class outputs is replaced by the number of required continuous outputs, the softmax activation is replaced by a linear activation and the

cross-entropy loss is replace by mean squared error loss.

Class imbalance (Batista, Prati, and Monard, 2004) has an effect on model's accuracy. A heavily outnumbered class may be difficult to learn. Rare events lead to the creation of imbalanced learning scenarios (Krawczyk, 2016). In the case of self-driving car data, the expectation is that most of steering angle labels will be close to zero (car driving straight). So some form of undersampling of majority classes (if viewing steering angle ranges, for instance, as would be achieved by binning) may help to address the issue.

3 Methods

3.1 Datasets

Labelled datasets, each image labelled with a steering angle, are required to train self-driving CNNs. Five datasets are considered Audi (Geyer et al., 2020), FordAV (Agarwal et al., 2020), Kitti (Geiger et al., 2013), Udacity ("Udacity Self-Driving Car Driving Data 10/3/2016 (dataset-2-2.bag.tar.gz)") and Unity, this last one being synthetic data generated specifically for this project. All provide a labelled dataset in the sense that a steering angle may be obtained for each image. In the case of Udacity simulator and SDSandbox data, the steering angle image label is implicitly stored. IN the remaining datasets, the steering angle may be inferred either by comparing timestamps in plain text log files, or extracting IMU data from rosbag files.

3.2 Keras

The Keras (Chollet et al., 2015) deep learning API (Application Programming Interface) written in Python (Van Rossum and Drake Jr, 1995) and in this case acting as a wrapper around the machine learning library Tensorflow (Abadi et al., 2016) is used to create, train and test models. The framework allows for designing, compiling and fitting models, saving and loading model weights, early-stopping training, saving best models as training progresses and using models to make predictions. The implementation details are abstracted, making for a clean programming interface, and simple to change hyperparameters such as activation functions, kernel size and stride, learning rate, optimizer and loss metric.

3.3 tcpflow

tcpflow (Elson, 2013, Garfinkel and Shick, 2013) is able to capture data packets, transmitted using the TCP (*Transmission Control Protocol* 1981) host-to-host protocol (which provides a process-to-process communication service) and store the data in a human readable format, such that it may be analysed and debugged. For this study, it is used to reconstruct and record data transmissions between SDSandbox and the prediction engine. A practical example is given in B.6. Figure **??** shows a plot of predicted and simulator steering angles. The prediction is for a single image frame sent over the network. As the simulation begins, frames are sent over the network and in this case, captured with tcpflow.

3.4 Git

Git (Chacon and Straub, 2014) is a distributed software version control tool, also known as SCM (Source Code Management). It is used in this project to keep track of changes made to software code. A change is recorded through a *commit*, which generates a unique commit hash. By comparing commits it is possible to determine how code was modified between commits. The ability to track such changes is useful to determine code changes over time, and to recall configurations for an experiment to be repeated. An example of the forensic use of Git can be seen in E.5.36, to narrow down what code generated a well performing model.

3.5 SDSandbox and the Unity Game Engine

SDSandbox (Self Driving Car Sandbox - Kramer, 2020) is a self-driving simulator that uses the Unity (Haas, 2014) game engine to simulate 3d environment car physics, as well as terrain and lighting. SDSandbox adds a user interface with a number of circuits, and functionality to create labelled datasets consisting of images (.jpg files) and steering and throttle data (corresponding .json files). It also allows a model to be driven by a prediction engine. In addition to the Unity simulator (sdsim directory), SDSandbox also provides code (src directory) to train models (train.py) and run inferences (predict_ client.py). A base model is supplied (models.py) that "uses NVidia PilotNet NN topology", with some modifications.

Setting up the SDSandbox simulator and prediction engine is further detailed in Appendix B.5.

3.6 SD Sandbox and Udacity Driving Simulator Comparison

Two driving simulators were trialed, an earlier version of Udacity (Udacity, 2017) and SDSandbox (see 3.5). Both use the Unity game engine. The former had a large user base on account of the accompanying MOOC (Massive Online Open Course), which has since started using the Unreal Engine (Epic Games, 2019) based Carla (Dosovitskiy et al., 2017) sim (short for simulator). SDSandbox is actively developed for the open source Donkey Car autonomous vehicle (DonkeyCar, 2020 and used by the do-it-yourself model-scale autonomous-vehicle community DIY Robocars (DIYRobocars, 2020). The majority of this community uses behavioral learning to train self-driving CNNs (see correspondence with SDSandbox main code maintainer in B.15.4).

Both simulators (Figure 3.1) are equivalent in the sense of acting as a testing environment for self-driving CNNs and also as a training data generator. The difference being SDSandbox is able to generate data using, in addition to manual steering, a self-driving PID (Bennett, 1993) closed loop controller as implemented in PIDController.cs source code file. The process provides feedback to the simulated car, which adjusts itself on the track, based on known geometrical constraints. The feature (option *Auto Drive w Rec*) makes generating training sets less

laborious. The manual option (*Joystick/Keyboard w Rec*) being, a simulated car must be manually steered around the track a number of times. The equivalent Udacity sim option being *TRAINING MODE*.

For inference, Udacity and SDSandbox have equivalent modes. The options being *AUTONOMOUS MODE* and *NN Control over Network*. Output image sizes are 320x160 and 160x120 respectively. The Udacity sim outputs steering angles to a single comma separated values' (.csv) file, as opposed to multiple .json files.

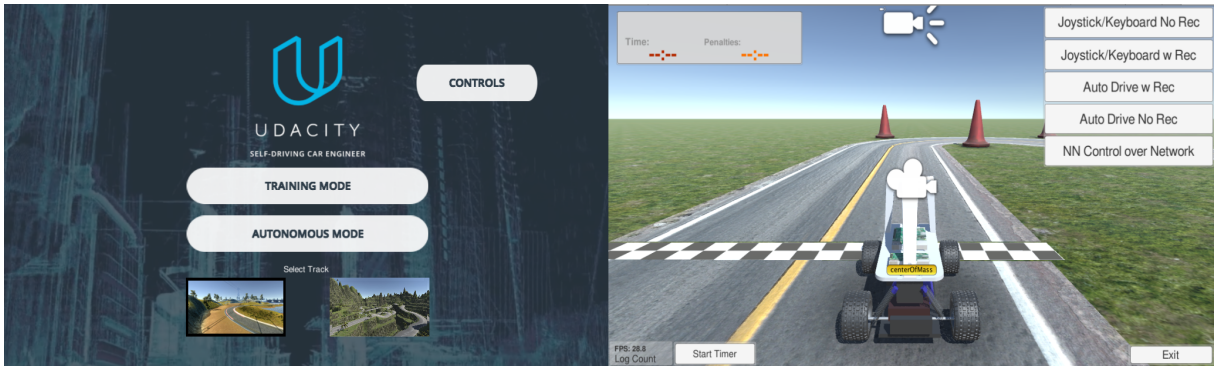

FIGURE 3.1: Left to right: legacy Udacity and SDSandbox Autonomous Driving Simulators

Appendix B contains further notes on obtaining the source code (B.5.1), running the Unity Hub utility (B.5.2), generating training datasets (B.5.5)and running the driving simulator (B.5.4)

3.7 Adding rain to images

At the core of testing self-driving CNNs in the rain is the ability to add rain to images, never seen before by the network, at inference time. Rain is added to images using the Automould (Saxena, 2017) library, in turn using opencv-python, a Python port of OpenCV (Mordvintsev and Abid, 2014), a library for Computer Vision and Machine Learning algorithms. Rain effect is created by adding lines of user defined slant, and random width. The image is then blurred by convolution with a 7x7 normalized box filter (kernel) as shown in equation 3.1, taking the average of all the pixels under the kernel area and replacing the central element on the image (OpenCV, 2020) with the result of the convolution operation (Appendix A, pg 3).

$$K = \frac{1}{49}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \tag{3.1}$$

The kernel acts as a low pass filter, removing high frequency content (such as noise and edges). The image is then moved from RGB (red, green, blue) to HSL (hue, saturation, lightness) space, where the lightness values are multiplied by 0.7. This has the effect of making the image darker. The process is completed by moving the image back to RGB space. Rain types are *drizzle* (a.k.a. "light", default, code checks for "heavy" and "torrential", any other non-empty string is considered drizzle/light rain), *heavy*, and *torrential*. Slant is defined as an angle in degrees from 0 to plus or minus 20.

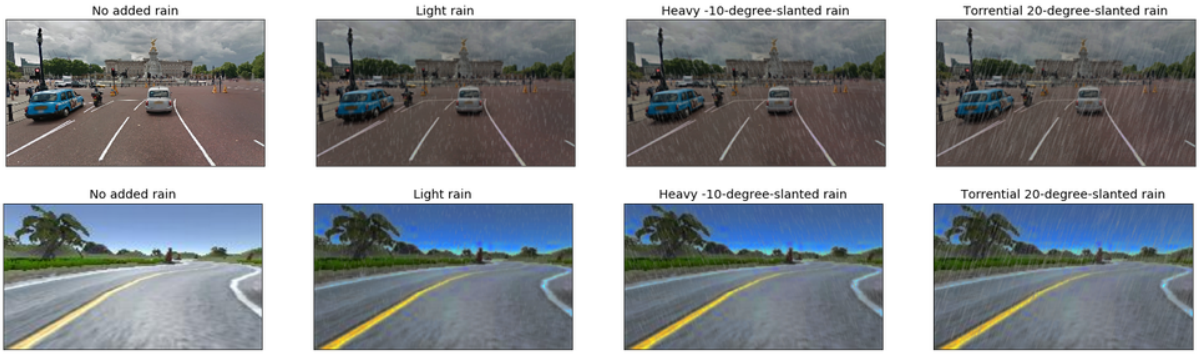

FIGURE 3.2: Left to right: images

Figure 3.2 shows on the top row, a real-life image and on the bottom row, an image created by SDSandbox simulator. The columns from left to right have: no rain added, straight falling (0 degree slant) light, heavy with a -10 degree slant added and torrential rain with a 20 degree slant added. During inference (running steering predictions) the image is (in addition to same pre-processing, performed during training time) processed such that it will contain rain never seen before by the network.

3.8 Data Augmentation and Pre-Processing

As discussed in 2, data augmentation can be used to reduce overfitting, by increasing the amount of training data. The data augmentation library used in this study is adapted from Shibuya, 2016. Critical to the design of computer vision CNNs is the geometry of the input image. The adapted augmentation library (Augmentation.py) first resizes the image to the expected size of acquired image in the original network design. This enables code re-usability, where different networks, using different image sizes can be tested using the same code. Increase by a factor of 1.6 in this study. The sequence shown in figure 3.3 presents image array sizes at every step, from top left to bottom right, the image is loaded and resized (to 320 width x 160 height pixels in this case), augmented with random horizontal flip (with corresponding steering negated), random horizontal shifts (with steering adjusted by adding or subtracting 0.002 degrees per shifted pixel, random addition of shadows and random modification of brightness, concluding the augmentation part.

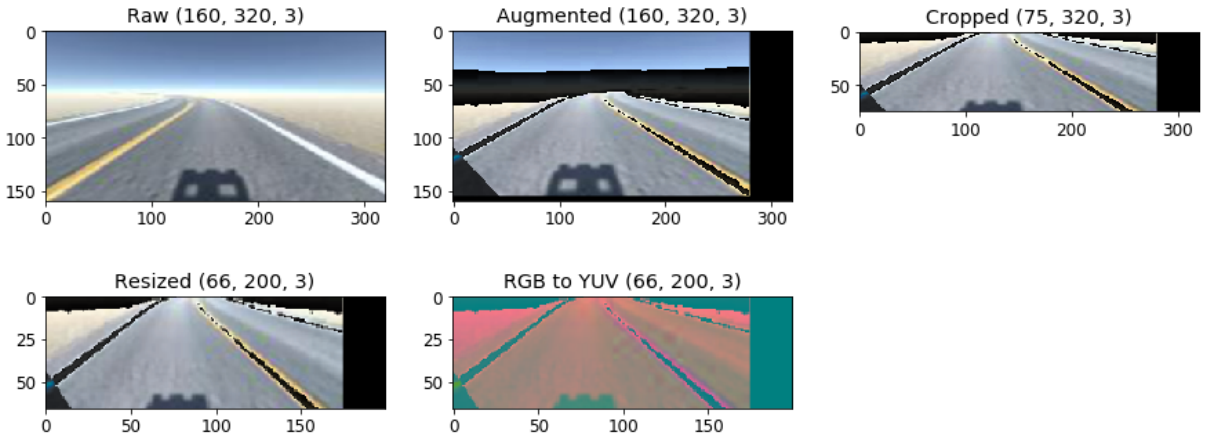

FIGURE 3.3: Stages of image augmentation (DriveNet and NaokiNet image geometries) and pre-processing with corresponding image array dimensions of each step

For the pre-processing part, the image is then cropped to remove car and horizon, resized to the dimensions determined by network design (200x66 pixels for baseline network). Crop parameters (amount of image to be removed from top and bottom sections) is network dependant and set in a configuration file (conf.py). In the example shown, 25 rows of pixels were removed from the bottom and 60 rows of pixels were removed from the top, resulting in a 320 x 75 pixel image. The image in then resized to the geometry to be presented to network, in this case 200 x 66, which is the geometry used by NaokiNet and DriveNet. Finally the image is moved from RGB to YUV space. In the RGB scheme, each pixel is represented by three channel intensities of red, green and blue. In YUV, also referred to as YCbCr (Maller, 2020) space, each pixel is represented by Y (luma), U (Cb - luminance value subtracted from red channel) and V (Cr - luminance value subtracted from red channel). It is a "lossy" process which degrades the data, and originally developed for colour to black and white television backward compatibility. Moving the image from RGB to YUV space has been demonstrated to give "better subjective image quality than the RGB color space", being better for computer vision "implementations than RGB due to the perceptual similarities to the human vision" (Podpora, Korbas, and Kawala-Janik, 2014). This scheme was used in Dave, the Autonomous off-road vehicle (LeCun et al., 2004) which used end to end learning.

3.9 Identifying rainy images with Amazon Mechanical Turk

Amazon Mechanical Turk Crowston, 2012 is an online marketplace where jobs can be outsourced using the *crowsourcing* model (Vukovic, 2009) which delivers a scalable workforce that may be adjusted in size as required. Mechanical Turk has been referred to as *artificial artificial intelligence* (Dai, Weld, et al., 2011) where a task requiring human intelligence level, such as labelling an image, can be assigned to a real person, in lieu of writing a computer algorithm capable of doing so. Mechanical Turk, launched in 2005, was instrumental in labelling the

Imagenet (Deng et al., 2009) dataset which lead to advances in computer vision image recognition with a number of deep (TODO explain "deep" at some point) network architectures (Krizhevsky, Sutskever, and Hinton, 2012, He et al., 2015, Szegedy et al., 2014 and Simonyan and Zisserman, 2015).

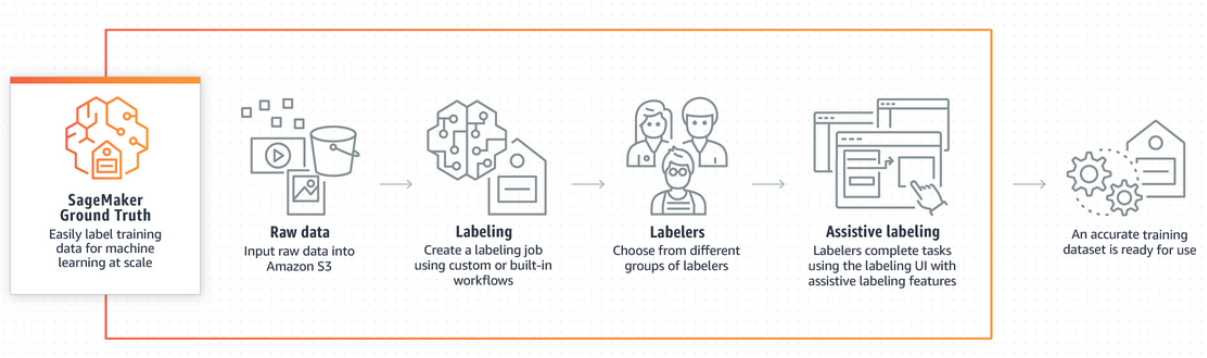

FIGURE 3.4: Amazon SageMaker Ground Truth workflow: a dataset is placed on Amazon S3, labeling job is created, workers label data using a labeling user interface resulting in a labeled dataset

For this study Amazon SageMaker Ground Truth (Amazon Web Services, 2020a) was used. It is a wrapper for Mechanical Turk, and facilitates the process of setting up the image labelling job, especially the UI (user interface) that the labeler workforce must used. The aim here is to find segments of real life footage that may contain rain. The Ford AV (Agarwal et al., 2020) has sections labeled as "cloudy" so may contain rain. 100 images, chosen at regular intervals, the expectation being this may help narrow down the location of rainy images, if any. To test the accuracy of the service, 5 images with rain are added to the dataset. The expectation being labels for these 5 images will be "rain". The unlabelled dataset is stored on AWS S3 (Amazon Web Services, 2020b). An "Image Classification (Single Label)" "Task type" is chosen, i.e. workers will verify if the proposed label "Rainy image" is correct. The process workflow is shown of Figure 3.4.

3.10 NVIDIA *End to End Learning for Self-Driving Cars* Network Architecture

The NVIDIA network (C.1) architecture, also known as *PilotNet* (Bojarski et al., 2020) is the baseline self-driving architecture used in this study. This comprises 3 (RGB channel) x 66 (height) x 200 (width) size input layer corresponding to an image presented to network. The input layer is followed by: a normalization layer generating a 3 channel 66 x 200 layer, a 24 5x5 kernel convolutional layer generating 31x98 feature maps, a 36 5x5 kernel convolutional layer generating 14x47 feature maps, a 48 5x5 kernel convolutional layer generating 5x22 size feature maps, a 64 3x3 kernel convolutional layer generating 3x20 size feature maps, a 64 3x3 kernel generating 1x18 size feature maps, a 1164 neuron flattened layer fully connected to a 100 neuron layer, fully connected to a 50 neuron layer, fully connected to a 10 neuron layer, fully connected to a one neuron output layer. The first 3 convolutional layers have stride equal 2

and the last 2 convolutional layers have stride equal 1. Padding is equal to zero, resulting in smaller feature maps with every convolutional layer. The size of feature maps generated by each convolutional layer is determined with equation 3.2 as described in Dumoulin and Visin, 2018:

$$n_{out} = \left\lfloor \frac{n_{in} + 2p - k}{s} \right\rfloor + 1 \tag{3.2}$$

where n_{out} is output size of convolved feature map, n_{in} is input size of image or feature map, p, k and s are padding, kernel and stride size respectively.

For example, to determine size of feature maps in the first convolutional layer $n_{out} = \lfloor (66 + (2 \times 0) - 5)/2 \rfloor) + 1 = 31$, $n_{out} = \lfloor (200 + (2 \times 0) - 5)/2 \rfloor + 1 = 98$. The Keras (Chollet et al., 2015) machine learning framework is used to build the model. The calculation for number of trainable parameters (weights) in a convolutional layer is given in equation 3.3:

$$n_p = m \times n \times d \times k + d \tag{3.3}$$

where m and n are the convolutional kernel dimensions, d is the number of feature maps in the current layer and k is the number of feature maps in the previous layer. Note d is added to account for *bias* term. For example, the number of trainable parameters for the first convolutional layer in C.1 is given by $n_p = 5 \times 5 \times 24 \times 3 + 24 = 1,824$. The number of trainable parameters in a fully connected layer is given by equation 3.4:

$$n_p = m \times n + n \tag{3.4}$$

where m is the number of inputs in previous layer and n is number of neurons in current layer. Note second term n is added to account for *bias* term. For example, for the first fully connected layer in C.1 the number of parameters is given by $n_p = 1164 \times 100 = 1,342,092$, where the inputs are values from previous flattened layer, that is, a vectorized representation of the last convolutional layer feature maps. The total number of trainable parameters generated by Keras is 1,595,511. This does not agree with the total given by Bojarski et al., 2016, their "network has about (...) 250 thousand parameters". Total memory required to store parameters assuming 32-bit float data type, is 6MB given by $1,595,511 \times 4 \div 1,024 \div 1,024$.

Since the NVIDIA model is based on work by Krizhevsky, Sutskever, and Hinton, 2012, "dropout" (Hinton et al., 2012) is assumed to have been used, with 50% dropout probability on every layer. Dropout removes neurons randomly and helps prevent overfitting, where a model performs well on training data and poorly on testing data, by preventing co-adaptation, resulting in neurons that can detect useful features in the multitude of contexts it must operate, and help the network produce the correct answer. As per correspondence with co-author(B.15 additional training hyperparameters are MSE (loss function) adadelta (optimizer), "learning rate: 1e-4 (but not really used in adadelta)" and 0.25 dropout. Additional assumptions on training include: 128 example batch size, weights initialized from a zero-mean normal distribution with standard deviation 0.01 and biases mostly initialized to 1 as per Krizhevsky, Sutskever, and Hinton, 2012, which applied to this baseline architecture initializes all biases to 1, except in convolutional layers 1 and 5, where biases are initialized to 0. Details in source code src/models.py, nivia_

baseline function.

3.11 Alternative CNN Models

In addition to PilotNet, two other designs are considers, one supplied with the SDSandbox framework, and referred to in this text as *TawnNet*, the other supplied by Shibuya, 2016 and referred to in this text as *NaokiNet*. These can be seen as simplified variants of PilotNet, and a good starting point, as the original NVIDIA paper makes available no public source code. These architectures are shown in C.3 and C.4 respectively. TawnNet being used as-is and NaokiNet being modified after TawnNet to include 5 dropout layers (originally only one after the last convolutional layer, with dropout set to 0.5) after each convolutional layer, with dropout set to 0.1. Additionally, the number of feature maps in the second convolutional layer was changed from 36 to 32.

3.12 Training Environments

Three training environments are used, a local workstation running Ubuntu 18.04 with 31GB RAM and 12-core CPU, City University's shared Camber (City University cloud server) running Ubuntu 16.04 with 125G RAM and 48-core CPU and the Intel's shared DevCloud running Ubuntu 18.04 with 250G RAM and 80-core CPU. None have GPU processors.

3.13 Training

Training can be performed in any of the environments once all the required python modules are installed and data have been downloaded or created. In case of Camber and the Intel DevCloud, jobs are batched. In the local environment SDSandbox python training scripts run from the command line. The neural networks in this project are trained using the Adam optimizer, learning rate set to 0.0001 except where noted, batch size of 64,

3.14 Evaluation

The evaluation can be performed qualitatively using a simulator as described in 3.6 to observe the simulated vehicle self-driving with respect to oversteering and understeering, crashes providing pass/fail metric.

The proposed quantitive evaluation metric is *goodness-of-steer*. In equation 3.5:

$$g_s(p,g) = \frac{\sum_i^N |p(i) - g(i)|}{N} \times n_c \tag{3.5}$$

where p, g are prediction and ground truth arrays, N is the number of predictions and n_c is the normalization constant (25 in example shown, 1 for values not normalized), which in this case is the maximum steering angle for the SDSandbox simulated vehicle. In plain english, g_s is defined as the sum of the absolute value of the difference between prediction and ground truth values, divided by the number of predictions multiplied by a normalization constant, that is the steering error average over all predictions. TODO read up on noise models (Gonzalez)

3.15 Creating Videos

Videos are used in this project for qualitative analysis as well as documentation. Three methods are used to create videos. The first is with the Kazam (Kazam, 2020) desktop screen capture utility, which records the contents of a given display monitor. This generates video captures such as shown in Figure 4.7, noting the video contains only one frame, and the three images shown in the figure represent different parts of the video. The second method was developed for this project (MakeVideo.py) and consists of parsing tcpflow logs, extracting images and predicted steering angles and creating a "side-by-side" video of the simulator being driven by the prediction engine, next to the re-created processed image presented to the network, generating video captures such as shown in Figure 4.8. Examples of running the script to generate videos can be seen in a number of documented "runs", for instance E.5.36 and E.5.36. The third method was also developed for this project (RecordVideo.py). It creates a recording at inference time with up to 3 (in case of added rain) side-by-side images in a single video. An example is shown in Figure 4.17, where most parameters (network name, rain type, predicted steering angle) are generated dynamically, and the "Intensity Multiplie" (sic) value is set in SDSandbox, hardcoded in predict_ client.py, this python script then committed to git repository and a record of commit hash, tracking the code state at the moment recording was made, is kept in appendix E.5 to make results repeatable. The concept of altering intensity multipliers to simulate glare on wet roads is in response to a suggestion by this project's supervisor (see correspondence with supervisor B.14.2). Examples of running the prediction engine to record can be seen in various runs, including E.5.51 and E.5.52.

3.16 Steering data plots

Another important analysis tool are steering data plots, be it steering angle labels for an image or predictions returned by a Keras model for an image. In this project plots are created with the python matplotlib.pyplot and seaborn modules, using both jupiter notebook such as GetSteeringAnglesFromtcpflow.ipynb and python scripts such as result_ plots.py and steerlib.py (F).

4 Results

4.1 Outputs

Since this dissertation's topic first took shape (B.14) a number of outputs were generated, not all recorded or saved. With respect to outputs that were recorded and saved, this project generated five classes of outputs: 1. models, 2. videos, 3. tcpflow logs, 4. synthetic datasets and 5. code. A number of "Runs" were recorded in E.5. A run may or may not generate a model, may or may not generate a video and may or may not generate a tcpflow log. A model may be run several times, as were the best nvidia1 (E.5.49) and nvidia2 (E.5.62) models, tested with different levels of rain and light intensity multipliers. A compilation of saved models, videos and tcpflow logs, shown in E.6 lists 73 models, 61 published (YouTube) videos and 30 tcpflow logs at the time of compilation. Public download links are available in E.7.

4.2 Datasets

The following training unity datasets were generated:

```
$ . count.sh
871403 files in directory .
25562 files in directory ./roboRacingLeague
25795 files in directory ./log_sample
82245 files in directory ./warehouse
44496 files in directory ./quarantine
90842 files in directory ./smallLoop
516997 files in directory ./genRoad
16570 files in directory ./genTrack
68886 files in directory ./smallLoopingCourse
```

where approximately half are the labels (.json files) so the total number of images generated is 435701 while the most used for training and testing were genTrack and genRoad for an approximate total of 266783 images used to train the best performing models. In addition to Generated Track and Generated Road, the RoboRacingLeague and Warehouse tracks were also attempted and could have potentially worked. In the end to keep the problem constrained, two tracks were used. From the real life datasets considered, Audi, FordAV, Kitti and Udacity. All were inspected and notes made in B.8 and the dissertation latex directory work-diary.txt.

Correspondences were held with authors (B.15) with respect to extracting steering angles from datasets, and ROS (Robot Operating System) was used to extract data from public dataset files provided in the rosbag format (B.8.1).

4.3 Locating rainy sections with Mechanical Turk

To investigate the effect of rain in self-driving CNNs in a real world dataset, an attempt was made to locate such images in available autonomous vehicle public datasets. The data that looked most promising was Ford V2 Log 1 and V3 Log 1 from the FordAV dataset (Agarwal et al., 2020), both described as "freeway, overpass, bridge, cloudy". The goal was the set to label a sample of the cloudy images with Amazon Mechanical Turk and 50 images were uniformly drawn from each log, plus 5 images with rain added with the Automold library using the RainyImagesDissertationPlot.ipynb script. The expectation being the 5 extra images be labelled as "Rain". The 105 images were then added to a SageMaker Ground Truth job. The platform works as a wrapper around Mechanical Turk as it facilitates the creation of user interfaces (Figure 4.1) Labelling tasks are classified as low, medium and high complexity. The prices

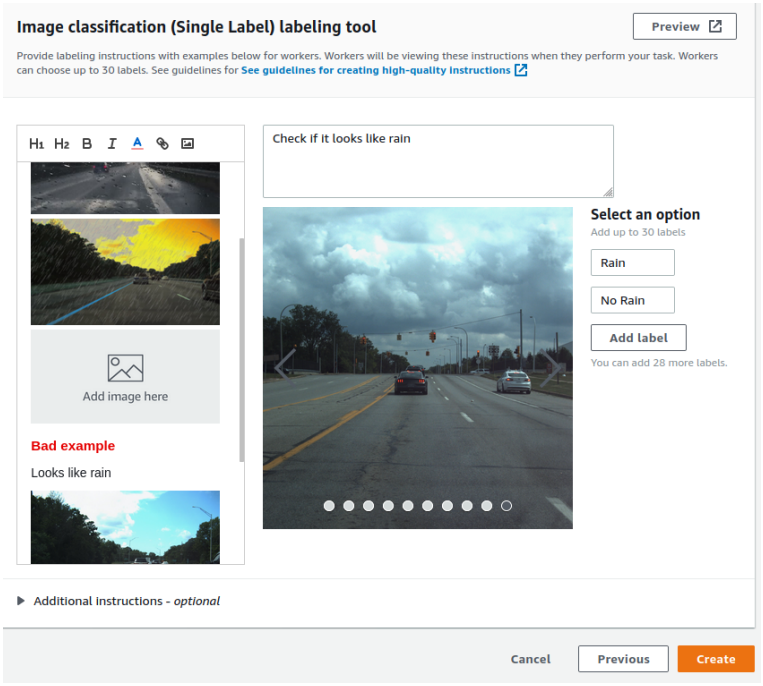

FIGURE 4.1: The SageMaker Ground Truth user interface template with two labels ("Rain", "No Rain"), the image to be labeled and examples of correct and incorrect classifications. On the top left are images supplied as "Good example" of rain. The lower of the two being an image modified with the Automold library to add rain. After the job was submitter, the image was labelled by a Mechanical Turk worker as "No Rain".

range from $0.012 for a low complexity task, with a 5 second estimate to complete, to $1.20 for a high complexity task with a 3.5 minute estimate to complete. The task can be configured with an output for time taken by a worker on a single task, the lowest time interval being one minute. There is also an option to assign more than one worker per dataset object, on

account that it can help increase the accuracy of the data labels. The task was created with the most basic options of one worker, a $0.012 price per task, a one minute timeout and remained live for 12 hours. The images were labelled by one Mechanical Turk worker and results made available in a json encoded file (dataset/mechanical-turk/2020-11-21_22 45 37.json). In case more than one worker is assigned to a same task, a confidence score is also provided in the results. Results showed no images had been labeled as "Rain". The assumption then being there are no sections containing rain in the dataset. The Automold library added rain images were also labeled as "No Rain" by the worker. Figure 4.2 shows one page of the SageMaker Ground Truth completed job, detailing 4 labelled images on the Labeling Job Summary page. A side-effect of the labelling job being Automold rain where not seen by a human as such. The implication is further discussed in 5.

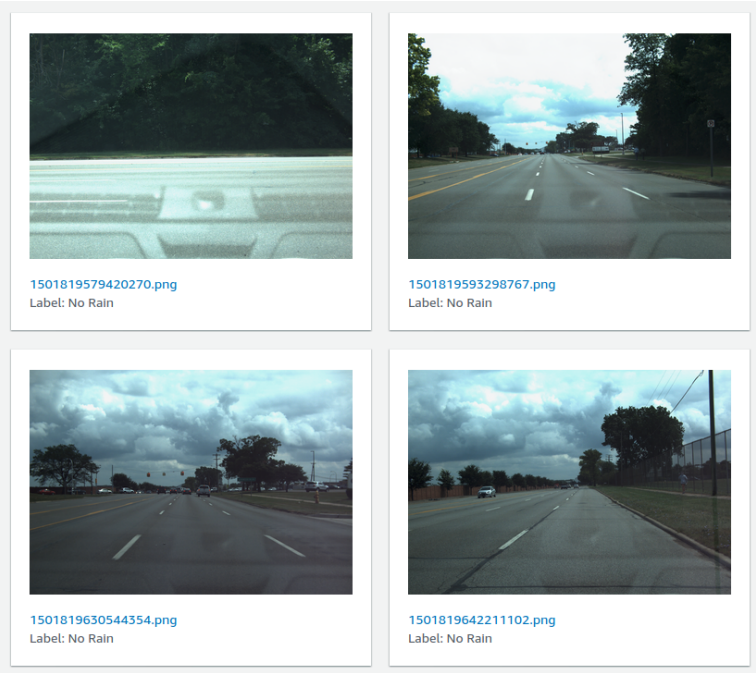

FIGURE 4.2: The SageMaker Ground Truth labeled images detail on Labeling Job Summary page

4.4 Generating Synthetic Datasets

4.4.1 Generated Track

Figure 4.3 shows a normalized histogram of 45410 steering angles obtained in 10 recorded sessions (data directory dataset/unity/smallLoop) for the small_ looping_ circuit track seen on the right. The circuit is the same as the Generated Track circuit, while the landscape in the latter excludes trees, bushes and tall grass. The simulator records at a, computational resources allowing, maximum rate of 60 fps (frames per second). The observed average on the track was approximately 24 fps, representing in this case 31m32s of recorded data in approximately 19 laps. The 24 fps average was obtained by recording ("Auto Drive w Rec" option) for one lap,

taking approximately 1m41s seconds and generating 2455 frames. The simulator registered fps rates higher than average on straight sections and lower than average on curved sections. This is assumed to be due to computational overheads imposed on the physics engine in the curved sections, resulting in less frames being processed and recorded. The maximum steering angle is set in the simulator with respect to a car. In this case it is plus or minus 25 degrees. The normalized value, in the range of -1 to 1 is recorded. Values displayed in the histogram are multiplied by the maximum steering angle, referred to in this study as the *normalization constant*. The data shows a high bias towards positive steering angles because the simulated vehicle drives clockwise around the track. From the starting line on the top left, there are two right turns, followed by a left turn, followed by two right turns.

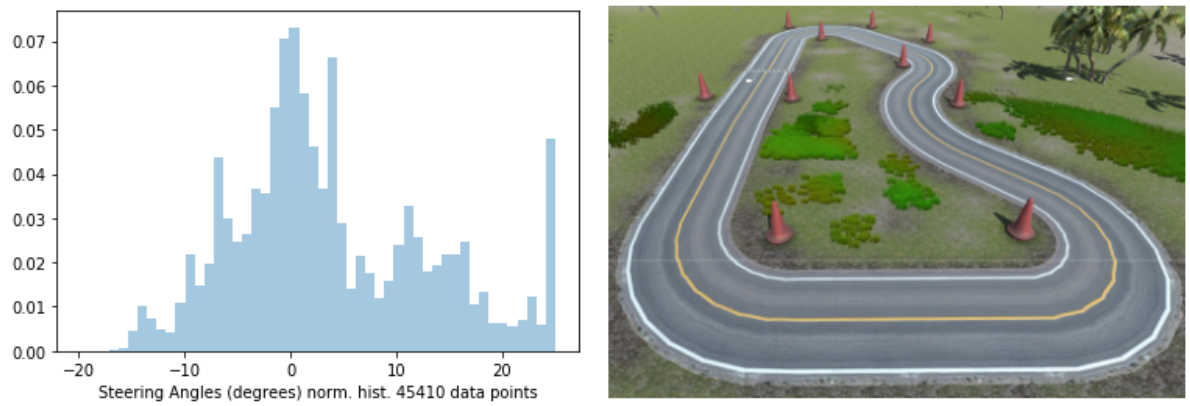

FIGURE 4.3: Normalized histogram of Unity 3D SDSandbox steering angles for 45410 image frames. The corresponding track (small_looping_couse) is shown on the right

Figure 4.4 shows the *ground truth* (not predicted) steering angles obtained in "Auto Drive w Rec" mode, for one lap driven around the Generated Track, saved to log directory dataset / Unity / genTrack / logs_ Wed_ Nov_ 25_ 23_ 39_ 22_ 2020 made available for download in E.7. The y axis is inverted such that plot follows steering adjustments, if observer tilts head to the right. The four troughs represent the four right turns. The hump between frames 500 and 700 represents the only (gentle) left turn in the circuit, where the steering angle is keep negative while the vehicle negotiates the bend.

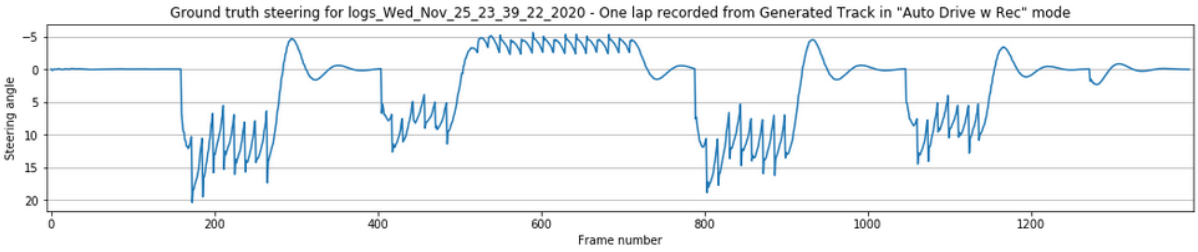

FIGURE 4.4: Ground truth steering values for logs_ Wed_ Nov_ 25_ 23_ 39_ 22_ 2020 recorded log.

4.4.2 Generated Road

Another simulated circuit used was the *Generated Road*, which creates a random path on every run. This can be seen in figure 4.5. The total number of frames collected for 15 *Auto Drive w Rec* sessions (saved in dataset/unity/genRoad) was 280727, approximately 3h14m37s. The outliers, mainly to left of histogram, are due to the simulated vehicle being left unattended, reaching the end of the road and continuing into a section with no road markings, becoming stuck in a hard-left turn, until the simulation was switched off and recording halted. The mean and standard deviation, for angles in the range -20 to + 20 (excludes outliers), are -0.18 and 5.37 degrees respectively. The number of outliers omitted from the plot (not excluded from data at this stage) was 2204 (0.79%). It can be seen that when more data points are collected, the histogram becomes "smoother", if compared to equivalent plot in Figure 4.3. Both boths are clearly centered around zero degrees, meaning the vehicle is driving straight more often than not. The Generated Road has a normally distributed steering angle distribution because of the same number of left and right turns in the circuit, which infers the data distribution is track-dependant.

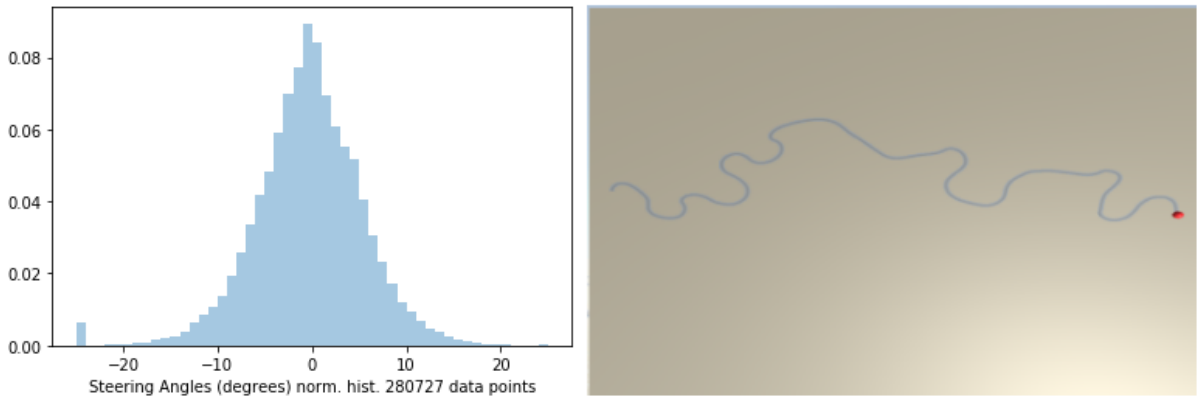

FIGURE 4.5: Normalized histogram of Unity 3D SDSandbox generated road, steering angles for 280727 image frames. A sample randomly generated road is shown on the right. Outliers in negative range are due to oversteering when vehicle reached the end of the road and simulator was left recording.

4.5 Training self-driving models

Model training was performed using a modified version of the SDSandbox/src/train.py script. A successful training run produces four files: a plain text .log file containing the script running time duration and last recorded values of accuracy and loss for training and validation (the training "history"), a .history file in the pickle format containing all history values, a model .h5 file in the pickle format containing the model structure and weights, and a .png image file containing a history plot for the training run. Figure 4.6 shows three training history plots: 20201124032017_ nvidia2.h5 trained for 23 epochs on the Intel DevCloud, 20201121090912_ nvidia_ baseline.h5 trained for 2 epochs on Camber and 20201120184912_ sanity.h5 trained for

82 epochs on the local workstation. The left y axis represents loss and the right y axis represents accuracy values. The x axis represents training epochs (one run through the entire dataset). The title for each plot contains the last recorded training history values and the model name.

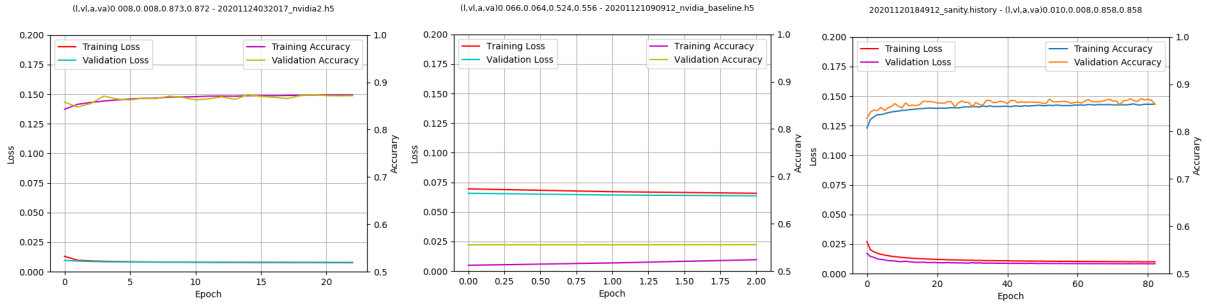

FIGURE 4.6: Training and validation accuracy and loss value plots for, left to right, models 20201124032017_ nvidia2.h5, 20201121090912_ nvidia_ baseline.h5 and 20201120184912_ sanity.h5

Listing 4.5 shows the corresponding log files for each run, training run documented in E.5.62 (devcloud), E.5.98 (camber) and E.5.37 (local) respectively.

```
Model name: ../trained_models/nvidia2/20201124032017_nvidia2.h5
Total training time: 4:06:55
Training loss: 0.008
Validation loss: 0.008
Training accuracy: 0.873
Validation accuracy: 0.872

Model name: ../trained_models//nvidia_baseline/20201121090912_nvidia_baseline.h5
Total training time: 0:05:25
Training loss: 0.066
Validation loss: 0.064
Training accuracy: 0.524
Validation accuracy: 0.556

Model name: ../trained_models/sanity/20201120184912_sanity.h5
Total training time: 16:08:01
Training loss: 0.010
Validation loss: 0.008
Training accuracy: 0.858
Validation accuracy: 0.858
```

Training and validation data was split at 80%/20% ratio. Early stopping was used with patience typically set to 6 epochs for validation loss (stop if no decrease over 5 epochs). Images were presented to network in batches of 64. The Adam optimizer was used with a learning rate

of 0.0001 and mean squared error loss function. Trainnig and testing runs are documented in appendix E.

4.6 Testing self-driving models without rain

To create a proof-of-concept, the initial model architecture used was TawnNet. In the documentation, the author states that with the SDSandbox framework, self-driving was achieved with a few hours of recorded laps and up to one day (24 hours) training time on GPU, with the code supplied "as is". This experiment was replicated (20.07.2020 on CPU) while the outcome was not as the car, after initially staying on the Generated Road, drove off the track. The experiment recorded in video `https://youtu.be/453mTlL2gvs`.

The first working model (able to successfully self-drive around the Generated Track) was the 20201107210627_ nvidia1.h5, with the nvidia1 (TawnNet) model on 07.11.2020. A video was generated using the recording screen utility Kazam (Kazam, 2020), recording at 15fps (frames per second), and published at `https://youtu.be/9z0mMtOnUUc`. Figure 4.7 shows 3 stills from the video containing from left to right, the game engine, the tcpflow TCP debug output and the prediction engine running. tcpflow was added to the process in this project as a debugging tool, and does not exist in the original SDSandbox.

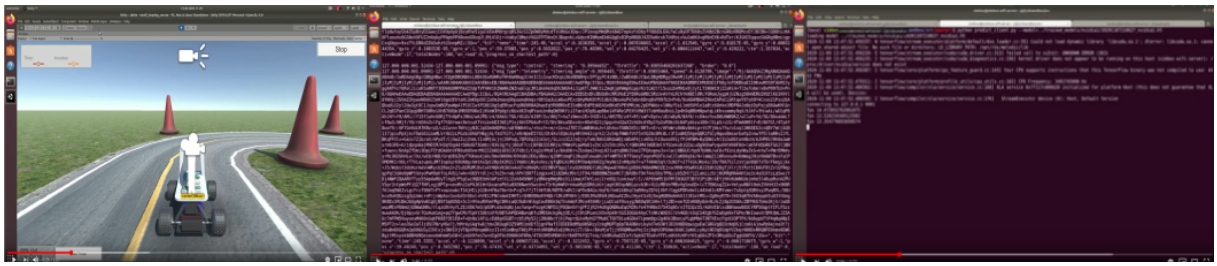

FIGURE 4.7: Stills of video `https://youtu.be/9z0mMtOnUUc` showing left to right: SDSandbox simulated car going around the Generated Track course, TCP Debug (tcpflow) and prediction engine (predict_ client.py) running

A record of what code generated the successfull model was not made at the time. Forensics in E.5.36 using git logs and further tests conducted in E.5.37, E.5.38, E.5.39 ("sanity" models) determined it was achieved by porting the data augmentation and pre-processing described in 3.8 to the SDSandbox training and prediction scripts. The augmentation is applied during training, with a probability of 0.6, pre-processing is applied to all images. Pre-processing is applied to all images received from simulator during inference time.

Once the sanity model could successfully drive around the Generated Track, tcpflow logs were captured and used to generate the video shown in Figure 4.8. A script was written for this end (MakeVideo.py) and the command line call can be seen in several runs e.g. E.5.41. The video in this case is obtained by processing all still images logged by tcpflow, as when they were sent from the simulator to the prediction engine during inference time. In MakeVideo.py, the adapted Shibuya, 2016 augmentation library is used to pre-process the original image such that it will be the same as the image presented to the network. The predicted angle is obtained from

tcpflow. Labels added to the images, then both images (original and processed) are concatenated horizontally and added to a video one frame at a time using the opencv-python module. The ratio for each individual frame in the composition is set to 800x600 pixels.

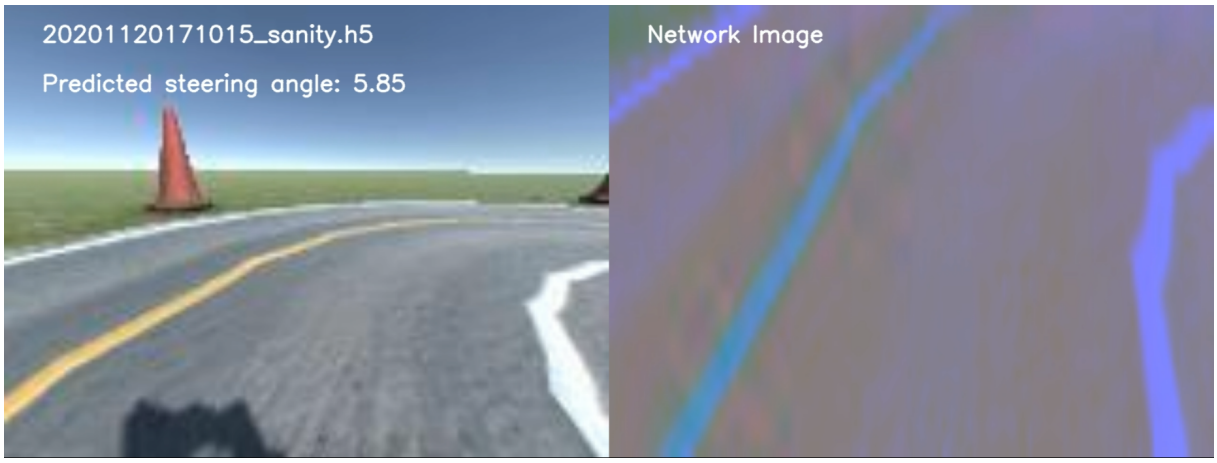

FIGURE 4.8: Video still from run 41 video `https://youtu.be/LEmZJJzJkEE` showing simulator image as sent over TCP network on the left with added CNN (20201123162643_ sanity.h5 model) steering angle prediction and processed image (as presented to CNN) on the right

Figure 4.9 shows the plot for **predicted** steering angles around Generated Track for run 41, whereas in Figure 4.4 shows **ground truth** steering angles are plotted. The predictions start at around 5 degrees steering slowly edging close to zero degrees. This can be observed on the video as the car moves closer to the left edge of the lane (positive steering), then continues straight. The plot for predicted angles seems smoother than for ground truth angles. This may be due to the PID settings where the response to going off the desired path is more abrupt, and "jerkier", in trying to correct itself.

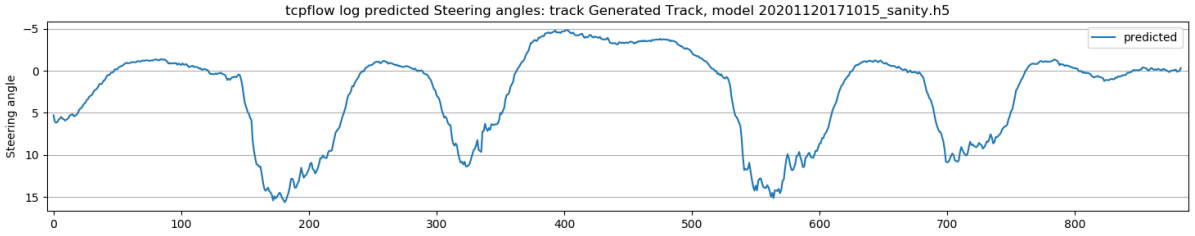

FIGURE 4.9: Steering angle plot generated from tcpflow log obtained in run 41 (E.5.41)

At this stage, the nvidia1 model was proven to self-drive. The nvidia2 and nvidia_ baseline models required more work. The first step was to "clean" the genRoad dataset. Figure 4.10 shows from left to right, an overlay of all plots of normalized histogram plots for steering values contained in unity/genRoad directory, the folder containing most outliers (logs_Thu_Jul_ _9_16_00_15_2020) and the resulting plot once the outliers' folder was removed from unity/genRoad. The presence of outliers is believed to have caused issues with the nvidia_ baseline model from runs E.5.14 to E.5.31. The next issue was found to be the ratio used to crop images presented to both "problematic" networks. Run 55 (E.5.55) provides analisys. Figure 4.11 shows on the left,

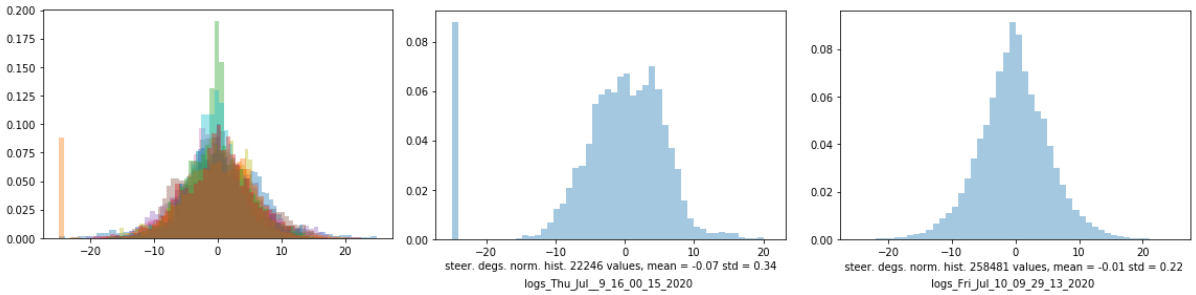

FIGURE 4.10: Left to right, normalized histograms of all genRoad folders, outlier's folder and all data with outliers removed.

the crop (pre-processed image) presented to nvidia1 and sanity models. The actual crop values were determined by the imported augmentation library, which had crop values set to "frame" the "part of interest" in images (320x160) generated by the Udacity sim. The crop turned out to work for the nvidia1 and sanity models with images (160x120) generated by the SDSandbox sim, which is a "fluke": by removing 60 pixels from the top and 25 pixels from the bottom of SDSandbox images, the result was an image that seems to best frame the road. The issue was corrected by using the third crop, from left to right and a working nvidia2 model was generated in run 62 (E.5.62). Crop values are set in script conf.py, adapted for this project.

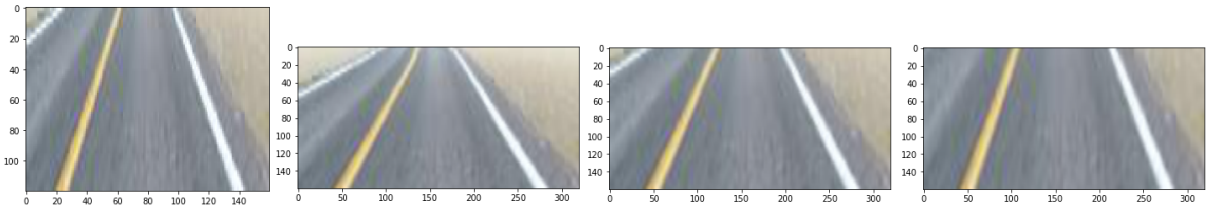

FIGURE 4.11: Left to right, nvidia1 network crop, nvidia2 network crops at top crop set to 70, 81 and 91 pixels left to right. The 81 pixel top crop was used for the nvidia2 network

Figure 4.12 shows a randomly Generated Road used for runs 93, 94 and 95, where the first few hundred frames go to near the top left of the circuit (long right 180 degree turn) shown in detail on the right hand image. The predicted steering angles shown in Figure E.31 for the well performing nvidia2 20201207192948_ nvidia2.h5 model, show the majority of values in the positive range, as the simulated vehicle follows the road and takes a right turn to do so.

4.7 Adding rain

Using the Automould library, scrip predict_ client.py was modified for rain to be added in real time for predictions. Two schemes were considered. The first where rain is added directly to the network image, as show in Figure 4.13. Although the procedure introduces noise to images, it is somewhat unrealistic, as rain is expected to be present on the acquired image. The second scheme, adopted for this project, was adding rain to the acquired image. This is shown in Figures 4.17, 4.18, and 4.19. Two additional command line parameters, *rain* and *slant* were

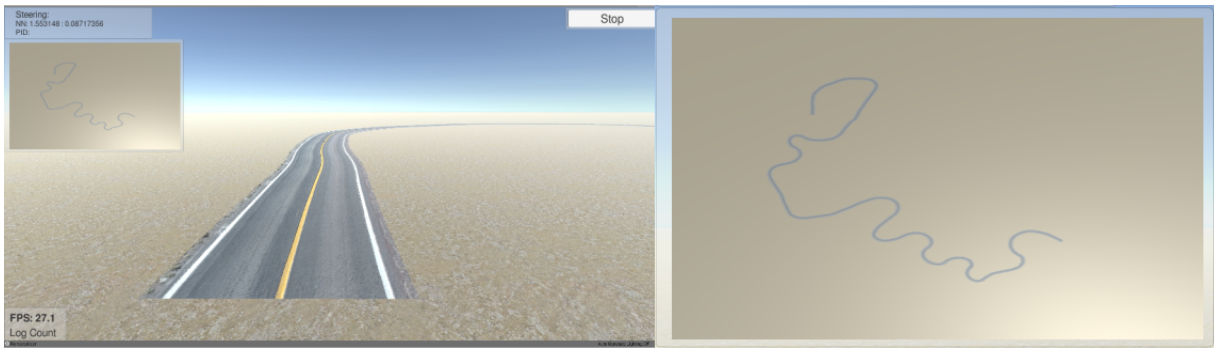

FIGURE 4.12: The randomly generated Generated Road circuit used in runs 93 and 94. Left image is a view of the simulator as presented on computer desktop, the right image is the augmented detail showing the Generated Road circuit, the same inset on left image.

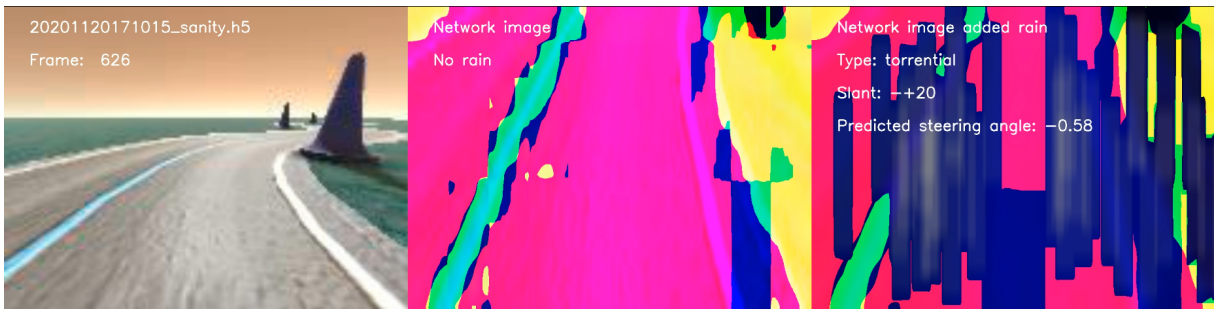

FIGURE 4.13: Video still showing left to right, the acquired image supplied by the simulator, a processed network image with no rain and the same image with torrential rain added. This last being the image presented to the network. A video was recorded in run E.5.43, `https://youtu.be/57jwwcjbfdE`, showing the model driving off the track.

added to the prediction script. This can be seen in a number of documented runs e.g. E.5.72 (light), E.5.73 (heavy) and E.5.74 (torrential).

4.8 Evaluation of self-driving cars using CNNs in the rain

Two types of predictions were run, realtime and simulator log synthetic dataset predictions. Realtime is done with the SDSandbox framework, that is, the Unity simulator is started, tcpflow is set to log, and the prediction script run, i.e. the same sequence shown in Figure 4.7. Synthetic dataset predictions were run using the steerlib.py script, written for this project. In the first type (realtime) there is no "ground truth", in the second type (dataset predictions) there is, as the dataset consists of labelled (steering angle values) images, that is, each image has an assigned steering angle set when the data was logged, which can be compared *a posteriori* to a predicted value. The outputs that generated g_s scores, plots and Table 4.1 rows are left as comments in the steerlib.py script, in the final project commit 86937130. To simulate the effect of added reflections, the sky in the Unity simulator was set to black and different Unity intensity multipliers (1, 4 and 8) were used. Using the procedure described in 4.7, 32 realtime predictions were run for the best performing nvidia2 and nvidia1 models.

4.8.1 Realtime predictions for best performing nvidia1 model

Figures 4.14, 4.15 and 4.16 show plotted steering angles for the best nvidia1 model, on the Generated Track, using different intensity multiplier values (1, 4 and 8) and rain levels (light, heavy, torrential). Blue lines are for "no rain" data generated in run 81 (E.5.81) with intensity multiplier 1. The difference in vertical elongation is accounted for by different y axis ranges. All plots were generated with result_ plots.py script written for this project. Note, for plots generated in Figure 4.14 Default Unity Skybox Material setting Default-Skybox (blue sky) was used. This is believed to not have affected the outcome, as the network image when compared to simulations using Skybox Material to SkyCarLightCoversGrey. Figure E.26, first row, shows the figure, with intensity multiplier 1 and light rain, presented to network (last on the right) being similar.

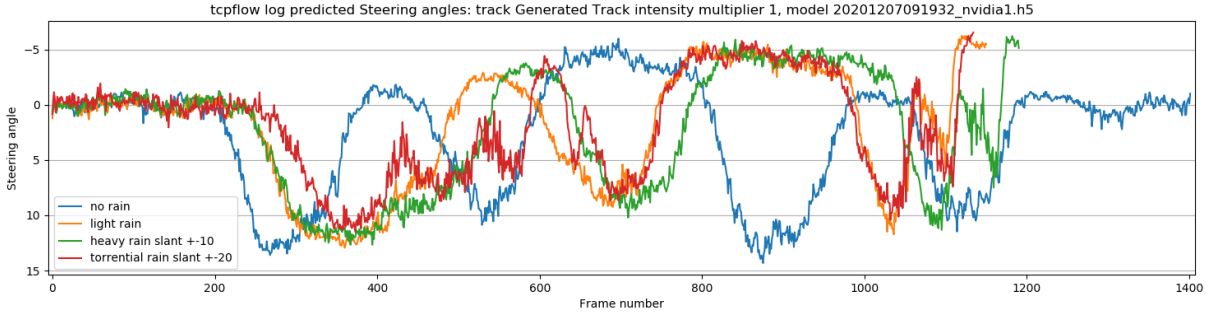

FIGURE 4.14: 201207091932_ nvidia1.h5 steering angle prediction plots for Generated Track, intensity multiplier 1, runs E.5.81 (no rain), E.5.72 (light) , E.5.73 (heavy) and E.5.74 (torrential).

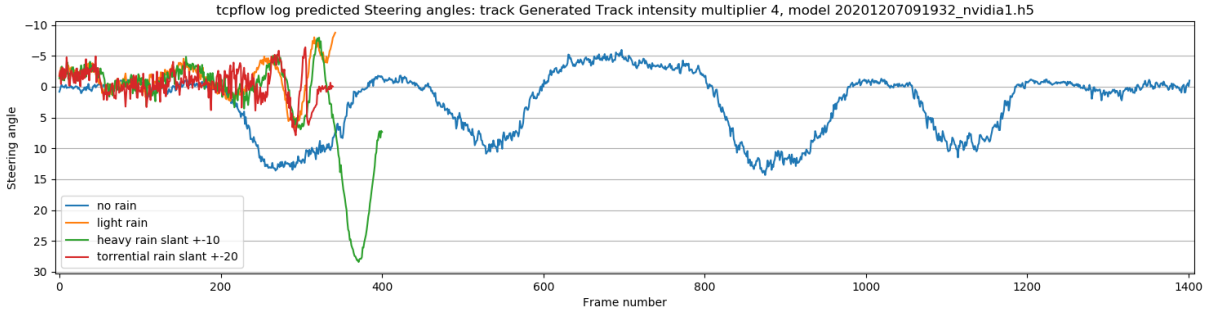

FIGURE 4.15: 201207091932_ nvidia1.h5 steering angle prediction plots for Generated Track, intensity multiplier 4, runs E.5.81 (no rain), E.5.75 (light), E.5.76 (heavy) and E.5.77 (torrential). The initial negative (left) steering values predicted by all rain models can be verified in corresponding videos.

In figure 4.14, the blue line "no rain" plot shows the circuit curve sequence, with the two right turns (rounded inverted peaks in the positive steering angle range) followed by a left turn (plateau in the negative steering angle range) followed by to right turns. With rain the nvidia1 models understeer (shown by the gentler dip if compared to blue "no rain" light in plot) when taking the first right turn, the "torrential rain" model steering erratically when adjusting the steering before taking the second right turn, while the "light" and "heavy" rain models are smoother. The torrential rain model also takes the second right turn erratically while the other

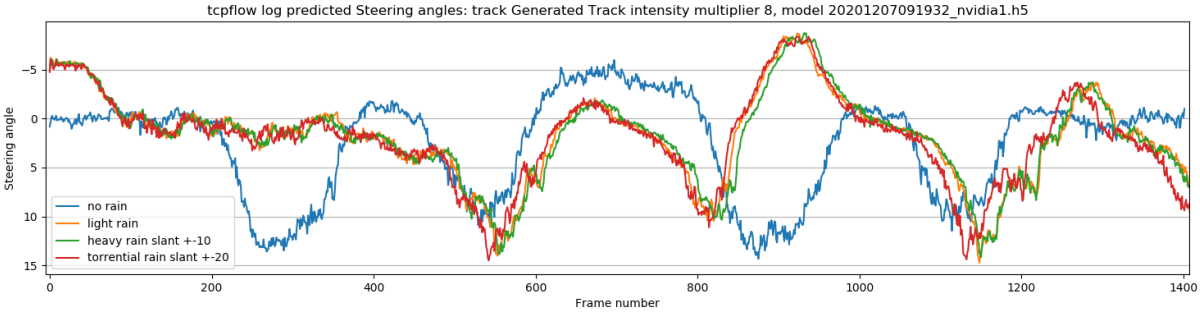

FIGURE 4.16: 201207091932_ nvidia1.h5 steering angle prediction plots for Generated Track, intensity multiplier 8, runs E.5.81 (no rain), E.5.78 (light), E.5.79 (heavy) and E.5.80 (torrential).

two are smoother. All three models take the long left turn quite well, becoming erratic on the third right turn where all drive off the track. The four plots are overlayed and the circuit geometry and consequent position of model on track can be inferred by the plot shape. The prediction latency (time taken to predict a steering angle given an image) for every weather condition is assumed to be very close in all four runs. This can be verified in data related to Figure 4.16 where all models completed the lap and all logs have approximately the same number of frames (x axis maximum value). The horizontal displacement between model prediction plots is due to models going around the track at different speeds. This can be verified in the corresponding videos by pausing all three near frame 1000 (labelled on left image), which shows "light" (E.5.72 - corresponding to orange line in plot) and "torrential" (E.5.74 - corresponding to red line in plot) ahead of "heavy" (E.5.72 - corresponding to green line in plot).

Figures 4.17, 4.18 and 4.19 shows stills from videos generated in runs E.5.75 (light), E.5.76 (heavy) and E.5.77 (torrential) respectively. The runs generated tcpflow (intensity multiplier 4) data used in Figure 4.15 orange, green and red lines respectively, all three driving straight off the first right turn, E.5.77 crashing into a bollard.

FIGURE 4.17: Still from video `https://youtu.be/qdTA5ho5VOE` showing a self-driving simulated vehicle driving off the Generated Track, where the Unity scene is set to a dark horizon (Setting Skybox Material to SkyCarLightCoversGrey) and the intensity multiplier is set to 4. Rain is set to light. The image sent from Unity simulator over the TCP network is shown on the left, the middle image has added rain and the image on the right is the image presented to the network. The tcpflow log and video were generated in run E.5.75 and represent the orange line in Figure 4.15.

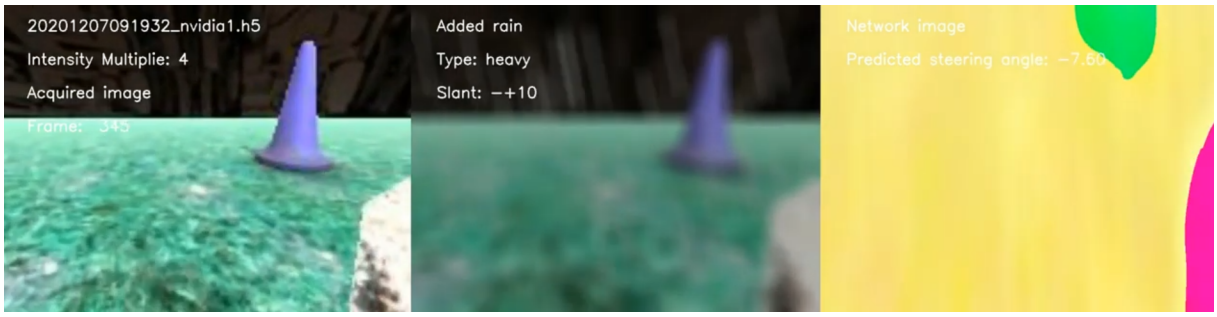

FIGURE 4.18: Still from `https://youtu.be/sKyoke3I084`. The tcpflow log and video were generated in run E.5.76 and represent the green line in Figure 4.15

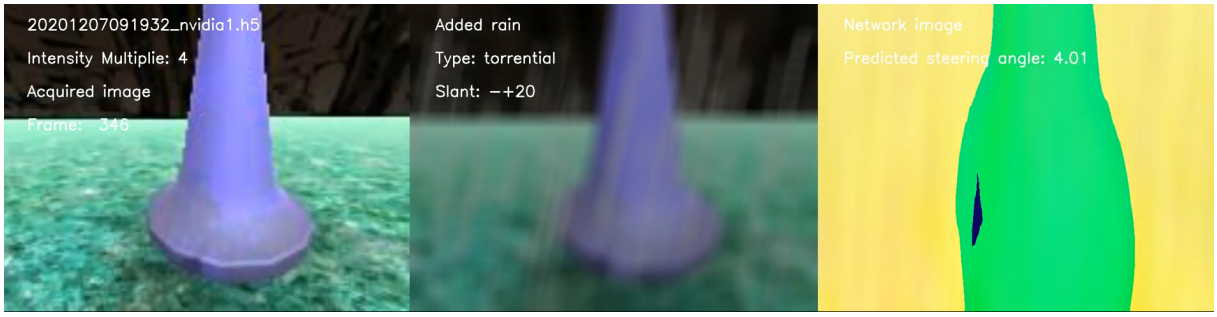

FIGURE 4.19: Still from `https://youtu.be/mDjtnnVZdic`. The tcpflow log and video were generated in run E.5.77 and represent the red line in Figure 4.15

4.8.2 Realtime predictions for best performing nvidia2 model

Figures 4.20, 4.21 and 4.22 show plotted steering angles for the best nvidia2 model, on the Generated Track, using different intensity multiplier values (1, 4 and 8) and rain levels (light, heavy, torrential). Blue lines are for "no rain" data generated in run 82 (E.5.81) with intensity multiplier 1. All plots were generated with result_ plots.py script. The compressed shape of the "no rain" plot in 4.22 compared to 4.20 and 4.21 is due to the predicted steering angle range for rain models in Figure 4.22 being wider, not that predicted angles generated by the models exceed in some cases the maximum Unity simulator steering angle.

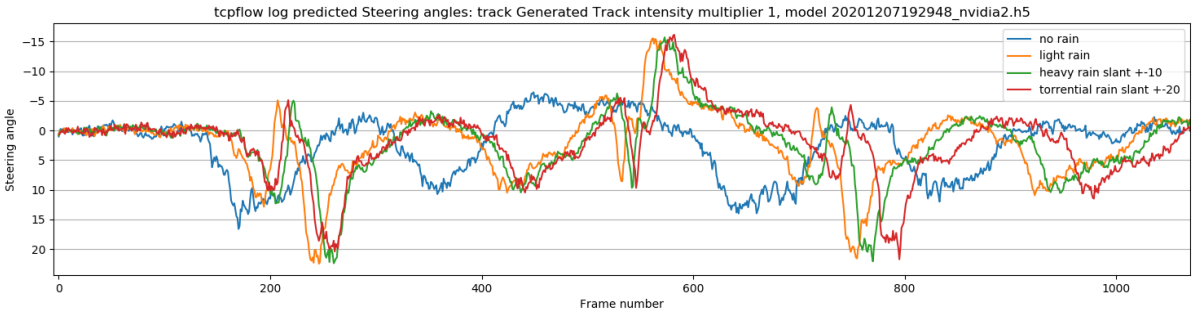

FIGURE 4.20: nvidia2 20201207192948_ nvidia2.h5 steering angle prediction plots for Generated Track, intensity multiplier 1, runs E.5.82 (no rain), E.5.83 (light) , E.5.84 (heavy) and E.5.85 (torrential). The rain models almost drive off the track near frame 200, then correct themselves. Another near drive-off-the-road incident is recorded near frame 500 followed by another one near frame 700. All rain models recovering and managing to complete the lap.

The horizontal shift between light (orange), heavy (green) and torrential rain (red) plots in Figure 4.20 is due to the models going around the track at different speeds. In this case it can be verified by pausing videos generated in the corresponding runs E.5.83, E.5.84 and E.5.85 near frame number 800, labelled on the first image from left to right. At this point, the model subject to light rain is ahead of the model subject to heavy rain, in turn ahead of the model subject to torrential rain, which reflects the displacement of the three plots with orange negative peak to the left ("ahead") of the 800 mark frame, the red negative peak near the 800 axis label and the green negative peak in the middle.

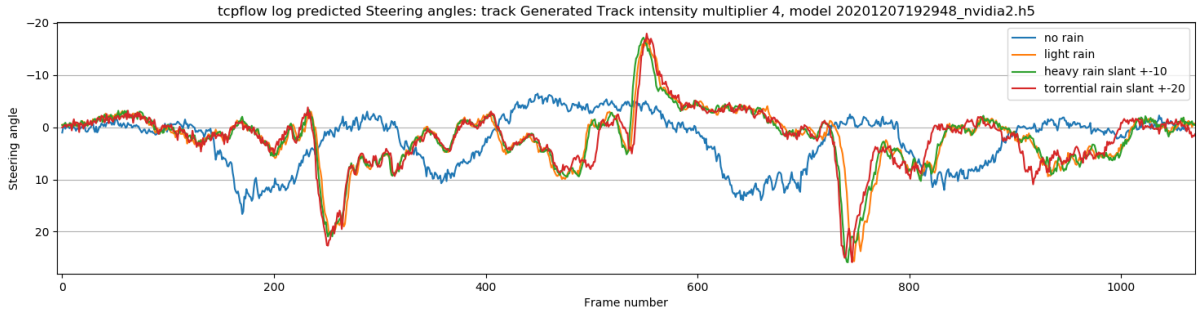

FIGURE 4.21: nvidia2 20201207192948_ nvidia2.h5 steering angle prediction plots for Generated Track, intensity multiplier 4, runs E.5.82 (no rain), E.5.86 (light) , E.5.87 (heavy) and E.5.88 (torrential).

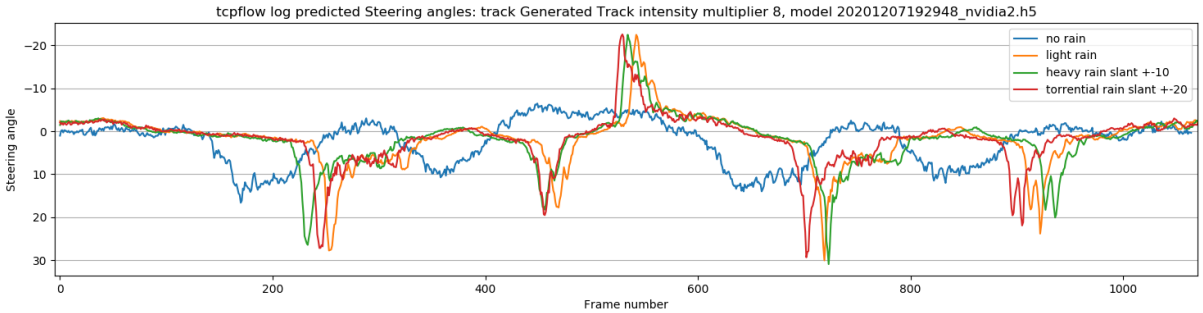

FIGURE 4.22: nvidia2 20201207192948_ nvidia2.h5 steering angle prediction plots for Generated Track, intensity multiplier 8, runs E.5.82 (no rain), E.5.89 (light) , E.5.90 (heavy) and E.5.91 (torrential).

The general shape of the negative and positive peaks for the best nvidia2 models driving in the rain reflect the more abrupt steering angles predicted by the model in situations when they were about to drive off the road. The nvidia1 model, when it did manage to complete a lap in the rain (Figure 4.16) presented comparatively smoother steering.

Figure E.26 in appendix E shows three rows of stills, generated in runs 83 (E.5.83), 86 (E.5.86) and 89 (E.5.89 for light rain and intensity multipliers 1, 4 and 8 respectively, The data generated was used to plot the orange lines in figures 4.20, 4.21 and 4.22 for the best performing nvidia2 model. The sequence highlights the effect of preprocessing, when noise has been applied in equal measure to images generated with different intensity multipliers, where in the first row (intensity multiplier 1), the road markings are still visible, in the second row (intensity multiplier 4) the road markings are hardly visible and the reflection in the middle of the lane is

generating a continuous vertical stripe on the network image, and in the third row, where the reflection on the road is such, road markings are no longer visible.

This concluded the 32 recorded realtime runs in the rain for the best performing models. One more video was generated with Kazam, showing 4 concurrent predictions on the same Generated Track using built in SDSandbox functionality. Figure E.34 shows a still from video `https://youtu.be/ayESXH9zZdM`. Bottom left is best nvidia2 20201207192948_ nvidia2.h5 model running predictions for no rain, bottom right is 20201207192948_ nvidia2.h5 light rain zero slant. Top left is best nvidia1 20201207091932_ nvidia1.h5 model, light rain zero slant (crashes on third right turn), top right is 20201207192948_ nvidia2.h5 torrential rain -+20 degree slant. This provides a qualitative analysis of how well the models are steering, where the bottom left nvidia1 in no rain is noticeably the smoothest steering model.

4.8.3 Synthetic dataset predictions

Table 4.1 shows Goodness-of-Steer results obtained for 32 sets of predictions, for four networks and two SDSandbox track logs, that is, the results were generated by running predictions on synthetic datasets, which contain ground truth steering values, like plotted in Figure 4.4. It is important to stress these are not realtime predictions, and the Unity simulator was not running in this case.

The networks are the best performing nvidia2 (obtained in run 62 - E.5.62), the second best performing nvidia1 (obtained in run 49 - E.5.49), "sanity check" (obtained in run 36 - E.5.36) and nvidia_ baseline (obtained in run 68 - E.5.68). Table data and steering angle plots was generated with steerlib.py script written for this project. The track logs are for Generated Track (one lap) and Generated Road (one stretch). There are 32 rows, each sequence of four (1-4, 5-8 and so on) representing one model subject to sunny weather followed by 3 types of rain predicting steering angles for one log.

nvidia2 predicted steering angle values used to generate g_s scores in rows 1, 2, 3 and 4, for the Generated Track log are plotted in Figure 4.23. The "no rain" predictions in row 1 (orange line on the plot), which has the lowest overall (1.68) g_s score, is clearly the closest to the ground truth blue line, while the predictions for images containing light, heavy and torrential rain are tightly overlapped (green, red an violet lines, g_s scores 2.12, 2.17, 2.30 respectively) are understeering on all turns (slightly above right turn ground truth plot in right turn sections and below ground truth plot in the left turn section. The understeering aspect was also noted in realtime predictions with respect to rainy predictions compared to best-run dry weather predictions as shown in Figures 4.20, 4.21 and 4.22.

For the best nvidia1 model, predicted steering angle values used to generated g_s values in rows 5, 6 and 7 and 8 are plotted in Figure 4.24. The "no rain" orange plot corresponding to row 5 shows some understeering, with the orange plot not following the ground truth blue plot as closely as the corresponding nvidia2 plot in Figure 4.23 . This is reflected in the slighty higher 1.82 g_s score in comparison to corresponding nvidia2 result in row 1. Predictions for rain follow the close overlap observed for the nvidia2 models, the g_s scores for nvidia1 rain

Goodness-of-steer results - Generated Track and Generated Road SDSandbox logs					
Generated Track log: logs_ Wed_ Nov_ 25_ 23_ 39_ 22_ 2020 (1394 images)					
ID	Keras model file name	Model	Rain Type	Slant	g_s
1	20201207192948_ nvidia2.h5	nvidia2 :)		0	1.68 *
2	20201207192948_ nvidia2.h5	nvidia2	light	0	2.12
3	20201207192948_ nvidia2.h5	nvidia2	heavy	10	2.17
4	20201207192948_ nvidia2.h5	nvidia2	torrential	20	2.30
5	20201207091932_ nvidia1.h5	nvidia1		0	1.82 *
6	20201207091932_ nvidia1.h5	nvidia1	light	0	2.11
7	20201207091932_ nvidia1.h5	nvidia1	heavy	10	2.13
8	20201207091932_ nvidia1.h5	nvidia1	torrential	20	2.28
9	20201207201157_ nvidia_ baseline.h5	nvidia2_ baseline		0	2.32 *
10	20201207201157_ nvidia_ baseline.h5	nvidia2_ baseline	light	0	3.12
11	20201207201157_ nvidia_ baseline.h5	nvidia2_ baseline	heavy	10	3.17
12	20201207201157_ nvidia_ baseline.h5	nvidia2_ baseline	torrential	20	3.39
13	20201120171015_ sanity.h5	nvidia1 :(		0	5.03 *
14	20201120171015_ sanity.h5	nvidia1	light	0	3.11
15	20201120171015_ sanity.h5	nvidia1	heavy	10	3.07
16	20201120171015_ sanity.h5	nvidia1	torrential	20	3.00
Generated Road log: logs_ Fri_ Jul_ 10_ 09_ 16_ 18_ 2020 (19679 images)					
ID	Keras model file name	Model	Rain Type	Slant	g_s
17	20201207192948_ nvidia2.h5	nvidia2 :)		0	2.99 *
18	20201207192948_ nvidia2.h5	nvidia2	light	0	3.20
19	20201207192948_ nvidia2.h5	nvidia2	heavy	10	3.22
20	20201207192948_ nvidia2.h5	nvidia2	torrential	20	3.27
21	20201207091932_ nvidia1.h5	nvidia1		0	3.87 *
22	20201207091932_ nvidia1.h5	nvidia1	light	0	3.75
23	20201207091932_ nvidia1.h5	nvidia1	heavy	10	3.70
24	20201207091932_ nvidia1.h5	nvidia1	torrential	20	3.57
25	20201207201157_ nvidia_ baseline.h5	nvidia2_ baseline :(		0	5.51 *
26	20201207201157_ nvidia_ baseline.h5	nvidia2_ baseline	light	0	4.97
27	20201207201157_ nvidia_ baseline.h5	nvidia2_ baseline	heavy	10	4.98
28	20201207201157_ nvidia_ baseline.h5	nvidia2_ baseline	torrential	20	5.05
29	20201120171015_ sanity.h5	nvidia1		0	3.85 *
30	20201120171015_ sanity.h5	nvidia1	light	0	3.06
31	20201120171015_ sanity.h5	nvidia1	heavy	10	3.05
32	20201120171015_ sanity.h5	nvidia1	torrential	20	3.02

TABLE 4.1: Goodness-of-Steer g_s results for best performing nvidia2 and nvidia1, plus "sanity" and nvidia_ baseline for comparison, obtained from one lap of Generated Track (logs_ Wed_ Nov_ 25_ 23_ 39_ 22_ 2020, rows 1 to 16) and one stretch of Generated Road (logs_ Fri_ Jul_ 10_ 09_ 16_ 18_ 2020, rows 17 to 32). Asterisks in g_s column indicate dry weather. Smiley face :) in Model column indicate best (lowest) g_s score. Sad face :(in Model column indicates worst (highest) g_s scores. The logs containing labelled image data used to compute the g_s scores were generated with Unity intensity multiplier 1 and Skybox-Material set to Default-Skybox, the default sunny dry weather.

FIGURE 4.23: Plots of SDSandbox log logs_ Wed_ Nov_ 25_ 23_ 39_ 22_ 2020 ground truth plus steering angle predictions used to generate g_s values for model nvidia2 in rows 1, 2, 3 and 4 in table 4.1

predictions are better than the corresponding nvidia2 values. The distinguishing feature being the closer predictions to the ground truth for the left turn section. This is consistent with the nvidia1 realtime plots in Figure 4.16, where the nvidia1 model steering angle plot in the closest to the profile of the left turn taken by the model predicing steering in dry weather (blue line), compared to all other equivalent realtime plots.

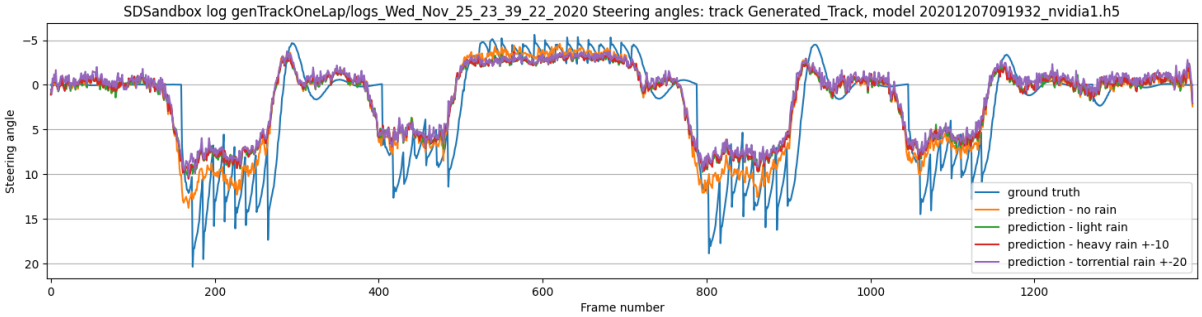

FIGURE 4.24: Plots of SDSandbox log logs_ Wed_ Nov_ 25_ 23_ 39_ 22_ 2020 ground truth plus steering angle predictions used to generate g_s values for model nvidia1 in rows 5, 6, 7 and 8 in table 4.1

Rows 9, 10, 11 and 12 represent data generated from the *nvidia2_ baseline* model, this is the 20201207201157_ nvidia_ baseline.h5 trained on run 68 (E.5.68), commit b1af57c, and is the same architecture as nvidia1, the only difference being it was trained with a 0.00001 learning rate, while the other 3 models tested were trained with 0.0001 learning rate. This nvidia_ baseline model was erroneously trained with nvidia1 geometry. This means that at inference time, the nvidia1 geometry must be used. When the steerlib.py script (commit 8693713) was run to generate the plots, the incorrect parameter *nvidia2_ baseline* typo was passed to the Augmentation library, the model geometry was not found and nvidia1 geometry was used by default. So the nvidia2_ baseline model should be interpreted in this context as nvidia1 trained with 0.00001 learning rate over 5 epochs. The plots show in addition to previous observations on close overlaps of steering angle predictions for rain, and no rain plot following ground truth more closely than rainy predictions, the general trend in this plot is severe understeering on left and right curves, the steering angle predictions rarely being negative values, this is reflected on the worst g_s scores for steering angle predictions in the rain for Generated Track log and could

be attributed to the learning rate used when training the model.

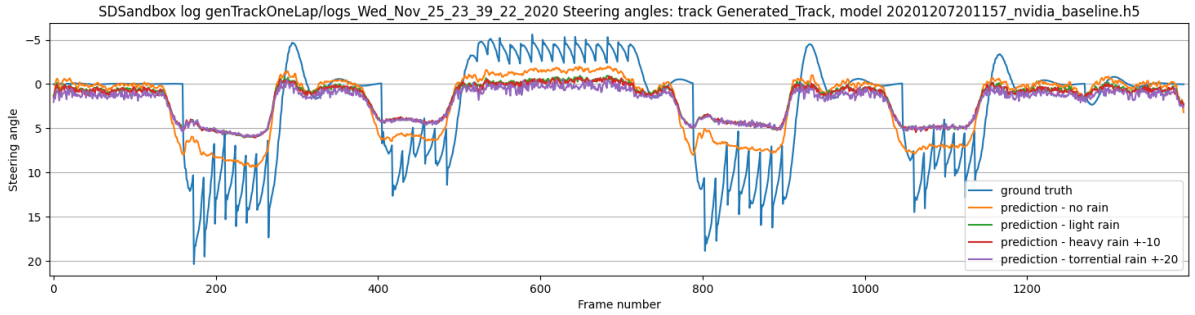

FIGURE 4.25: Plots of SDSandbox log logs_ Wed_ Nov_ 25_ 23_ 39_ 22_ 2020 ground truth plus steering data predictions used to generate gos values for model nvidia_ baseline in rows 9, 10, 11 and 12 in table 4.1

sanity predicted steering angles for Generated Track log used to generate g_s scores in rows 13, 14, 15 and 16 are plotted in figure 4.26 This model, unlike the previous 3 trained on genTrack data, was trained on the log_ sample dataset (small_ looping_ course, run E.5.36), which is the same as the Generated Track course with the addition of trees, bushes, mounts and foliage. The "no rain" plot shows the model oversteering on right turns and understeering on the left turn, reflected in the worst overall g_s score (5.03) for Generated Track log "no rain" predictions, though the closely overlapped rainy predictions achieve better scores (3.11, 3.07 and 3.00) than nvidia2_ baseline (3.12, 3.17 and 3.39) suggesting the lower learning rate value used for the nvidia2_ baseline model may be the cause.

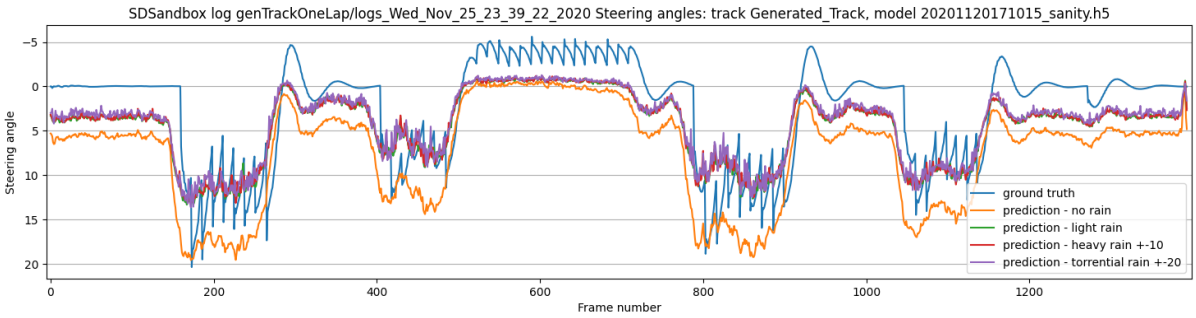

FIGURE 4.26: Plots of SDSandbox log logs_ Wed_ Nov_ 25_ 23_ 39_ 22_ 2020 ground truth plus steering data predictions used to generate gos values for model sanity in rows 13, 14,15 and 16 in table 4.1

No plots were generated for rows 17 to 32 because given the number of images (19679), the plotted lines would have been much thicker and not made good material for qualitative analysis (see Figure E.33). Given there are several right and left turns, in the Generated Road log used for predictions to obtain this set of g_s scores, the higher score (worse) values seem justified as there are more steering adjustments to be made along the path compared to the Generated Track circuit. The overall trend being, nvidia2 had the best overall (2.99) score, nvidia2_ baseline had the worst overall (5.51) score, the sanity model had the best overall rainy prediction scores (3.06, 3.05 and 3.02) and nvidia2_ baseline had the corresponding worst overall (4.97,

4.98 and 5.05), making nvidia2_ baseline the overall worst performing network for the Generated Road log, noting the learning rate used for this model was different.

4.9 Modifications to original source codes

Modifications were required to original source codes that were integrated in this project: Saxena, 2017 (Automold), Shibuya, 2016 (Augmentation), and Kramer, 2020 (SDSandbox). The automold library is designed to process lists of RGB images represented as numpy (Harris et al., 2020) arrays. Function *add_ rain_ single* was added to Automold.py script to deal with image arrays being processed one at a time when running predictions with added rain in real time. The Augmentation library was incorporated into a python class in Augmentation.py. The original code was modified such that before beginning augmentation, the image is resized to the expected acquired size as produced by camera (e.g. 320x160 for nvidia2 and nvidia_ baseline models, 160x120 for nvidia1 and sanity models). This was set as a configuration parameter in the SDSandbox script conf.py. The preprocessing was modified to use top and bottom crop sizes from configurable values set in conf.py.

The modifications to SDSandbox were made to scripts conf.py, predict_ client.py and train.py. conf.py was modified such that parameters could be set then shared across function calls. For instance, when running script predict_ client.py in E.5.83, conf.py is used to store rain, slant and record arguments in configuration settings, further checked throughout the running script to determine actions. Image geometry parameters image width, height, depth, expected network width and height, top and bottom crop values for preprocessing was stored as python lists in conf.py. This ensured that the same image geometry was being used during training and inference time for the specified network. This is set with command line argument –modelname for inference in predict_ client.py (e.g. E.5.83) and –model in train.py (e.g. E.5.92) for training. predict_ client.py was modified to use the Augmentation class, the modified conf.py and the Automold library, to preprocess image with correct geometry, before presenting to network for prediction, and to add rain to images. The script was also modified to generate videos making use of the RecordVideo.py class. train.py was modified to take additional command line arguments –aug and –preproc to determine if augmentation and preprocessing were to be performed for testing. The script was also modified to save: the training history, a history training image and a training log file. Network image geometries settings in conf.py are also used by train.py in the modified version.

5 Discussion

The objectives set out in 1.2 to evaluate self-driving CNNs in the rain were achieved by:

- Determining a game engine able to support the experiments
- Creating a set of synthetic datasets (with functionality provided out-of-the-box by SDSandbox)
- Identifying software able to augment and add rain-like patterns to images, modified and integrated into the original SDSandbox framework
- Training a number of CNN models in dry weather
- Establishing a workflow to modify images to be presented to the network during inference time

and failed in:

- Determining a metric for evaluating how well a self-driving CNN is performing in the rain with respect to steering angle ground truth values

Results presented are valid in the scope of a simulated environment and a CNN predicting steering angles for an image with added rain-like effect. In the experiment conducted using Amazon Mechanical Turk (4.3), with five artificial rainy images added, to a batch of 100 images from the FordAV dataset, no images were classified as "rainy" by a (assumed) real person. Given this outcome, added rain is best interpreted qualitatively as added noise.

This study highlighted the importance of network architecture, image geometry, image preprocessing, cropping the most representative image area and moving the image from RGB to YUV space, making an argument for CNNs being able to generalise. Models trained on Generated Track were able to self-drive on Generated Road (run 95 E.5.95 shows a 16m14s self-drive). Models trained in dry weather were able to drive in simulated wet-like weather. To that effect, particularly significant are the steering plots presented in Figures 4.14, 4.15 and 4.16, for the nvidia1 model driving in all weather with intensity multiplier set to 1. Figures 4.14 and 4.15 show the model crashed in the rain when Unity intensity multipliers are set to 1 and 4. Figure 4.16 shows the nvidia1 model is able to drive around the track when intensity multiplier is set to 8. Figure E.26 shows video stills from acquired to presented to network for a single type of rain (light) and each intensity multiplier. The steering plot in Figure 4.14 (crashed) applying to

the first row, Figure 4.15 (crashed) applying to the second row and 4.16 (did not crash) applying to third row. The nvidia1 model therefore did not crash when there were no lane markings left on the road, and only two distinctive "road" and "non-road" features were present in the network image. The result suggests that rain-like noise did have an impact in the nvidia1 performance, and when the glare was such (intensity multiplier set to 8) that it had the effect of a filter leading to a network image consisting of two features, the nvidia1 model was able to drive around the track. nvidia1 can be seen as the borderline case (the effect of intensity multipliers together with added rain being interpreted as different noise levels), where the model is able to drive around the track depending on network image noise level. The nvidia1 network architecture is therefore on the limit of being able to learn self-driving in the rain.

Since both modesl were trained with the same hyperparameters, the task at hand is then to determine what are the network design differences that could account for performance differences between nvidia2 and nvidia1. The comparison is made between architectures in C.4 and C.3, and the compiled models in commits df953d2 and 55cb00b for nvidia2 and nvidia1 respectively. The architectures show that the input shapes are different (nvidia2 - 66h x 200w x 3, nvidia1 - 160h x 120w x 3). Kernel size, stride and padding are the same for both networks in all convolutional layers, though the kernel size changes from 5x5 to 3x3 and stride from 2 to 1, in the last two convolutional layers.

In the first convolutional layer, both models have the same number of feature maps (24) but a different feature map geometry (nvidia2 - 31h x 98w | nvidia1 - 58h x 78w) as a result of the input geometry being different (see equation 3.2). The number of trainable parameters for both models is 1824 (see equation 3.3). For the second convolutional layer, the same number of feature maps are generated for both models (32) with geometries 47w x 14h (nvidia2) and 37w x 27h (nvidia1). The number of trainable parameters is 19232 for both models. In the third convolutional layer, 48 feature maps are generated for nvidia2 and 64 feature maps are generated for nvidia1. The feature map geometries are 22w x 5h (nvidia2) and 17w x 12h (nvidia1) with 38448 and 51264 trainable parameters respectively. This is accounted for by the different number of feature maps in the current layer, term d in 3.3. What becomes noticeable in the third convolutional layer are the feature map aspect ratios. nvidia1 started at 160/120 = 1.33:1 and is at this point 17/12 = 1.4:1, while nvidia2 started at 200/66 = 3:1 and is now 22/5 = 4.4:1, meaning it is increasingly more panoramic, while nvidia1 is still relatively square in comparison.

In the fourth convolutional layer, both models generate 64 feature maps with geometries 20w x 3h (nvidia2) and 15w x 10h (nvidia1). The number of trainable parameters is 27712 and 36928 respectively on account of term k in 3.3 being different, that is, on account of the number of feature maps in the previous layer being different. In the fifth convolutional layer, both models generate 64 feature maps with geometries 18w x 1h (nvidia2) and 13w x 8h (nvidia1) with the same 36928 number of trainable parameters. The aspect ratios in this final convolutional layer are 18:1 and 1.6:1 respectively. In other words, the nvidia2 feature map is an 18 pixel wide by one pixel high sliver. In the following flattened layer, nvidia2 and nvidia1 have 1152 and 6656

inputs respectively. This is given in 5.1:

$$m = w \times h \times k \tag{5.1}$$

where m is the number of inputs in the flattened layer, k is number of feature maps and w is width, h is height of feature map in the previous convolutional layer. In the next fully connected layer both networks have 100 units. nvidia2 and nvidia1 have 115300 and 665700 trainable parameters respectively. The difference is due to different number of inputs in the flattened layer, term m in 3.4. In the next fully connected layer, both networks have 50 units and 5050 trainable parameters on account of both having the same number of units in the current and previous layer. At this point, there is a difference in the network design. nvidia2 has one additional fully connected layer with 10 units, and 510 trainable parameters (50x10, and ten added to account for bias term). The final output layer is the same for both models, with two units (steering and throttle outputs). The number of trainable parameters is 22 and 102 for nvidia2 and nvidia1, respectively. The total number of trainable parameters being 245,026 and 817,028 for nvidia2 and nvidia1 respectively. Proportionally, nvidia2 is less than 1/3 the size of nvidia1, noting the original nvidia2 (NaokiNet) architecture has 36 feature maps in the second convolutional layer and 21636 trainable parameters, then 43248 trainable parameters in the third convolutional layer, leading to a total of 252,230 trainable parameters. The conclusion is image and feature map geometry choices in the nvidia2 network accounted for the better performance in the rain. This opens a number of interesting investigation avenues such as what would be the minimum number of parameters required to train a network to perform the same self-driving in the rain task, what "sliver" geometries would most favour the task and what other scenarios favour a horizontal sliver geometry in the last convolutional layer feature map, as well as what scenarios would favour a vertical sliver and what could be achieved with ensembles (Ren, Zhang, and Suganthan, 2016) combining models with both horizontal and vertical sliver geometry feature maps.

The smoother steering nvidia1 network showed self-driving around the Generated Track (Figure 4.16) in torrential rain, having crashed in light and heavy rain (Figures 4.14 and 4.15) could perhaps also be leveraged in ensemble models, where the smoother steering nvidia1 model, with the bigger last convolutional layer feature map, being used for overall steering, until the "jerkier" sliver model showed a spike that was acted upon to stop the vehicle from driving off the road, the steering then returned to the smoother model. The best performing nvidia2 and nvidia1 models were trained in 1m39s and 1m23s over 5 epochs, (accuracy and validation plots shown in E.22 and E.13) with validation accuracies converging within 2 epochs, that is, the function being approximated by the network converging quickly to a local minima. The model size for both nvidia2 and nvidia1 on disk is 9.5M. Given fast training times, relatively small datasets, and small model size, the result makes another case for the use of ensembles of CNNs, each dealing with a specific task.

Training accuracy metrics were found not to be a good performance indicator. As an example, nvidia2 E.5.67 has a marginally higher validation accuracy than the best performing E.5.62 (see plots E.24 and E.22), however, the former crashes. The difference between the former and the

later being the former has zero centered pixel values presented to the network and no 10-unit last fully connected layer. The lesson learnt was qualitative analysis had to be carried out for each model with simulator testing. This creates an issue with employing schemes such as grid search (Bergstra and Bengio, 2012) or automated machine learning (Feurer et al., 2015), if relying on training accuracy alone.

The quantitative evaluation metric g_s (3.5) proposed in this study proved to accurately reflect qualitative analysis for dry weather self-driving, as the best observed model obtained the best g_s score (table 4.4, row 1). For wet weather it proved inadequate. The nvidia1 model, known to crash when running real time predictions in all rain conditions with intensity multiplier 1 and 4 (Figures 4.14 and 4.15), obtained better g_s scores (rows 5, 6 and 7) in the rain than the nvidia2 model (rows 2, 3, 4) for SDSandbox log data (not self-driving in real time) on the Generated Track. On the Generated Road, nvidia2 outperformed nvidia1 for all weather. The original expectation was the g_s score being able to generate lower scores for the best performing model in all cases. A suggestion for future work is to evaluate a *sliding window* approach as used in digital signal processing, where a sequence of frames is analysed at a time, the g_s score for the sequence could then be noted, the window moved forward by one frame, and so on until the entire sequence is inspected. The expectation being that events such as nvidia1 model crashes shown in 4.14 could be quantified by a higher-than-threshold g_s score in the window sequence, each crash adding up into a penalty term. Also, the function name choice was unfortunate, as higher scores mean worse steering. An improved G_s metric is suggested in 5.2:

$$G_s(p, g) = \left(\frac{\sum_i^N |p(i) - g(i)|}{N} \times n_c + C_c \right)^{-1} \tag{5.2}$$

where C_c is the crash count penalty term, and the exponentiation leads to lower results translating into higher scores.

The project achieved its aim in providing evaluation for variations of one specific CNN architecture (NVIDIA PilotNet). There was not sufficient time to attempt ResNet, Google LeNet and VGG architetures, or to get the original AlexNet working as deep regression networks, modifying the output layer to perform regression istead of classification. It took longer than expected, 4 months (1.1), where original estimate was 1 (A) to get a model self-driving around the Generated Track. If the project was to start again with the current working setup, additional architectures could be tested. As a result of examining 4 additional real life datasets: Audi, FordAV, Kitti and Udacity, it became clear that for the first three, there was no standard for storing steering angle values, or even storing a steering angle with the acquired image at all, like the FordAV dataset which relied on IMU readings extracted from rosbag files, that required further processing to be converted to a steering angle (B.15.3), unforeseen at the outset of this project. In essence, some public datasets are primarily meant for multi-sensor fusion, not end to end self-driving models. The time invested in evaluating public datasets overran by 9 weeks. It would have been a better use of time to stick to the end to end datasets if the project was to start again. A work diary (work-diary.txt) is provided in the latex files used to format the final dissertation pdf.

6 Evaluation, Reflections and Conclusions

The choice of objectives, tools and techniques enabled and demonstrated the evaluation of self-driving CNNs in the rain. From that perspective, the overall aims of this project were achieved. The main contribution was to create, from pre-existing tools that were successfully integrated and extended, an environment readily available for anyone researching similar topics related to the effect of noise on CNNs, using game engines. The general conclusion is rain-like image noise does affect self-driving CNN performance, and the choice of network architecture and image geometry is important to minimise performance degradation. Aspects of network design that help to deal with such noise were successfully identified by careful network design scrutiny. The effect of network architecture, especially geometry of the last convolutional layer feature maps was established, and the impact such design choice has on the quality of self-driving in the rain, generating smoother or jerkier steering patterns, demonstrated. This achievement did provide some explainability about network design with respect to predictions. A metric for quantitative analysis was proposed and proved partially consistent with qualitative analysis. An improved scheme was then proposed. Network training times and network sizes were investigated, and ensemble models suggested as future work. Experiments were conducted, documented and outputs carefully annotated. Care was also taken in producing visual displays, be it still or moving images, to help inform decisions. Cross-referencing consistency was a prime concern throughout, especially with respect to the results and discussion. Those are the positive outcomes.

One aspect that was not accounted for originally, and lacking in this project was a study of camera characteristics that determine geometric properties of the acquired image, demonstrated here to impact network performance on the image size level alone. Important factors not considered include frame of reference, world frame, camera frame, image plane, image frame, intrinsic parameters and extrinsic parameters (Sala, 2006).

From the original 92 deliverables *deliverables* outlined in D this project managed to deliver D1, D2 and D4 datasets. D3 was not attempted (as over time came the awareness that it may not be good practice to mix datasets, later confirmed with understanding of image geometry and its impact on network architecture. All simulator testing tracks S1, S2 and S3. Model N1 (PilotNet), plus deliverable IDs 13 through 16. Completing a total of 11 out of 92. A total closer to the original goal would have been achieved, had the AlexNet, VGGNet, InceptionNet/GoogleLeNet and ResNet models been adapted for self-driving, which in hindsight does not seem realistic, hence the original goal was overambitious.

The naming conventions used in throughout this project, for data directories, models, tcpflow logs, etc, could have benefited from some time invested in devising a more human-readable and self-documenting scheme, as well as code documentation. An effort was made to this end while the research work was being carried out. Attempts to improve experiment documentation can be seen in E.5 where the level of detail and layout increase over time. Still it was laborious sometimes to recall an experiment's setup and results, which could be addressed with additional planning.

Although CNNs are not well understood, they are heavily used. A lot of effort has gone into creating models originally designed for image classification. Future work could determine if alternative deep models (ResNet, LeNet, VGGNet) could be used for self-driving. These models have been applied successfully to multi-class classification problems. Assuming the network design somehow is optimized for this type of task, it would be interesting to transform regression problems into multi-class classification problems to make optimal use of such networks. This could be attained by quantizing (assigning a continuous value within a defined range to a discrete value) and binning the outputs, subject to the quantized values having acceptable steering precision, in the case of a self-driving car, the minimum acceptable steering change. The network output would become discrete, and the network still leveraged as a classifier - the original intent. Perhaps such scheme could also produce usable results for other computer vision applications relying on regression models and wishing to use state-of-the-art classification models.

The work plan originally set out in appendix A was changed and an estimate of the revisions are shown in Figure 1.1. In hindsight, the original plan seems to reflect a *waterfall* (Balaji and Murugaiyan, 2012) software development model, where all requirements are defined at the outset, the work then separated into logical blocks and sequenced following a delivery schedule. In practice, many (as opposed to the originally expected few) areas were found to overlap such as selecting (and creating) toolsets, replicating existing and creating alternative models, augmenting data and evaluating models, while creating metrics. The end of October and beginning of November 2020 task overlap reflect the busiest research period, when a working model, qualitative evaluation toolset and a quantitative evaluation metric were obtained. The work followed what could be described as an *agile* process, with short *sprints* and concurrent changes reflected across several tasks. Still, the work plan was essential is conceptualising and structuring the effort needed to complete this project. Ultimately the contributions in explaining network design and self-driving, and evaluation provided in this study could help make self-driving vehicles relying on CNNs in the rain safer.

Bibliography

Abadi, Martín et al. (2016). "Tensorflow: A system for large-scale machine learning". In: *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pp. 265–283.

Acemoglu, Daron and Pascual Restrepo (2018). *Artificial intelligence, automation and work*. Tech. rep. National Bureau of Economic Research.

Agarwal, Siddharth et al. (2020). "Ford Multi-AV Seasonal Dataset". In: *arXiv preprint arXiv:2003.07969*.

Amazon Web Services (2020a). *Amazon SageMaker Ground Truth Overview*. [Online; accessed 20-November-2020]. URL: `https://aws.amazon.com/sagemaker/groundtruth/`.

— (2020b). *Amazon Simple Storage Service Documentation*. [Online; accessed 20-November-2020]. URL: `https://docs.aws.amazon.com/s3/index.html`.

Applanix (2020). *Applanix POS LV*. URL: `https://www.applanix.com/products/poslv.htm` (visited on 2020).

Balaji, S and M Sundararajan Murugaiyan (2012). "Waterfall vs. V-Model vs. Agile: A comparative study on SDLC". In: *International Journal of Information Technology and Business Management* 2.1, pp. 26–30.

Batista, Gustavo EAPA, Ronaldo C Prati, and Maria Carolina Monard (2004). "A study of the behavior of several methods for balancing machine learning training data". In: *ACM SIGKDD explorations newsletter* 6.1, pp. 20–29.

Bennett, Stuart (1993). "Development of the PID controller". In: *IEEE Control Systems Magazine* 13.6, pp. 58–62.

Bergstra, James and Yoshua Bengio (2012). "Random search for hyper-parameter optimization". In: *The Journal of Machine Learning Research* 13.1, pp. 281–305.

Bojarski, Mariusz et al. (2016). *End to End Learning for Self-Driving Cars*. arXiv: `1604.07316 [cs.CV]`.

Bojarski, Mariusz et al. (2020). *The NVIDIA PilotNet Experiments*. arXiv: `2010.08776 [cs.CV]`.

Camber Documentation (2019). *Connecting to the cluster*. [Online; accessed 20-November-2020]. URL: `https://moodle.city.ac.uk/pluginfile.php/1651287/mod_forum/attachment/307780/Cluster\%20access\%20documentation.pdf`.

Chacon, Scott and Ben Straub (2014). *Pro git*. Springer Nature.

Chollet, Francois et al. (2015). *Keras*. URL: `https://github.com/fchollet/keras`.

Cord, A. and Nicolas Gimonet (2014). "Detecting Unfocused Raindrops: In-Vehicle Multipurpose Cameras". In: *IEEE Robotics & Automation Magazine* 21, pp. 49–56.

Cowan, Brent and Bill Kapralos (2014). "A survey of frameworks and game engines for serious game development". In: *2014 IEEE 14th International Conference on Advanced Learning Technologies*. IEEE, pp. 662–664.

Crowston, Kevin (2012). "Amazon mechanical turk: A research tool for organizations and information systems scholars". In: *Shaping the Future of ICT Research. Methods and Approaches*. Springer, pp. 210–221.

Cybenko, George (1989). "Approximation by superpositions of a sigmoidal function". In: *Mathematics of control, signals and systems* 2.4, pp. 303–314.

Dai, Peng, Daniel Sabby Weld, et al. (2011). "Artificial intelligence for artificial artificial intelligence". In: *Twenty-Fifth AAAI Conference on Artificial Intelligence*. Citeseer.

Deng, Jia et al. (2009). "Imagenet: A large-scale hierarchical image database". In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee, pp. 248–255.

DIYRobocars (2020). *For people who want to make and race DIY autonomous cars of any size, from tiny 16th scale to full-size*. [Online; accessed 07-October-2020]. URL: https://diyrobocars.com.

DonkeyCar (2020). *An opensource DIY self driving platform for small scale cars*. [Online; accessed 07-October-2020]. URL: https://diyrobocars.com.

Dosovitskiy, Alexey et al. (2017). "CARLA: An Open Urban Driving Simulator". In: *Proceedings of the 1st Annual Conference on Robot Learning*, pp. 1–16.

Dumoulin, Vincent and Francesco Visin (2018). *A guide to convolution arithmetic for deep learning*. arXiv: 1603.07285 [stat.ML].

Elson, Jeremy (2013). *tcpflow - TCP flow recorder*. https://linux.die.net/man/1/tcpflow. Accessed: 2020-15-12.

Epic Games (Apr. 25, 2019). *Unreal Engine*. Version 4.22.1. URL: https://www.unrealengine.com.

Feurer, Matthias et al. (2015). "Efficient and robust automated machine learning". In: *Advances in neural information processing systems*, pp. 2962–2970.

Fleetwood, Janet (Apr. 2017). "Public Health, Ethics, and Autonomous Vehicles". In: *Am J Public Health* 107(4).9, 532–537. DOI: 10.2105/AJPH.2016.303628. URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5343691/.

Fu, X. et al. (2017). "Clearing the Skies: A Deep Network Architecture for Single-Image Rain Removal". In: *IEEE Transactions on Image Processing* 26.6, pp. 2944–2956. DOI: 10.1109/TIP.2017.2691802.

Garfinkel, Simson L and Michael Shick (2013). *Passive tcp reconstruction and forensic analysis with tcpflow*. Tech. rep. NAVAL POSTGRADUATE SCHOOL MONTEREY CA.

Garg, Kshitiz and Shree K Nayar (2007). "Vision and rain". In: *International Journal of Computer Vision* 75.1, pp. 3–27.

Geiger, Andreas et al. (2013). "Vision meets robotics: The kitti dataset". In: *The International Journal of Robotics Research* 32.11, pp. 1231–1237.

Geyer, Jakob et al. (2020). "A2D2: Audi Autonomous Driving Dataset". In: arXiv: 2004.06320 [cs.CV]. URL: https://www.a2d2.audi.

Glorot, Xavier and Yoshua Bengio (2010). "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256.

Grigorescu, Sorin et al. (Apr. 2020). "A survey of deep learning techniques for autonomous driving". In: *Journal of Field Robotics* 37.3, 362–386. ISSN: 1556-4967. DOI: 10.1002/rob.21918. URL: http://dx.doi.org/10.1002/rob.21918.

Haas, John K (2014). "A history of the unity game engine". In:

Harris, Charles R. et al. (Sept. 2020). "Array programming with NumPy". In: *Nature* 585.7825, pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

He, Kaiming et al. (2015). *Deep Residual Learning for Image Recognition*. arXiv: 1512.03385 [cs.CV].

Hinton, Geoffrey E. et al. (2012). *Improving neural networks by preventing co-adaptation of feature detectors*. arXiv: 1207.0580 [cs.NE].

Hornik, Kurt, Maxwell Stinchcombe, Halbert White, et al. (1989). "Multilayer feedforward networks are universal approximators." In: *Neural networks* 2.5, pp. 359–366.

Intel (2020). *A Development Sandbox for Data Center to Edge Workloads*. https://software.intel.com/content/www/us/en/develop/tools/devcloud.html. Accessed: 2020-07-06.

Ioffe, Sergey and Christian Szegedy (2015). *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. arXiv: 1502.03167 [cs.LG].

Kazam (2020). *A screencasting program created with design in mind*. [Online; accessed 23-November-2020]. URL: https://launchpad.net/kazam.

Kramer, Tawn (2020). *Sandbox Car Simulator*. [Online; accessed 07-October-2020]. URL: https://github.com/tawnkramer/sdsandbox.

Krawczyk, Bartosz (2016). "Learning from imbalanced data: open challenges and future directions". In: *Progress in Artificial Intelligence* 5.4, pp. 221–232.

Krizhevsky, Alex, Vinod Nair, and Geoffrey Hinton (2009). "CIFAR-10 Canadian Institute for Advanced Research". In: URL: http://www.cs.toronto.edu/~kriz/cifar.html.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton (2012). "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*, pp. 1097–1105.

Kurihata, Hiroyuki et al. (2005). "Rainy weather recognition from in-vehicle camera images for driver assistance". In: *IEEE Proceedings. Intelligent Vehicles Symposium, 2005.* IEEE, pp. 205–210.

LeCun, Y et al. (2004). "Dave: Autonomous off-road vehicle control using end-to-end learning". In: *Courant Institute/CBLL, http://www. cs. nyu. edu/yann/research/dave/index. html, Tech. Rep. DARPA-IPTO Final Report*.

Lecun, Yann et al. (1998). "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE*, pp. 2278–2324.

LeCun, Yann et al. (1998). "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11, pp. 2278–2324.

LeCun, Yann A et al. (2012). "Efficient backprop". In: *Neural networks: Tricks of the trade*. Springer, pp. 9–48.

Li, Fei-Fei, Andrej Karpathy, and Justin Johnson. "CS231n: Convolutional Neural Networks for Visual Recognition 2016". In: (). URL: http://cs231n.stanford.edu/.

Li, Hao et al. (2017). *Visualizing the Loss Landscape of Neural Nets*. arXiv: 1712.09913 [cs.LG].

Lin, Min, Qiang Chen, and Shuicheng Yan (2013). *Network In Network*. arXiv: 1312.4400 [cs.NE].

Lin, Ming, Jaewoo Yoon, and Byeongwoo Kim (2020). "Self-Driving Car Location Estimation Based on a Particle-Aided Unscented Kalman Filter". In: *Sensors* 20.9. ISSN: 1424-8220. DOI: 10.3390/s20092544. URL: https://www.mdpi.com/1424-8220/20/9/2544.

Loeb, Benjamin and Kara M Kockelman (2019). "Fleet performance and cost evaluation of a shared autonomous electric vehicle (SAEV) fleet: A case study for Austin, Texas". In: *Transportation Research Part A: Policy and Practice* 121, pp. 374–385.

Maller, Joe (2020). *Joe Maller: FXScript Reference: RGB and YUV Color*. [Online; accessed 16-December-2020]. URL: http://joemaller.com/fcp/fxscript_yuv_color.shtml.

Mordvintsev, Alexander and K Abid (2014). "Opencv-python tutorials documentation". In: URL: https://buildmedia.readthedocs.org/media/pdf/opencv-python-tutroals/latest/opencv-python-tutroals.pdf.

Nair, Vinod and Geoffrey E Hinton (2010). "Rectified linear units improve restricted boltzmann machines". In: *ICML*.

OpenCV (2020). *OpenCV - Smoothing Images*. [Online; accessed 02-December-2020]. URL: https://docs.opencv.org/master/d4/d13/tutorial_py_filtering.html.

Perez, Luis and Jason Wang (2017). "The effectiveness of data augmentation in image classification using deep learning". In: *arXiv preprint arXiv:1712.04621*.

Pezoa, Felipe et al. (2016). "Foundations of JSON schema". In: *Proceedings of the 25th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, pp. 263–273.

Podpora, Michal, Grzegorz Pawel Korbas, and Aleksandra Kawala-Janik (2014). "YUV vs RGB-Choosing a Color Space for Human-Machine Interaction." In: *FedCSIS (Position Papers)*, pp. 29–34.

Quigley, Morgan et al. (2009). "ROS: an open-source Robot Operating System". In: *ICRA workshop on open source software*. Vol. 3. 3.2. Kobe, Japan, p. 5.

Ren, Ye, Le Zhang, and Ponnuthurai N Suganthan (2016). "Ensemble classification and regression-recent developments, applications and future directions". In: *IEEE Computational intelligence magazine* 11.1, pp. 41–53.

Riggs, William and Sven A Beiker (2019). "Business Models for Shared and Autonomous Mobility". In: *Automated Vehicles Symposium*. Springer, pp. 33–48.

Rumelhart, David E, Geoffrey E Hinton, and Ronald J Williams (1986). "Learning representations by back-propagating errors". In: *nature* 323.6088, pp. 533–536.

Sala, Pablo (2006). *Camera Models and Parameters*. [Online; accessed 20-December-2020]. URL: http://ftp.cs.toronto.edu/pub/psala/VM/camera-parameters.pdf.

Saxe, Andrew M, James L McClelland, and Surya Ganguli (2013). "Exact solutions to the nonlinear dynamics of learning in deep linear neural networks". In: *arXiv preprint arXiv:1312.6120*.

Saxena, Ujjwal (2017). *Automold Road Augmentation Library*. URL: https://github.com/UjjwalSaxena/Automold--Road-Augmentation-Library.

Shibuya, Naoki (2016). *Car Behavioural Cloning*. URL: https://github.com/naokishibuya/car-behavioral-cloning.

Simonyan, Karen and Andrew Zisserman (2015). *Very Deep Convolutional Networks for Large-Scale Image Recognition*. arXiv: 1409.1556 [cs.CV].

Su, Jiawei, Danilo Vasconcellos Vargas, and Kouichi Sakurai (Oct. 2019). "One Pixel Attack for Fooling Deep Neural Networks". In: *IEEE Transactions on Evolutionary Computation* 23.5, 828–841. ISSN: 1941-0026. DOI: 10.1109/tevc.2019.2890858. URL: http://dx.doi.org/10.1109/TEVC.2019.2890858.

Szegedy, Christian et al. (2014). *Going Deeper with Convolutions*. arXiv: 1409.4842 [cs.CV].

Taeihagh, Araz and Hazel Si Min Lim (July 2018). "Governing autonomous vehicles: emerging responses for safety, liability, privacy, cybersecurity, and industry risks". In: *Transport Reviews* 39.1, 103–128. ISSN: 1464-5327. DOI: 10.1080/01441647.2018.1494640. URL: http://dx.doi.org/10.1080/01441647.2018.1494640.

Tellaeche, A. and I. Maurtua (2014). "6DOF pose estimation of objects for robotic manipulation. A review of different options". In: *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, pp. 1–8.

Transmission Control Protocol (Sept. 1981). RFC 793. DOI: 10.17487/RFC0793. URL: https://rfc-editor.org/rfc/rfc793.txt.

Udacity. "Udacity Self-Driving Car Driving Data 10/3/2016 (dataset-2-2.bag.tar.gz)". In: (). URL: https://github.com/udacity/self-driving-car.

— (2017). *Udacity Car Simulator*. [Online; accessed 07-October-2020]. URL: https://github.com/udacity/self-driving-car-sim.

Van Rossum, Guido and Fred L Drake Jr (1995). *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam.

Vukovic, Maja (2009). "Crowdsourcing for enterprises". In: *2009 congress on services-I*. IEEE, pp. 686–692.

Webster, Dereck D and Toby P Breckon (2015). "Improved raindrop detection using combined shape and saliency descriptors with scene context isolation". In: *2015 IEEE International Conference on Image Processing (ICIP)*. IEEE, pp. 4376–4380.

West, Darrell M (2016). "Moving forward: self-driving vehicles in China, Europe, Japan, Korea, and the United States". In: *Center for Technology Innovation at Brookings: Washington, DC, USA*.

Wiesler, Simon and Hermann Ney (2011). "A convergence analysis of log-linear training". In: *Advances in Neural Information Processing Systems*, pp. 657–665.

Wikipedia contributors (2020). *Torrent file*. [Online; accessed 07-October-2020]. URL: https://en.wikipedia.org/wiki/Torrent_file.

Yoneda, Keisuke et al. (2019). "Automated driving recognition technologies for adverse weather conditions". In: *IATSS research* 43.4, pp. 253–262.

Yurtsever, Ekim et al. (2020). "A Survey of Autonomous Driving: Common Practices and Emerging Technologies". In: *IEEE Access* 8, 58443–58469. ISSN: 2169-3536. DOI: 10.1109/access.2020.2983149. URL: http://dx.doi.org/10.1109/ACCESS.2020.2983149.

Zang, S. et al. (2019). "The Impact of Adverse Weather Conditions on Autonomous Vehicles: How Rain, Snow, Fog, and Hail Affect the Performance of a Self-Driving Car". In: *IEEE Vehicular Technology Magazine* 14.2, pp. 103–111. DOI: 10.1109/MVT.2019.2892497.

Zhang, Chiyuan et al. (2017). *Understanding deep learning requires rethinking generalization*. arXiv: 1611.03530 [cs.LG].

A RPMI Project Proposal Proposal

CITY
UNIVERSITY OF LONDON
EST 1894

Evaluation of self-driving cars using CNNs in the rain

Individual Project Proposal

Daniel Sikar - MSc Data Science PT2
Supervisor: Artur Garcez

May 10, 2020

1 Introduction

Land vehicles that can drive autonomously, without human intervention, have been increasingly researched in the last 4 decades, harnessing the development in computer hardware, where decreasing size and increasing computational power allow such vehicles to carry mobile computing systems capable of performing the signal and algorithmic processing required to successfully self-drive along a path. Parallel to this development, improvements in one area of research in artificial intelligence (AI) known as machine learning has generated models known as convolutional neural networks (CNNs), successfully applied to the field of computer vision, where current state-of-the-art technology has surpassed human accuracy. Computer vision is a core component of self-driving, which relies heavily on images.

From the Stanford Cart, a camera-equipped roving robot, with a remote time-sharing computer as its ”brain”, and ”taking about five hours to navigate a 20 meter course” (Turk et al. 1988) to the current Tesla fleet with billions of accumulated self-driven miles (Karpathy 2020), self-driving cars are an increasing presence on public roads.

Automakers such as Tesla, Nissan, Audi, General Motors, BMW, Ford, Honda, Toyota, Mercedes and Volkswagen, and technology companies such as Apple, Google, NVidia and Intel, are currently researching self-driving vehicles (Ni et al. 2020). This involves a number of tasks related to decision-making and resulting motion control: path planning, scene classification, obstacle detection, lane recognition, pedestrian detection, traffic signage detection (including traffic lights). Reliability of self-driving cars under changing weather conditions is one important factor, considered since the earlier days of research (Turk et al. 1988).

Intel subsidiary MobileEye, NVIDIA and Tesla have developed, and continue to develop dedicated self-driving compute platforms, EyeQ, NVIDIA DRIVE AGX and HW respectively. All three platforms have custom-designed processors, optimised for AI computing. Dedicated inputs receive camera data, and sensor data to measure object proximity, using a combination of radar, sonar and lidar (radio, sound and light waves respectively). These inputs translate into three-

dimensional position and orientation information used for path planning. Tesla has used both MobileEye and NVIDIA platforms, before developing its own (Wikipedia contributors 2020).

Elon Musk, co-founder and CEO of Tesla, with respect to self-driving cars states that "(...) right now AI and Neural Nets are used really for object recognition (...), identifying objects in still frames and tying it together in a perception path planning layer thereafter. (...) Over time I would expect that it moves really to (...) video in, car steering and pedals out" (Musk 2019).

Based on this scenario, our **research question** is how do different CNN architectures compare in the rain, the **purpose of this research** is "to find evidence to inform practice" (Oates 2006) on autonomous vehicles transitioning to a CNN and computer-vision-only solution. The **product of this research** is an evaluation and the **intended beneficiary** is primarily the author, hoping this work will create academic and professional opportunities, as well as anyone else researching in the area that may use this work as a stepping stone or starting point.

2 Critical Context

A CNN is a class of artificial neural network, or simply neural network, having a basic unit known as a neuron, a mathematical model developed by McCulloch and Pitts (1943), based on biological models of the human brain. Neurons are defined as real numbers, in the form of weighted inputs and a bias, and an activation function, which, given a sum of input values multiplied by weights, plus the bias, generates an output value. The combination of weights and bias represents an encoding, the biological equivalent being a memory.

Based on the McCulloch-Pitts neuron, Donald Hebb (1949) created the first learning algorithm, that enables a neural network consisting of one single neuron to "learn", or encode, a memory, through an iteration process until, given a cost (or error) function, a set of weights and bias is found such that the same desired output is produced after a number of iterations, and the weights and bias reach stable values while the error is minimized. The Hebbian learning algorithm was able to learn simple tasks such as how to compute the OR truth table, but not more complex tasks such as the XOR truth table. This hindrance is known as a linear separability problem, where, using the inputs as coordinates, the output classes (0,1) of the XOR truth table plotted on a Cartesian plane, cannot be separated by a straight line. Solutions were eventually found, one involving the addition of another layer containing one neuron, known as a "hidden" layer, plus a connection between neurons, plus connections between inputs and the hidden layer neuron. The intuition being, a neural network with more neurons and more connections is able to learn more complex tasks. Such neural network architectures are known as multi-layer perceptrons (MLP) (Garcez 2018). In the model previously described, inputs are multiplied by weights. In the CNN model, inputs are "convolved" with weights. The concept is borrowed from the digital signal processing domain, where a vector known as a filter or kernel, is combined with a signal to generate a filtered output signal. The operation is expressed by:

$$conv(s1, s2)[t1] = \sum_{t=0}^{N2-1} s1[t1 - t]s2[t] \tag{1}$$

where $s1$ is the input signal, $s2$ is the kernel, $N2$ is the length of $s2$ and $t1$ is the time when input signal $s1$ was acquired. Convolution is similar to cross-correlation (where a measure of similarity between two signals is obtained) except convolution involves "flipping" one of the inputs. This can be observed by the indexing in $s1[t1 - t]$ (Pauwels and Weyde 2020).

For the case of a two-dimensional input and kernel, the operation takes the form:

$$J(x,y) = K * I = \sum_{n,m} K(n,m)I(x-n, y-m) \tag{2}$$

Where J is the convolved signal, K is the kernel, I is the input signal, and n, m are the kernel indexes. We see by the input signal indexing $I(x-n, y-m)$ that both input signal axes are "flipped".

A typical CNN architecture will have a number of convolutional layers, following by a fully connected MLP, that is, where every neuron is connected to each other. The convolutional layers are able to compress the input, without losing discriminative information, into a lower dimensional space, where different input categories can be efficiently represented. The fully-connected MLP layers then perform classification. Convolutional layers in neural networks with the ability to "self-organize", were introduced by the neocognitron network, proposed by Fukushima 1980, particularly efficient for image pattern recognition.

Finding optimal weights and biases for a neural network in a large search space is a mathematically intractable problem that can be solved by the use of back-propagation and gradient descent algorithms, like proposed by Rumelhart, Geoffrey E. Hinton, and Williams 1986 and Lecun et al. 1998, which lead to the wide adoption of CNNs.

The concepts described have been implemented in several machine learning libraries such as Facebook's PyTorch, Google's Tensorflow and MATLAB toolkits. Designing a network from scratch can amount in some cases to writing a few lines of code.

Self-driving vehicles have generally relied on designs where various sensor inputs are combined to generate a control signal (Turk et al. 1988) such as the pioneering Autonomous Land Vehicle (ALV) (Figure 1) . The ALV computer vision system, Vision Task Sequencer (VITS), treats the task of identifying the road ahead as a general classification problem with each pixel in any given image belonging to one of two classes, road and non-road. This is achieved by feature extraction, clustering and segmentation. Feature extraction is the process of engineering features that allow classes to be distinguished. Clustering partitions the feature space into mutually exclusive regions and segmentation assigns each image pixel to one regions The goal being to define regions that can be separated by a hyperplane (with dimension of the feature space minus one) boundary. Colour is used as the feature space. Other image features such as texture, saturation and reflectance from the laser scanner were not used.

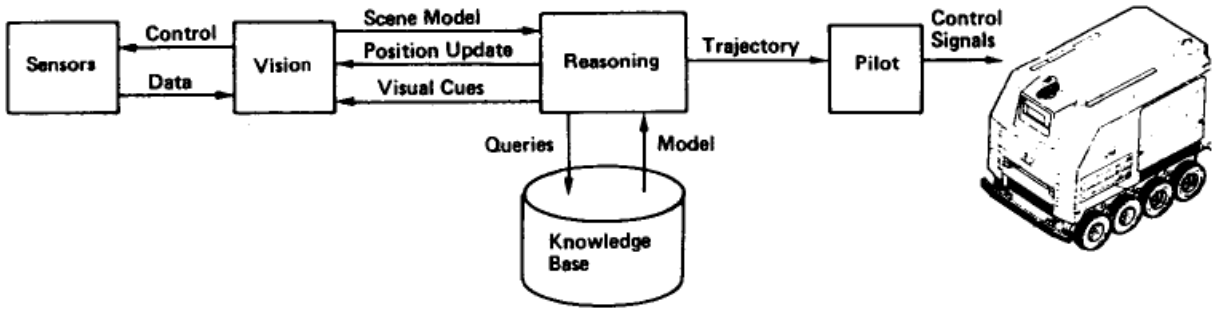

Figure 1: The ALV system configuration

Back-propagation was used to train the neural network for the next-generation ALV, the ALVINN, Autonomous Land Vehicle In a Neural Network (Pomerleau 1989), which has a 3-layer network

designed for the task of road following, taking images from a camera and a laser range finder (such as lidar) as input and producing the direction the vehicle should travel in order to follow the road as output. The neural network architecture consists of 30x32 video inputs, 8x32 laser range finder inputs 29 hidden units, one road intensity feedback unit and 45 direction output units. The road intensity feedback unit indicates whether the road is lighter or darker than the non-road in the previous image. The network is fully connected, that is, each of the input units is fully connected to the hidden layer, and each unit of the hidden layer is fully connected to the output layer. These improvements eventually lead to the RALPH system, used for a self-driving vehicle that in 1995 drove almost entirely autonomously across the USA.

The NVIDIA CNN (Figure 2) maps raw pixels from a single front-facing camera directly to steering commands. (Bojarski et al. 2016). A CNN image-in, steering-out had been previously implemented in a sub-scale radio control (RC) vehicle by the Defense Advanced Research Project Agency (DARPA) Autonomous Vehicle (DAVE) project. The end to end approach avoids the need for manual feature-extraction, clustering, segmentation and rules-based programming used to combined inputs and generate an output.

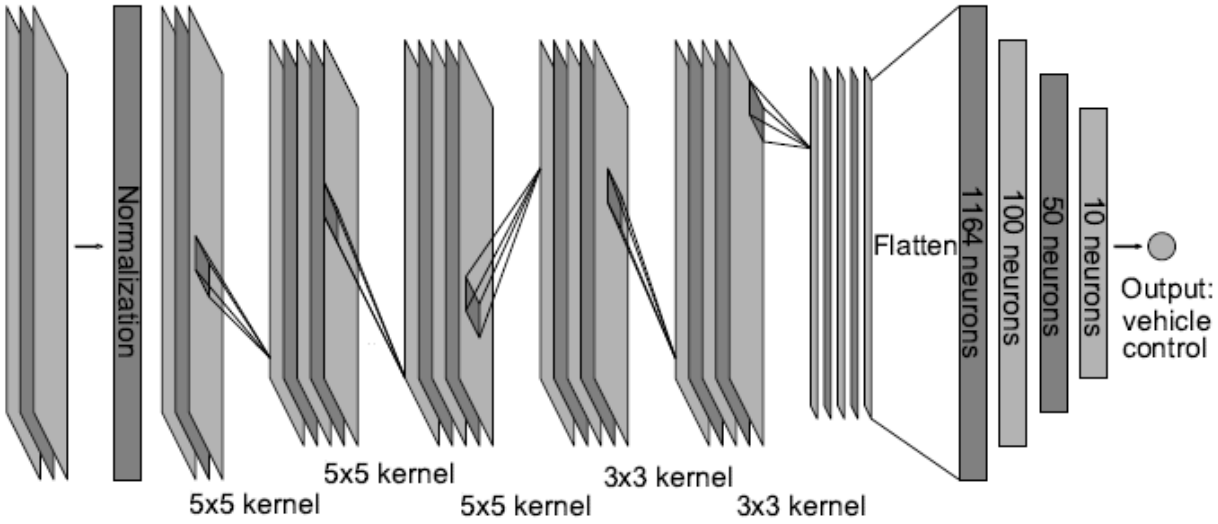

Figure 2: NVIDIA End-to-End CNN architecture

The NVIDIA CNN has approximately 27 million connections and 250 thousand parameters. The network consists of 9 layers, including a normalization layer, 5 convolutional layers and 3 fully connected layers leading to an output control value which is the inverse of the turning radius $\frac{1}{r}$ to avoid a singularity when driving straight (infinite radius). To remove a bias towards driving straight, a higher proportion of frames representing curves were added to the training data. The data was also augmented by random rotations to teach the network how to recover from unexpected shifts. The network runs on dedicated graphical processing units (GPU) hardware able to apply far more data and computational power to the task compared to previously used central processing units (CPU). This is the baseline network we are aiming to implement as a starting point to our evaluation.

3 Approaches: Methods & Tools for Analysis & Evaluation

In this section we describe the components shown in our work breakdown structure (Figure 3)

3.1 Literature Survey

We have conducted the initial literature survey using Google searches that usually led to the required publication. When we could not access an article, we used the library website and were generally able to access articles via institutional login.

3.2 Data & Tools

We identified 4 image datasets, 3 environments to train and test our models, and one image labelling crowdsourcing tool, to find relevant "rainy" sections in our data.

The **Ford Multi-AV Seasonal Dataset** is a multi-sensor dataset collected by a fleet of Ford autonomous vehicles at different days and times during 2017-18. They contain inertial measurement units (IMU) that provide orientation which are our required steering angles.

The **Audi dataset** provides 40,000 frames with semantic segmentation image and point cloud labels, of which more than 12,000 frames also have annotations for 3D bounding boxes. In addition, sensor data (approx. 390,000 frames) for sequences with several loops, recorded in three cities. The data need to be evaluated for steering angle labels

The **KITTI dataset** and benchmarks for computer vision research in the context of autonomous driving. The dataset has been recorded in and around the city of Karlsruhe, Germany using the mobile platform AnnieWay and has IMU labels.

The **Udacity self-driving datasets** two datasets of interest, containing labeled images with IMU and lidar values.

We intend to locate segments where rain is present with crowdsourcing **Amazon Mechanical Turk**. The expected outcome of this step is at least dozens of sequences containing rain.

We intend to use data augmentation to create synthetic data, with properties that would emulate rain characteristics, such as superimposing rain drops to images, then adding effects such as blurring, reflection and diffusion. The expected outcome of this step is at least dozens of sequences containing the augmented data plus metadata indicating the level of each applied effect.

Three environments have been identified to run CNN training and testing: the **Intel DevCloud**, the **NVidia open DRIVE Constellation** platform, and the the **Udacity Self-Driving Car Simulator**.

For initial network design we have identified **PyTorch**, **Keras** (Python Tensorflow wrapper) and **MATLAB** toolkits as potential tools.

3.3 Develop Models

Our data consists of videos, taken by cameras mounted on moving vehicles, which can be interpreted as sequences of still images taken at fixed-time intervals. Each still image is labelled with a quantity we call θ, that represents a direction from 0 to 359 degrees, in which the vehicle was travelling at the moment the image was stored. $\Delta\theta$ between two images, taken at intervals i and $i+n$, represents the amount of steering that was applied to the vehicle after n intervals. Therefore

the assumption is we are dealing with a regression problem, where, given a sequence of images as described, we want our models to approximate θ, keeping the autonomous vehicle on the road.

Our starting point is the NVIDIA CNN (Figure 2) model. Thereafter we expect to implement a number of alternative models to evaluate individually and compare to each other. The alternative models we are looking to implement are:

AlexNet - five convolutional layers, some of which are followed by max-pooling layers, and three fully-connected layers with a final 1000-way softmax. To reduce overfitting in the fully-connected, dropout is used as a regularization method (Krizhevsky, Sutskever, and Geoffrey E Hinton 2012).

VGGNet - implementation showing increased depth up to 16-19 weight layers can be achieved by using an architecture with very small (3 x 3) convolution filters (Simonyan and Zisserman 2015).

Inception-v1, Inception-v2, Inception-v3 - variations of a deep convolutional neural network architecture, one implementation being GoogleLeNet, using the Network-in-Network approach in order to increase the representational power of neural networks, 1 x 1 convolutional layers are added to the network, acting as dimension reduction modules to remove computational bottlenecks, that would otherwise limit the size of the network. This allows for not just increasing the depth, but also the width of the network without a significant performance penalty (Szegedy et al. 2015).

ResNet - residual learning framework to optimize training of networks, where layers are reformulated as learning residual functions with reference to the layer inputs, instead of learning unreferenced functions (He et al. 2015).

3.4 Evaluation

Developing the evaluation metric is part of this project. We are specifically interested in the steering aspect, and how much error (oversteering or understeering) would define an accident.

The initial step is to, once the networks have been trained and tested on our original ”dry” data, obtain a ground truth, where no accidents (autonomous vehicles driving off the road) are registered. We will then use our ”rainy” image sequences, plus augmented data edited in, and replacing original ”dry” data. The expectation is that at a given threshold of added rainy and augmented data, the autonomous network under test will output steering angles for a sufficient amount of frames that would cause the car to go off the road.

Once a few of these cases have been identified for our baseline model, the sequences will be used for all networks, and an evaluation made which should answer our research question.

3.5 Complete Report

The report writing component of this work is intended to be a continuous process, from the project start date until estimated completion date. The final reporting component being the first draft, then two rounds of reviews and addressing comments. The deliverable for this component is the final project dissertation.

4 Work Plan

Our work plan consists of a work breakdown structure (Figure 3) and a Gantt chart (Figure 4). We identified 5 principal components, each successive component has a dependency on, and cannot start before, the previous component being completed, which represents a milestone.

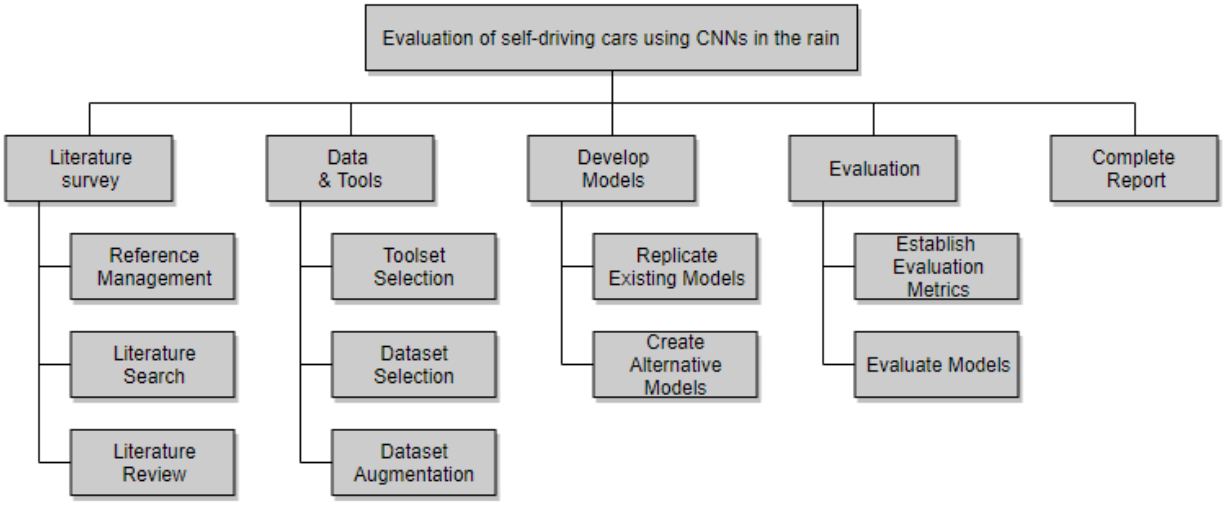

Figure 3: Work breakdown structure

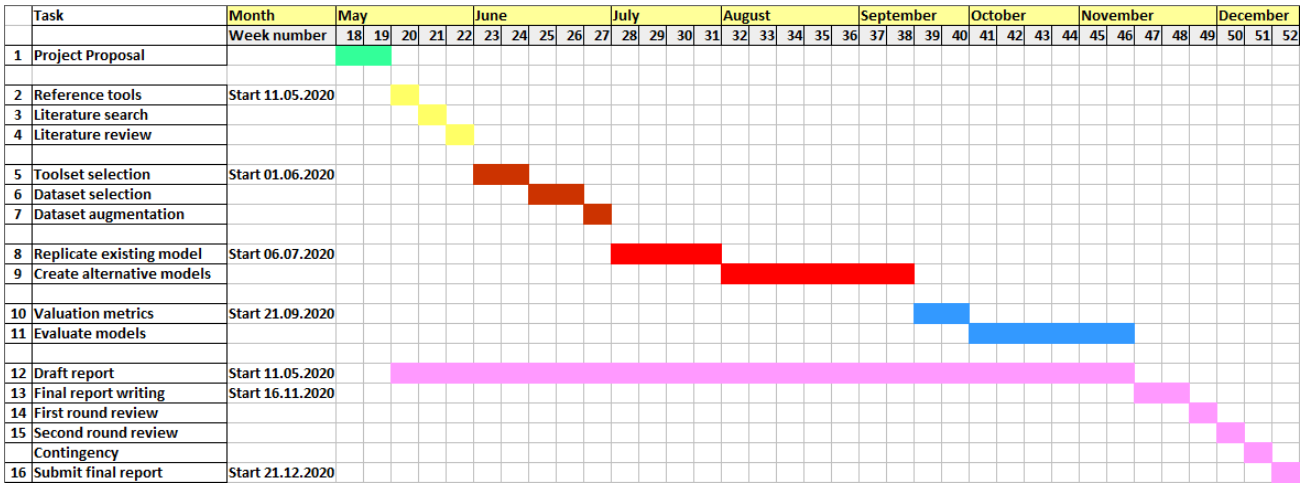

Figure 4: Gantt chart

5 Risks

To generate our risk register (Table 1), as suggested in Dawson 2009 we use:

$$I = L * C \tag{3}$$

where Likelihood L is categorized according to three-point scale Low/Medium/High, Consequence C according to five-point scale Very Low/Low/Medium/High/Very High and Risk Impact I is the product.

Description	**L**	**C**	**I**	**Mitigation**
Insufficient domain knowledge	2	4	8	Extend scope for literature survey and toolset selection and decrease scope for alternative models
Cannot source rain image sequences	1	5	5	Use augmented rain image sequences only
Unable to replicate NVIDIA CNN model	2	3	6	Use Udacity Unity engine as a new starting point
Sequence learning approach does not work or is too complex to implement.	2	4	8	Use single labelled image as input
Unable to create alternative models	2	4	8	Use NVIDIA CNN only, comparing ground truth ”dry” with ”rainy” data
Accidental loss of data.	2	5	10	Keep all source code, reporting and data online (github.com, overleaf.com, aws.amazon.com, camber.city.ac.uk and devcloud.intel.com)
Continued COVID-19 lockdown disruption	2	3	6	Scale down project
Final report delay	2	5	10	One additional week has been added for contingency

Table 1: Risk register

6 Ethical, Legal and Professional Issues

Having completed *Part A: Ethics Checklist* of the Research Ethics Review Form, as determined by the Computer Science Research Ethics Committee (CSREC 2020), we find this work complies with research ethics guidelines and does not require ethical approval.

References

Bojarski, Mariusz et al. (2016). "End to End Learning for Self-Driving Cars." In: *CoRR* abs/1604.07316. URL: http://dblp.uni-trier.de/db/journals/corr/corr1604.html#BojarskiTDFFGJM16.

CSREC (2020). *Research Governance Framework*. Department of Computer Science Research Ethics Committee, City, University of London.

Dawson, Christian W. (2009). *Projects in Computing and Information Systems: A Student's Guide*. 2nd. Pearson Prentice Hall. ISBN: 978-0-273-72131-4.

Fukushima, Kunihiko (1980). "Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position". In: *Biological Cybernetics* 36, pp. 193–202.

Garcez, Artur (2018). *INM427 Neural Computing Lecture Notes*.

He, Kaiming et al. (2015). *Deep Residual Learning for Image Recognition*. arXiv: 1512.03385 [cs.CV].

Karpathy, Andrej (2020). *AI for Full-Self Driving, Scale ML Conference*. URL: https://www.youtube.com/watch?v=hx7BXih7zx8&t=1m30s (visited on 05/01/2020).

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton (2012). "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., pp. 1097–1105.

Lecun, Yann et al. (1998). "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE*, pp. 2278–2324.

Musk, Elon (2019). *TESLA Autonomy Day Event*. URL: https://www.youtube.com/watch?v=tbgtGQIygZQ&t=2h25m52s (visited on 05/01/2020).

Ni, Jianjun et al. (2020). "A Survey on Theories and Applications for Self-Driving Cars Based on Deep Learning Methods". In: *Applied Sciences* 10.8. ISSN: 2076-3417. DOI: 10.3390/app10082749. URL: https://www.mdpi.com/2076-3417/10/8/2749.

Oates, Briony J. (2006). *Researching information systems and computing*. 1st. SAGE Publications. ISBN: 9781446235447.

Pauwels, Johan and Tillman Weyde (2020). *INM378 Digital Signal Processing and Audio Processing Lecture Notes*.

Pomerleau, Dean A. (1989). "ALVINN: An Autonomous Land Vehicle in a Neural Network". In: *Advances in Neural Information Processing Systems 1*. Ed. by D. S. Touretzky. Morgan-Kaufmann, pp. 305–313. URL: http://papers.nips.cc/paper/95-alvinn-an-autonomous-land-vehicle-in-a-neural-network.pdf.

Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams (1986). "Learning Representations by Back-propagating Errors". In: *Nature* 323.6088, pp. 533–536. DOI: 10.1038/323533a0. URL: http://www.nature.com/articles/323533a0.

Simonyan, Karen and Andrew Zisserman (2015). "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *International Conference on Learning Representations*.

Szegedy, Christian et al. (2015). "Going Deeper with Convolutions". In: *Computer Vision and Pattern Recognition (CVPR)*. URL: http://arxiv.org/abs/1409.4842.

Turk, M. A. et al. (1988). "VITS-a vision system for autonomous land vehicle navigation". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 10.3, pp. 342–361.

Wikipedia contributors (2020). *Tesla Autopilot*. [Online; accessed 02-May-2020]. URL: https://en.wikipedia.org/wiki/Tesla_Autopilot.

B Methods

Procedures used to set up experiments are detailed in this appendix.

B.1 Generating graphs

Two graphs are used, steering angle histograms and steering angle plots. A set of functions have been written for this purpose in steerlib.py. For example to generate a histogram of steering angles from a

B.2 Network architectures for future reference

B.2.1 ResNet

Discuss the concept of "residuals". ResNet makes use of skip-connections (spelling), these "network architecture designs (e.g., skip connections) produce loss functions that train easier", producing smoother loss surfaces (Li et al., 2017) The "Degradation problem itself stems from the fact that it is easier to learn 0 than to learn 1" https://www.youtube.com/watch?v=jio04YvgraU "if an identity mapping were optimal, it would be easier to push the residual to zero than to fit an identity mapping by a stack of nonlinear layers" He et al., 2015. Explain what variation A, B or C we will be using (zero padding, weights * x, etc. "Essemble" of ResNets was used. 18 layers seems to be a threshold, beyond that, training becomes difficult. Skip connections make deeper networks, i.e. networks with more layers, easier to train compared to networks without skip connections. Another wording, deeper neural networks (i.e. layer count equal or greater than 20) become increasingly difficult to train, generating poorer accuracy than deep networks with less than 20 layers. This is known as "degradation problem" (citation needed). Skip connections make deeper architectures easier to train, by smoothing the loss function.

B.2.2 GoogleLeNet

This network (Szegedy et al., 2014) was the winning entry in the ILSVCC 2014 Classification Challenge. It uses the LeNet-5 (Lecun et al., 1998) model as a starting point, following the now traditional design of ConvNets with the addition of *Inception* modules as introduced by Lin, Chen, and Yan, 2013 in the *Network in Network* model.

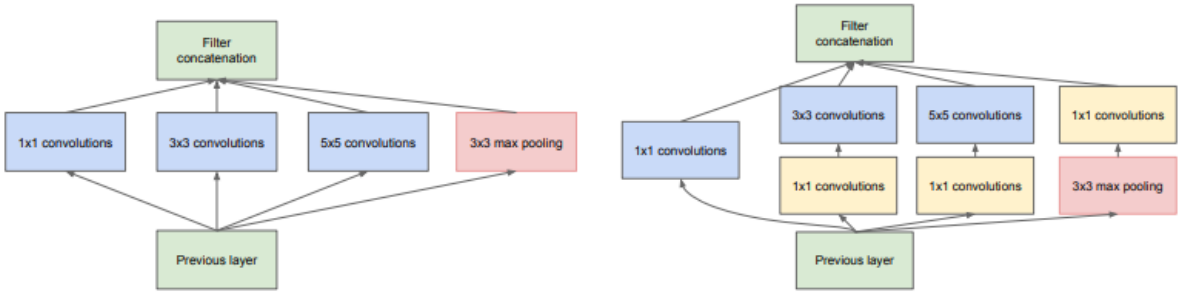

FIGURE B.1: Inception naive (left) and dimension reduction model (right)

The designers of the GoogleLeNet architecture state that power and memory use is important to consider and aim to keep a computation budget of 1.5 billion multiply adds at inference time such that the resulting model could be used in practice at a reasonable cost.

B.2.3 VGGNet

Another one there was not enough time to trial.

B.3 Cleaning SDSandbox data

Data corruption was found with SDSandbox generated datasets where some .json files (containing steering angles and throttle for a corresponding image) were not generated, the missing values leading to training failure. There "orphaned" images were deleted with custom script src/utils/jsonclean.py

B.4 Degradation

Our NVIDIA architecture is very similar to the one proposed by Bojarski et al. 2016. Our alternative AlexNet, GoogleLeNet, VGGNet and ResNet however have much fewer parameters, never tested in full. This is due to the "degradation" problem, where the number of parameters is too large for the dataset, and the network fails to converge.

B.5 Running the car simulator

The main testing environment consisted of a Dell Precision Tower 5810 with a 6 core Intel Xeon Processor and 32GB memory running Ubuntu 18.04. Unity Hub 2.3.2 was installed then run as sudo:

```
$ sudo ./UnityHub.AppImage --no-sandbox
```

The car simulator source was cloned from:

```
$ git clone https://github.com/dsikar/sdsandbox.git
```

The Unity project contained in sdsandbox/sdsim can then added and loaded. Once the menu scene runs, one of 5 circuits can be chosen. Once the chosen circuit is loaded, there are options to *Auto Drive w Rec* (generate test data and steering angle using PID control) or *NN Control over Network* (send images over network and receive predicted steering angles). The first case will output files to ../output directory, the second case will send and listen to network messages. The handshake process has been captured with tcpflow and stored to src/debug/tcpflow_output.txt. The prediction engine starts returning predicted steering angles after the 4th frame sent by simulator as described in section B.6

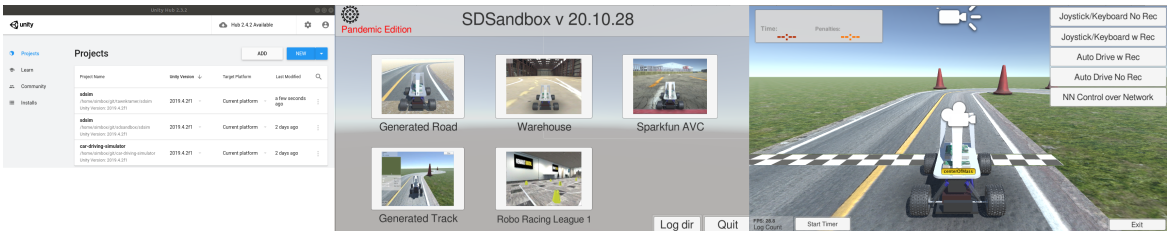

FIGURE B.2: Left to right: Unity Hub, SDSandbox home screen and simulation ready to run

B.5.1 Installing the source code

The code is cloned from:

```
git clone https://github.com/dsikar/sdsandbox
```

This will download the source code to the sdsandbox directory.

B.5.2 Running the Unity Hub application image

Unity Hub is downloaded to disk and started:

```
$ sudo ./UnityAppImage --no-sandbox
```

B.5.3 Adding the SDSanbox project to Unity Hub

Once source code has been cloned, and Unity Hub is running, the project is added by clicking "ADD" and selecting directory sdsandbox/sdsim. The project will then appear as an list entry in Unity Hub and can be opened by clicking.

B.5.4 Running the driving simulator

Once the project is open, the "menu" scene is selected, and the play arrow button clicked. The simulator is ready to run.

B.5.5 Recording one lap of training data

Once the simulator is running, A "Log dir" log directory for storing data outputs is set, then "Generated Track" double-clicked, then "Auto Drive w Rec" option selected. This will set the simulator car going around the track. The control settings (displayed on main screen) are:

```
Max Speed: 1.772174
Prop: 49
Diff: 80
Steer Max: 25
```

The are the maximum speed, the values Prop and Diff for updating steering angle and the maximum steering angle. Once a lap is completed, the Stop button is clicked, then the Play button is clicked to stop the simulator. Images in .jpg format and steering angles stored in .json files are created in sub-directory sdsim/log, and are ready to be moved to the "dataset" directory:

```
$ python prepare_data.py --out-path=../dataset/genTrack
```

The previous command will move all images stored in the sim output directory to a date-labelled folder.

B.6 Network monitoring and debugging with tcpflow

tcpflow (Garfinkel and Shick, 2013) was used to monitor network traffic between car simulator and neural network prediction engine. Once the simulator is setup to run in neural network mode and before the predict_client.py prediction engine starts, tcpflow is launched and set to listen on the loopback interface port 9091, and output network traffic packets to console:

```
$ sudo tcpflow -i lo -c port 9091
```

The prediction script then runs and JSON, (Pezoa et al., 2016) a lightweight data-interchange format, TCP encoded packets may be monitored and debugged. Simulator decoded packets are distinguished by *telemetry* and prediction engine packets by *control* msg_types respectively, as shown in excerpt:

```
127. (...) {"msg_type":"telemetry",(...),"image":(...)
127. (...) {"msg_type": "control", "steering": "-0.09476048" (...)
```

The packets carry the image sent from sim to prediction engine, and returned steering angle prediction.

B.7 Generating Plots

This section describe the tools used to generate plots.

B.7.1 Steering angle comparison

Plots were generated with Jupyter Notebook:

```
src/utils/
```

B.8 Datasets

A template directory structure was created such that downloaded data could be accessed in code with the same paths. The structure exists in the datasets repository and once cloned creates the template directory:

```
$ git clone https://github.com/dsikar/msc-data
$ tree -d msc-data/
msc-data/
 audi
 ford
 kitti
 mechanical-turk
 udacity
 unity
```

B.8.1 Ford AV Dataset

The steering angles can be extracted from .bag files using ROS commands:

```
    # In one terminal, start ros engine
    $ roscore
    # In another terminal, inspect content of bag file
    $ time rosbag info Sample-Data.bag
    (...)
             /imu                 146939 msgs    : sensor_msgs/Imu
    (...)
             /pose_ground_truth   146136 msgs    : geometry_msgs/PoseStamped
             /pose_localized       16100 msgs    : geometry_msgs/PoseStamped
             /pose_raw            146190 msgs    : geometry_msgs/PoseStamped
(...)
    # And subscribe to topic of interest
```

```
$ rostopic echo /imu | tee sample_imu.yaml
# In another terminal, playback bag file
$ time rosbag play --immediate Sample-Data.bag --topics /imu
# Sanity check, count number of acquisitions
$ cat sample_imu.yaml | grep "orientation:" | wc -l
```

The snippet above generates file imu.yaml, with all pose data generated by imu device. From this file we extract the steering angle, which is the z axis (yaw) of the orientation field (TODO check .yaml dialect). Images can be extracted from the same bag file with the Python 2.7 bag_to_images.py script:

```
$ python2 bag_to_images.py Sample-Data.bag ~/git/msc-data/ford/sample/ros/ \
    /image_front_left
```

Each image is an attribute in a dictionary, which also contains seconds (secs) and nano seconds (nsecs) attributes within the header attribute:

```
header:
  seq: 213414
  stamp:
    secs: 1501822147
    nsecs: 684951066
  frame_id: "camera_front_left"
height: 215
width: 414
encoding: "8UC3"
is_bigendian: 0
(...)
```

Thus a timestamp can be obtained for each image extracted. This is done with script parse_yaml_time.py:

While the steering angles are extracted The image can be matched with a steering angle by obtaining the timestamp of image, the full secs

B.8.2 Ford AV Dataset

The Ford Autonomous Vehicle is a From (Applanix, 2020) is a professional-grade, compact,fully integrated, turnkey position and orientation system combining a differential GPS, an inertial measurement unit(IMU) rated with 1° of drift per hour, and a 1024-count wheel encoder to measure the relative position, orientation,velocity, angular rate and acceleration estimates of the vehicle. The Ford AV data set provides the 6-DOF pose (6DOF pose estimation of objects

is the task of estimating the coordinates (X, Y,Z) and rotation angles (Yaw, Pitch and Roll) of an object with respect to a previously established reference coordinate system. (Tellaeche and Maurtua, 2014) estimates obtained by integrating the (linear) acceleration and (angular) velocity.

B.8.3 Kitti

The Kitti dataset TODO add provenance, add description Data format, steering angle description obtained from oxts/dataformat.txt

```
yaw:   heading (rad),       0 = east,  positive = counter clockwise,\
    range: -pi   .. +pi
```

Note, this may not be the steering angle - TBC. So additional work may be needed to obtain. There are some pointers online, a simple approach seems to be: This is also an interesting approach: With python code:

```
def convert_trans_rot_vel_to_steering_angle(v, omega, wheelbase):
  if omega == 0 or v == 0:
     return 0
  radius = v / omega
  return math.atan(wheelbase / radius)
```

Another possible method "Estimation of the Steering Angle Based on Extended Kalman-Filter"

B.9 Automold

The rainy images were created with:

`github.com/dsikar/automold/RainyImagesDissertationPlot.ipynb`

B.10 Development Environments

Three development environments are planned to be used, a local environment to run the Unity3D SDSdanbox application, and two additional environments, the Intel Dev-Cloud (Intel, 2020) and City University's Camber server (Camber Documentation, 2019).

B.10.1 Intel DevCloud

Intel provides a 200GB storage quota. Storage use can be checked with getquota command:

```
$ getquota
199.78 GB out of 200.00 GB (99.89%) used
```

Jobs are queued with qsub command:

```
$ qsub -l nodes=1:gpu:ppn=2 ford_sample_download.sh -l walltime=23:59:59
```

where the actual commands that run e.g. downloading data, training and testing networks are scripted in a batch file e.g. ford_sample_download.sh, train.sh. The queue can be checked with watch command:

```
$ watch -n 1 qstat
```

Jobs can run for a maximum of 24 hours, any job exceeding that execution time is terminated automatically. Jobs can be deleted from queue with qdel command.

B.11 Unity

Unity for Ubuntu is a single image file, that can be downloaded and run:

```
$ ./UnityHub.AppImage
```

This will start the Unity Hub app. A license must be installed by downloading a file, logging into Unity, uploading file then downloading a second license file. This is added to Unity Hub. The next step is to get an editor. Editors are available at:

```
https://unity3d.com/get-unity/download/archive
```

From archive pages, a link is obtained for desired editor (2019.3.0 is this case). To load the editor, Unity Hub must be closed and the re-opened with the obtained link:

```
$ ./UnityHub.AppImage unityhub://2019.3.0f6/27ab2135bccf
```

Source code for the simulator can then be cloned locally

```
git clone https://github.com/tawnkramer/sdsandbox.git
```

The project can then be added to Unity Hub by ADDing and navigating to sdsandbox/sdsim directory. It will then be listed on Unity Hub. Thereafter, Unity Hub can be started with

```
$ ./UnityHub.AppImage
```

B.12 Tensorflow and Keras

The versions used were 2.2.0 and 2.4.3 respectively. To ensure the same versions are installed in all development platforms, run:

```
$ pip install keras --user
$ pip install tensorflow --user
# to check versions
$ python3
>>> import keras
>>> keras.__version__
'2.4.3'
>>> import tensorflow
>>> tensorflow.__version__
'2.2.0'
```

If the modules are already present in the environment, but a lower (earlier) version e.g.:

```
>>> tensorflow.__version__
'1.15.4'
```

it can be upgraded by running:

```
$ pip install --ignore-installed --upgrade tensorflow==2.2.0 --user
```

B.12.1 Recording one loop around a track

To record the images and steering angles generated by going around a track once, on the terminal:

```
$ sudo ./UnityAppimage --no-sandbox
```

Open the project in sdsim directory (commit ed0cc0b)

B.12.2 Running simulator predictions

To run predictions, and to monitor TCP traffic:

```
# Run unity from one terminal run:
$ sudo ./Unity.AppImage --no-sandbox
# To monitor TCP traffic, from another terminal run:
$ sudo tcpflow -i lo -c port 9091 > /tmp/tcpflow.log
# To run predictions, from another terminal run:
$ python predict_client.py \
--model=../trained_models/nvidia_baseline/20201120124421\_nvidia\_baseline.h5
```

B.13 ROS

The Robot Operating System (Quigley et al., 2009) is middleware, that is, placed between the operating system an the application program. It helps manage complexity and distributed systems, such as a managing the process of recording several sensor outputs in a moving vehicle. The term "plumbing" is sometimes used, as parts of a distributed application are connection with "data pipes". ROS was used to store data from the FORD AV dataset.

Info on how to get a steering angle (YAW) using quaternions and the IMU data.

B.14 Correspondence with supervisor

B.14.1 Correspondence with Artur Garcez 1

Garcez, Artur
Fri 14/02/2020 09:59
Hi Daniel,
See if you can find a data set which would offer you a systematic way of evaluating how CNNs can be fooled by visual illusions caused by rain drops on the windscreen.
Artur

From: PG-Sikar, Daniel <Daniel.Sikar@city.ac.uk>
Sent: 14 February 2020 08:56
To: Garcez, Artur
Subject: MSC Data Science - Final year project

Hi Artur,

I am currently shopping around for a project. I have spoken to a couple of companies but not sure data will be available in good time - ML applied to signal processing.

So I ask if you have any projects, ideally related to some kind of signal processing, alternatively, anything related to computer vision which could potentially provide a good hook for the self-driving car project we spoke (briefly) about?

Kind regards

Daniel - PT2

B.14.2 Correspondence with Artur Garcez 2

Daniel,

Out of curiosity, is there any easy way of increasing glare from the sun in the simulator?

This would be a nice UK-specific example to give: a situation with light rain and glare possibly creating difficult illusions.

The big picture being there's not much hope training such models for billions of hours in California and transferring to London!

Best,
Artur

B.14.3 Correspondence with Artur Garcez 3

Garcez, Artur
Tue 12/05/2020 15:31
Thank you Daniel.

It is good to see that you've identified a selection of suitable data sets and network models

Suggest you list specific tasks next with their deliverables. It is probably a good idea to s

The above tasks can be as specific as: train AlexNet, edit the data to add reflections, test

Artur

Prof Artur d'Avila Garcez, FBCS
Director - Research Centre for Machine Learning
Chair - The City Data Science Institute
City, University of London
Email: a.garcez@city.ac.uk
URL: http://staff.city.ac.uk/~aag/

B.15 Correspondence with authors

B.15.1 Correspondence with Urs Muller 1

Correspondence with **Urs Muller**, co-author of Bojarski et al., 2016, with respect to network training settings.

From: Urs Muller <umuller@nvidia.com>Sent: 16 November 2020 19:30
To: PG-Sikar, Daniel <Daniel.Sikar@city.ac.uk>
Subject: Re: End to End Learning for Self-Driving Cars - Network Training Parameters

CAUTION: This email originated from outside of the organisation.
Do not click links or open attachments unless you recognise the
sender and believe the content to be safe.

Hi Daniel,

We used the following settings (we haven't documented them in any publication):

loss function: MSE
optimizer: adadelta
learning rate: 1e-4 (but not really used in adadelta)
dropout: 0.25

Best regards,
Urs

From: PG-Sikar, Daniel <Daniel.Sikar@city.ac.uk>
Sent: Sunday, November 15, 2020 6:53To: Urs Muller <umuller@nvidia.com>
Subject: End to End Learning for Self-Driving Cars - Network Training Parameters

External email: Use caution opening links or attachments

Hi,

With respect to your network training, I am investigating the effect of noise
(rainy images) on network performance, using your architecture as a baseline.

Would you be able to point me towards any documentation detailing loss function,
learning rate, optimizer and layer droupout probabilities used when training
your network?

Thanks in advance for any help and kind regards,

Daniel Sikar | MSc Data Science Candidate
School of Mathematics, Computer Science and Engineering
City University of London

B.15.2 Correspondence with Urs Muller 2

Urs Muller <umuller@nvidia.com>
Tue 17/11/2020 11:47
CAUTION: This email originated from outside of the organisation. Do not click links or open a

Hi Daniel,

Yes, correct. We cropped everything above the horizon. The lower edge of the crop is as low a

Best regards,
Urs
From: PG-Sikar, Daniel <Daniel.Sikar@city.ac.uk>
Sent: Tuesday, November 17, 2020 2:56
To: Urs Muller <umuller@nvidia.com>
Subject: Re: End to End Learning for Self-Driving Cars - Network Training Parameters

External email: Use caution opening links or attachments
Hi Urs,

Thanks for your reply. One more question with respect to the image size presented to your net

Kind regards,

Daniel

B.15.3 Correspondence with Ankit Vora

Correspondence with **Ankit Vora**, co-author of Agarwal et al., 2020, with respect to extracting steering angles from rosbag files.

Vora, Ankit (A.) <avora3@ford.com>
Thu 05/11/2020 13:55
CAUTION: This email originated from outside of the organisation.
Do not click links or open attachments unless you recognise the
sender and believe the content to be safe.

Hi Daniel,

The information you are looking for will be in the /pose_ground_truth message.
That message represents the position and orientation of the vehicle.
You will have to convert quaternions to yaw, pitch, roll angles and
then use the yaw values.

Ankit
From: PG-Sikar, Daniel <Daniel.Sikar@city.ac.uk>
Sent: Wednesday, October 28, 2020 7:00 PM
To: Vora, Ankit (A.) <avora3@ford.com>
Subject: Ford AV Dataset - obtaining steering angles

Hi,
I am studying your AV datast and have extracted the /imu data from provided .bag files,
I am trying to work out if the steering angle can be inferred from the data?
Your article states the dataset provides 6DOF pose estimation.
The extracted /imu topic looks like:

```
header:
  seq: 137596
  stamp:
    secs: 1501822140
    nsecs: 610330104
  frame_id: "imu"
orientation:
  x: 0.00133041249526
  y: 0.00486443211508
  z: 0.584795486661
  w: 0.811165091756
orientation_covariance: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
angular_velocity:
  x: -0.0018081133007
  y: -0.0105949952451
  z: -0.00118502619208
angular_velocity_covariance: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
linear_acceleration:
  x: -0.0136899789795
  y: 0.199539378285
  z: 0.305044800043
linear_acceleration_covariance: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

I guess I am interested in the yaw, which would be the value z, but
having looking at the pattern it does not look like an angle that
could be associated to steering, positive and negative values close
to zero.

Could you help me identify the steering angle, or if it is not
present suggest any approaches to extract it from the data?

Anyway help would be greatly appreciated.

Daniel Sikar
Daniel Sikar | MSc Data Science Candidate
School of Mathematics, Computer Science and Engineering
City University of London

B.15.4 Correspondence with Maxime Ellerbach

On the majority of the Donkey Race community using behavioural cloning for self-driving CNN training.

Maxime Ellerbach is the main maintainer of the SDSandbox project.

Maxime Ellerbach

14 Nov 2020, 14:44 (4 days ago)

Oh, I think the drive menu is quite old, it was added before May update,
at this time Tawn was still doing the changes.
Now I'm the main maintainer of the project, for the moment I'm mainly
doing some backend changes to make the track creation process easier.

For the training part, most of us use behavioral cloning !
But they are mostly using the donkeycar framework, on my side I don't
use any framework and it performs great.
I would be happy if you could join us in a next virtual race !

Cheers,
Maxime

-----Message d'origine-----
De : Daniel Sikar <dsikar@gmail.com>
Envoyé : samedi 14 novembre 2020 15:25

À : Maxime Ellerbach <maxime@ellerbach.net>
Objet : Re: SDSandbox output image size

Thanks, that is super helpful. What really helped me was the drive menu being added to the Generated Track.
I have not checked closely the commit history (I have been following SDSandbox since February) but given you have been the most active user I take it was from you.
I did add some data augmentation to training that I thought might be helpful for others doing behavioural cloning, but understand that most (or all) of you in the virtual race league are using reinforcement learning models, making more use out of the Dokey-Gym environment than I am. Anyway, I hope to get to RL before long and to join the league!
Daniel

C Network architectures

C.1 NVIDIA baseline architecture

This is the starting point after the work of Bojarski et al., 2016

```
>>> import models
>>> mymodel = models.nvidia_baseline(1)
>>> models.show_model_summary(mymodel)
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
img_in (InputLayer)          [(None, 66, 200, 3)]      0
_________________________________________________________________
lambda (Lambda)              (None, 66, 200, 3)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 31, 98, 24)        1824
_________________________________________________________________
dropout (Dropout)            (None, 31, 98, 24)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 14, 47, 36)        21636
_________________________________________________________________
dropout_1 (Dropout)          (None, 14, 47, 36)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 5, 22, 48)         43248
_________________________________________________________________
dropout_2 (Dropout)          (None, 5, 22, 48)         0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 3, 20, 64)         27712
_________________________________________________________________
dropout_3 (Dropout)          (None, 3, 20, 64)         0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 1, 18, 64)         36928
_________________________________________________________________
dropout_4 (Dropout)          (None, 1, 18, 64)         0
```

```
_________________________________________________________________
flattened (Flatten)          (None, 1152)              0
_________________________________________________________________
dense_1 (Dense)              (None, 1164)              1342092
_________________________________________________________________
dropout_5 (Dropout)          (None, 1164)              0
_________________________________________________________________
dense_2 (Dense)              (None, 100)               116500
_________________________________________________________________
dropout_6 (Dropout)          (None, 100)               0
_________________________________________________________________
dense_3 (Dense)              (None, 50)                5050
_________________________________________________________________
dropout_7 (Dropout)          (None, 50)                0
_________________________________________________________________
dense_4 (Dense)              (None, 10)                510
_________________________________________________________________
dropout_8 (Dropout)          (None, 10)                0
_________________________________________________________________
steering (Dense)             (None, 1)                 11
=================================================================
Total params: 1,595,511
Trainable params: 1,595,511
Non-trainable params: 0
_________________________________________________________________
[(None, 66, 200, 3)]
(None, 66, 200, 3)
(None, 31, 98, 24)
(None, 31, 98, 24)
(None, 14, 47, 36)
(None, 14, 47, 36)
(None, 5, 22, 48)
(None, 5, 22, 48)
(None, 3, 20, 64)
(None, 3, 20, 64)
(None, 1, 18, 64)
(None, 1, 18, 64)
(None, 1152)
(None, 1164)
(None, 1164)
(None, 100)
(None, 100)
```

```
(None, 50)
(None, 50)
(None, 10)
(None, 10)
(None, 1)
```

C.2 NVIDIA 0201207201157_ nvidia_ baseline.h5

This is the same as nvidia1

```
commit b1af57c
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
img_in (InputLayer)          [(None, 120, 160, 3)]     0
_________________________________________________________________
lambda (Lambda)              (None, 120, 160, 3)       0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 58, 78, 24)        1824
_________________________________________________________________
dropout (Dropout)            (None, 58, 78, 24)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 27, 37, 32)        19232
_________________________________________________________________
dropout_1 (Dropout)          (None, 27, 37, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 12, 17, 64)        51264
_________________________________________________________________
dropout_2 (Dropout)          (None, 12, 17, 64)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 10, 15, 64)        36928
_________________________________________________________________
dropout_3 (Dropout)          (None, 10, 15, 64)        0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 8, 13, 64)         36928
_________________________________________________________________
dropout_4 (Dropout)          (None, 8, 13, 64)         0
_________________________________________________________________
flattened (Flatten)          (None, 6656)              0
_________________________________________________________________
dense_2 (Dense)              (None, 100)               665700
```

```
_________________________________________________________________
dense_3 (Dense)              (None, 50)                5050
_________________________________________________________________
steering (Dense)             (None, 2)                 102
=================================================================
Total params: 817,028
Trainable params: 817,028
Non-trainable params: 0
_________________________________________________________________
[(None, 120, 160, 3)]
(None, 120, 160, 3)
(None, 58, 78, 24)
(None, 58, 78, 24)
(None, 27, 37, 32)
(None, 27, 37, 32)
(None, 12, 17, 64)
(None, 12, 17, 64)
(None, 10, 15, 64)
(None, 10, 15, 64)
(None, 8, 13, 64)
(None, 8, 13, 64)
(None, 6656)
(None, 100)
(None, 50)
(None, 2)
```

C.3 NVIDIA1 architecture

The TawmNet network architecture is as shown, with a total of 817,028 trainable parameters. The design is exactly as used by Kramer, 2020.

```
>>> import models
>>> mymodel = models.nvidia_model1(2)
>>> models.show_model_summary(mymodel)
Model: "model_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
img_in (InputLayer)          [(None, 120, 160, 3)]     0
_________________________________________________________________
lambda_3 (Lambda)            (None, 120, 160, 3)       0
```

```
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 58, 78, 24)        1824
_________________________________________________________________
dropout_15 (Dropout)         (None, 58, 78, 24)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 27, 37, 32)        19232
_________________________________________________________________
dropout_16 (Dropout)         (None, 27, 37, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 12, 17, 64)        51264
_________________________________________________________________
dropout_17 (Dropout)         (None, 12, 17, 64)        0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 10, 15, 64)        36928
_________________________________________________________________
dropout_18 (Dropout)         (None, 10, 15, 64)        0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 8, 13, 64)         36928
_________________________________________________________________
dropout_19 (Dropout)         (None, 8, 13, 64)         0
_________________________________________________________________
flattened (Flatten)          (None, 6656)              0
_________________________________________________________________
dense_6 (Dense)              (None, 100)               665700
_________________________________________________________________
dense_7 (Dense)              (None, 50)                5050
_________________________________________________________________
steering_throttle (Dense)    (None, 2)                 102
=================================================================
Total params: 817,028
Trainable params: 817,028
Non-trainable params: 0
_________________________________________________________________
[(None, 120, 160, 3)]
(None, 120, 160, 3)
(None, 58, 78, 24)
(None, 58, 78, 24)
(None, 27, 37, 32)
(None, 27, 37, 32)
(None, 12, 17, 64)
(None, 12, 17, 64)
(None, 10, 15, 64)
```

```
(None, 10, 15, 64)
(None, 8, 13, 64)
(None, 8, 13, 64)
(None, 6656)
(None, 100)
(None, 50)
(None, 2)
```

C.4 NVIDIA2 Architecture

The modified NaokiNet architecture, with dropout layers after TawnNet, and the second convolutional layer number of feature maps changed from 36 to 32.

```
>>> import models
>>> mymodel = models.nvidia_model2(2)
>>> models.show_model_summary(mymodel)
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
img_in (InputLayer)          [(None, 66, 200, 3)]      0
_________________________________________________________________
lambda (Lambda)              (None, 66, 200, 3)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 31, 98, 24)        1824
_________________________________________________________________
dropout (Dropout)            (None, 31, 98, 24)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 14, 47, 32)        19232
_________________________________________________________________
dropout_1 (Dropout)          (None, 14, 47, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 5, 22, 48)         38448
_________________________________________________________________
dropout_2 (Dropout)          (None, 5, 22, 48)         0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 3, 20, 64)         27712
_________________________________________________________________
dropout_3 (Dropout)          (None, 3, 20, 64)         0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 1, 18, 64)         36928
_________________________________________________________________
```

```
dropout_4 (Dropout)          (None, 1, 18, 64)         0
_________________________________________________________________
flattened (Flatten)          (None, 1152)              0
_________________________________________________________________
dense (Dense)                (None, 100)               115300
_________________________________________________________________
dense_1 (Dense)              (None, 50)                5050
_________________________________________________________________
dense_2 (Dense)              (None, 10)                510
_________________________________________________________________
steering_throttle (Dense)    (None, 2)                 22
=================================================================
Total params: 245,026
Trainable params: 245,026
Non-trainable params: 0
_________________________________________________________________
[(None, 66, 200, 3)]
(None, 66, 200, 3)
(None, 31, 98, 24)
(None, 31, 98, 24)
(None, 14, 47, 32)
(None, 14, 47, 32)
(None, 5, 22, 48)
(None, 5, 22, 48)
(None, 3, 20, 64)
(None, 3, 20, 64)
(None, 1, 18, 64)
(None, 1, 18, 64)
(None, 1152)
(None, 100)
(None, 50)
(None, 10)
(None, 2)
```

C.5 Alexnet

Adapted to achieve same geometry - NB Input size (described as 224x224) is suggested to be 227x227 "CS231n: Convolutional Neural Networks for Visual Recognition 2016". Though changing kernel size to 8x8 also generates a 55x55 feature map in the first convolutional layer.

```
models.show_model_summary(model)
```

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
img_in (InputLayer)          [(None, 224, 200, 3)]     0
_________________________________________________________________
lambda (Lambda)              (None, 224, 200, 3)       0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 55, 49, 48)        9264
_________________________________________________________________
maxpool2d_1 (MaxPooling2D)   (None, 55, 49, 48)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 27, 24, 128)       55424
_________________________________________________________________
maxpool2d_2 (MaxPooling2D)   (None, 27, 24, 128)       0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 13, 11, 192)       221376
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 13, 11, 192)       331968
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 13, 11, 128)       221312
_________________________________________________________________
maxpool2d_3 (MaxPooling2D)   (None, 13, 11, 128)       0
_________________________________________________________________
dropout (Dropout)            (None, 13, 11, 128)       0
_________________________________________________________________
flattened (Flatten)          (None, 18304)             0
_________________________________________________________________
Dense_1 (Dense)              (None, 2048)              37488640
_________________________________________________________________
Dense_2 (Dense)              (None, 50)                102450
_________________________________________________________________
dense (Dense)                (None, 10)                510
_________________________________________________________________
steering_throttle (Dense)    (None, 2)                 22
=================================================================
Total params: 38,430,966
Trainable params: 38,430,966
Non-trainable params: 0
_________________________________________________________________
[(None, 224, 200, 3)]
(None, 224, 200, 3)
```

```
(None, 55, 49, 48)
(None, 55, 49, 48)
(None, 27, 24, 128)
(None, 27, 24, 128)
(None, 13, 11, 192)
(None, 13, 11, 192)
(None, 13, 11, 128)
(None, 13, 11, 128)
(None, 13, 11, 128)
(None, 18304)
(None, 2048)
(None, 50)
(None, 10)
(None, 2)
```

C.6 Training and testing networks

A network is trained by running the script:

```
$ python --model=nvidia1
    --outdir=../trained_models
    --epochs=1
    --inputs=../dataset/unity/roboleague/log/*.jpg
    --aug=false
    --crop=false
```

This will generate a model, saved in the .h5 format, a python dictionary object with training and testing accuracy and loss values, and an accuracy and loss plot with information additional labels to help identify the trained model. Finally, a log file is saved to disk containing model name, training time and last recorded loss and accuracy values for training and testing datasets.

run experiment on vanilla code (no augmentation) Record cash crash and deviation from actual to predicted steering angles.

D Deliverables

The tables presented next are motivated by the discussion held with the project supervisor (B.14.3). 92 deliverables were identified for this research. Table D.1 refers to datasets that will be used to train our models. Table D.2 refers to the simulator tracks used to test models. Table D.3 refers to the models (i.e. network architectures) to be trained with our datasets and tested on our simulator tracks, finally tables D.4 through to D.8 refer to training tasks performed with a permutation of models and datasets, and metrics obtained by testing each model on each simulator track.

Deliverables - Datasets			
ID	Task	Deliverable	Description
1	Download	D1	Udacity real world dataset
2	Generate	D2	Unity3D simulator data
3	Combine	D3	Udacity real and Unity3D simulator data
4	Gather	D4	Mechanical Turk dry/rainy FordAV dataset

TABLE D.1: Datasets used to train models

Deliverables - Simulator testing tracks			
ID	Task	Deliverable	Description
5	Build	S1	Unity simulator sunny/dry
6	Build	S2	Unity simulator added rain/reflections 1
7	Build	S3	Unity simulator added rain/reflections 2

TABLE D.2: Simulator tracks to test models

Deliverables - Models			
ID	Task	Deliverable	Description
8	Build	N1	NVIDIA end-to-end architecture
9	Build	N2	AlexNet
10	Build	N3	VGGNet
11	Build	N4	InceptionNet
12	Build	N5	ResNet

TABLE D.3: Models to be trained and tested

Deliverables - NVIDIA					
ID	Task	Network/Model	Data/Sim	Deliverable	Description
13	Train	N1	D1	N1M1	Model
14	Test	N1M1	S1	N1R1	Metric
15	Test	N1M1	S2	N1R2	Metric
16	Test	N1M1	S3	N1R3	Metric
17	Train	N1	D2	N1M2	Model
18	Test	N1M2	S1	N1R3	Metric
19	Test	N1M2	S2	N1R4	Metric
20	Test	N1M2	S3	N1R5	Metric
21	Train	N1	D3	N1M3	Model
22	Test	N1M3	S1	N1R7	Metric
23	Test	N1M3	S2	N1R8	Metric
24	Test	N1M3	S3	N1R9	Metric
25	Train	N1	D4	N1M4	Model
26	Test	N1M4	S1	N1R10	Metric
27	Test	N1M4	S2	N1R11	Metric
28	Test	N1M4	S3	N1R12	Metric

TABLE D.4: NVIDIA model and metric deliverables

Deliverables - AlexNet					
ID	Task	Network/Model	Data/Sim	Deliverable	Description
29	Train	N2	D1	N2M1	Model
30	Test	N2M1	S1	N2R1	Metric
31	Test	N2M1	S2	N2R2	Metric
32	Test	N2M1	S3	N2R3	Metric
33	Train	N2	D2	N2M2	Model
34	Test	N2M2	S1	N2R3	Metric
35	Test	N2M2	S2	N2R4	Metric
36	Test	N2M2	S3	N2R5	Metric
37	Train	N2	D3	N2M3	Model
38	Test	N2M3	S1	N2R7	Metric
39	Test	N2M3	S2	N2R8	Metric
40	Test	N2M3	S3	N2R9	Metric
41	Train	N2	D4	N2M4	Model
42	Test	N2M4	S1	N2R10	Metric
43	Test	N2M4	S2	N2R11	Metric
44	Test	N2M4	S3	N2R12	Metric

TABLE D.5: AlexNet model and metric deliverables

Deliverables - VGGNet					
ID	Task	Network/Model	Data/Sim	Deliverable	Description
45	Train	N3	D1	N3M1	Model
46	Test	N3M1	S1	N3R1	Metric
47	Test	N3M1	S2	N3R2	Metric
48	Test	N3M1	S3	N3R3	Metric
49	Train	N3	D2	N3M2	Model
50	Test	N3M2	S1	N3R3	Metric
51	Test	N3M2	S2	N3R4	Metric
52	Test	N3M2	S3	N3R5	Metric
53	Train	N3	D3	N3M3	Model
54	Test	N3M3	S1	N3R7	Metric
55	Test	N3M3	S2	N3R8	Metric
56	Test	N3M3	S3	N3R9	Metric
57	Train	N3	D4	N3M4	Model
58	Test	N3M4	S1	N3R10	Metric
59	Test	N3M4	S2	N3R11	Metric
60	Test	N3M4	S3	N3R12	Metric

TABLE D.6: VGGNet model and metric deliverables

Deliverables - InceptionNet					
ID	Task	Network/Model	Data/Sim	Deliverable	Description
61	Train	N4	D1	N4M1	Model
62	Test	N4M1	S1	N4R1	Metric
63	Test	N4M1	S2	N4R2	Metric
64	Test	N4M1	S3	N4R3	Metric
65	Train	N4	D2	N4M2	Model
66	Test	N4M2	S1	N4R3	Metric
67	Test	N4M2	S2	N4R4	Metric
68	Test	N4M2	S3	N4R5	Metric
69	Train	N4	D3	N4M3	Model
70	Test	N4M3	S1	N4R7	Metric
71	Test	N4M3	S2	N4R8	Metric
72	Test	N4M3	S3	N4R9	Metric
73	Train	N4	D4	N4M4	Model
74	Test	N4M4	S1	N4R10	Metric
75	Test	N4M4	S2	N4R11	Metric
76	Test	N4M4	S3	N4R12	Metric

TABLE D.7: InceptionNet model and metric deliverables

Deliverables - ResNet					
ID	Task	Network/Model	Data/Sim	Deliverable	Description
77	Train	N5	D1	N5M1	Model
78	Test	N5M1	S1	N5R1	Metric
79	Test	N5M1	S2	N5R2	Metric
80	Test	N5M1	S3	N5R3	Metric
81	Train	N5	D2	N5M2	Model
82	Test	N5M2	S1	N5R3	Metric
83	Test	N5M2	S2	N5R4	Metric
84	Test	N5M2	S3	N5R5	Metric
85	Train	N5	D3	N5M3	Model
86	Test	N5M3	S1	N5R7	Metric
87	Test	N5M3	S2	N5R8	Metric
88	Test	N5M3	S3	N5R9	Metric
89	Train	N5	D4	N5M4	Model
90	Test	N5M4	S1	N5R10	Metric
91	Test	N5M4	S2	N5R11	Metric
92	Test	N5M4	S3	N5R12	Metric

TABLE D.8: ResNet model and metric deliverables

E Results

This appendix contains notes related to obtaining results that were and were not used in the final dissertation. Results that were not used are left for completeness, as they may be used for future work.

E.1 Ford AV Dataset steering angles

E.1.1 Ford AV Dataset

The steering angles can be extracted from .bag files using ROS commands:

```
    # In one terminal, start ros engine
    $ roscore
    # In another terminal, inspect content of bag file
    $ time rosbag info Sample-Data.bag
    (...)
             /imu                 146939 msgs    : sensor_msgs/Imu
    (...)
             /pose_ground_truth   146136 msgs    : geometry_msgs/PoseStamped
             /pose_localized       16100 msgs    : geometry_msgs/PoseStamped
             /pose_raw            146190 msgs    : geometry_msgs/PoseStamped
(...)
    # And subscribe to topic of interest
    $ rostopic echo /imu | tee sample_imu.yaml
    # In another terminal, playback bag file
    $ time rosbag play --immediate Sample-Data.bag --topics /imu
    # Sanity check, count number of acquisitions
    $ cat sample_imu.yaml | grep "orientation:" | wc -l
```

The snippet above generates file imu.yaml, with all pose data generated by imu device. From this file we extract the steering angle, which is the z axis (yaw) of the orientation field (TODO check .yaml dialect). Images can be extracted from the same bag file with the Python 2.7 bag_to_images.py script:

```
    $ python2 bag_to_images.py Sample-Data.bag ~/git/msc-data/ford/sample/ros/ \
        /image_front_left
```

Each image is an attribute in a dictionary, which also contains seconds (secs) and nano seconds (nsecs) attributes within the header attribute:

```
header:
  seq: 213414
  stamp:
    secs: 1501822147
    nsecs: 684951066
  frame_id: "camera_front_left"
height: 215
width: 414
encoding: "8UC3"
is_bigendian: 0
(...)
```

Thus a timestamp can be obtained for each image extracted. This is done with script

parse_yaml_time.py:

```
$ python ./ford/sample/parse_ford_yaml_time.py
header -> {'seq': 213414, 'stamp': {'secs': 1501822147, 'nsecs': 684951066}, 'frame_id':
'camera_front_left'}
1501822147,684951066
header -> {'seq': 213415, 'stamp': {'secs': 1501822147, 'nsecs': 820741891}, 'frame_id':
'camera_front_left'}
1501822147,820741891
(...)
```

While the steering angles are extracted (TODO ADD QUATERNION CORRESPONDENCE) The image can be matched with a steering angle by obtaining the timestamp of image, the full secs

E.2 Audi

The Audi Autonomous Driving Dataset (A2D2) authors (Geyer et al., 2020) are motivated by the fact that research in machine learning, mobile robotics and autonomous driving is accelerated by the availability of high quality annotated data. This statement can be verified by the advances in image classification with deep neural networks since **IMAGENET** became available. The A2D2 data was acquired with a human-driven Audi Q7 e-tron equipped with six cameras (front left, front center, front right, back left, back center, back right) and five LiDAR sensors. The authors claim this resulted in 360° camera and LiDAR coverage. Additionally, several bus data signals from the vehicle were recorded such as velocity, acceleration and steering wheel angle. The total size is 2.3TB. Our datasets of interest are the Though, since we are interested in image and steering angle only, our data can be narrowed to

```
https://aev-autonomous-driving-dataset.s3.eu-central-1.amazonaws.com/
README-SensorFusion.txt

- steering_angle_calculated
- steering_angle_calculated_sign

- 'cam_front_center'
```

The dataset total size is approximately 2.3TB, although the majority is sensor fusion data note used in this study. The data can be dwonloaded from

```
https://www.a2d2.audi/a2d2/en/download.html
```

This was supplied for three cities: Gaimersheim, Ingolstadt and Munich. To perform initial investigations we chose data from Munich and downloaded the "Camera - Front Center" images, constituting 27451 images 3.2MB in size each and dimension 1920x1208 pixels, total size on disk is about 92GB. We also downloaded the 176MB "Bus Signals" file. The image naming convention uses a timestamp in the format:

```
20190401145936_camera_frontcenter_000017970.png
```

The "Bus Signals" file is JSON encoded and provides several signals such as acceleration, angular velocity and vehicle speed. Our signals of interest are the steering angle calculated and steering angle sign. In the bus signals file (20190401121727_bus_signals.json) we parsed our values of interest e.g.

```
(...)
    "steering_angle_calculated": {
        "unit": "Unit_DegreOfArc",
        "values": [
            [
                1554115464698116,
                2.4
            ],
(...)
```

where the unit is degree of arc and the values are inferred to be a timestamp when the measurement was acquired and the angle (1554115464698116 and 2.4 respectively, in the example shown). and found 91968 entries for each of the steering angle and sign. This is over 3 times the amount of corresponding images. Since there was no obvious key to match the steering angle and sign with a corresponding image, we wrote to the supplied enquiry email address aevdrivingdataset@audi.de with regard to this problem and receiving no reply, abandoned the dataset deeming it unusable for our purposes. Note: we did try converting the integer into a date using python, which resulted in an error:

```
import datetime
audi_timestamp = 1554115464698116
date = datetime.datetime.fromtimestamp(audi_timestamp / 1e3)
print(date)
# ValueError: year 51217 is out of range
```

(TODO ADD CORRESPONDENCE WITH AUTHOR)

E.3 Unity3D changing sky hue

The simulator is started by running:

```
$ sudo ~/.Unity.AppImage --no-sandbag
```

This will load the Unity Hub application. A project can be added, which will be repository cloned from Kramer, 2020. Once loaded, the menu scene is selected and the project is run. Once running, an output directory must be chosen. A track is then selected (TODO track list), then the option "Auto with REC TODO double check". TODO ADD IMAGE SEQUENCE Once a number of laps have been completed, the images can be moved to a labelled folder using prepare_data.py script. This will move images to a user defined directory, create a sub-directory named with a date and timestamp

The sky colour can be changed via Window > Renderering > Lighting settings, then under Environment changing "Skybox Material". Default Skybox Material is "Default-Skybox". Suggested for darker background is "Usa_Number_M" - blue.

Changing camera sensor image output size

There are two ways to change the size of images output by simulator camera sensor, one is by editing Donkey.Prefab file and changing lines 3415 and 3416:

```
  width: 160
  height: 120
```

The other was is through Unity Changing sky colour The sky colour can be changed via Windows > Renderer > Lighting Settings menu, then under Environment changing "Skybox Material" chossing a different material. Suggested for darker background is "Usa_Number_M". The simulator lighting can be made darker, in the same menu, under Environment Lighting > Intensity MUltiplier. A value of 1 is chosen to generate training datasets, and a value of 0.26 when running the simulator in NN Control over Network mode, that is using the prediction engine to predict steering angles.

E.4 Udacity

1. Link trail - medium (Indian guy) -> medium (Chinese guy)-> github (Japanese guy)

2. Udacity data Data is available for download in torrent file format (Wikipedia contributors, 2020) and consist of Robot Operating System (ROS) rosbag compressed files.

`https://github.com/udacity/self-driving-car/tree/master/datasets`

3. Rosbag

E.5 Training and Testing Log

The first two entries are incomplete, have been commented out and can be seen in the latex AppendixD-results.tex file.

E.5.1 Run 3

This model has a single output, steering angle, and produced very low accuracy. Notes:

```
commit 423b5b783565b60e72f970485d9b3aa9887f5453
training time
dataset: sample_data
```

E.5.2 Run 4 -

Also not doing well Next run, 2 outputs (steering and throttle)

E.5.3 Run 5 - 20201102081239

```
commit 076e8b32664738df6af9e14d75355504eb2a94b4
Much better results with two outputs.
cat ../dataset/unity/log_sample/logs_Mon_Jul_13_08_29_01_2020/record_11659.json
Both are floats - steering angle and throttle
loss: 0.0105 - acc: 0.8502 - val_loss: 0.0111 - val_acc: 0.8617

dataset: sample_data
model: nvidia1
outputs: 2

Comment: Network trained with no augmentation of cropping.
```

From this point onwards, a model name, if generated, is given with every run.

E.5.4 Run 5 - 20201102090041_nvidia2

```
commit 42dabb6321ad25f667c8663b63412c88c96b3b38
model: nvidia2
outputs: 2
dataset: log_sample (size: 12k)
$ python train.py --model=nvidia2 --outdir=../trained_models
loss: 0.0108 - acc: 0.8483 - val_loss: 0.0117 - val_acc: 0.8495
```

E.5.5 Run 6 - 20201102094552_nvidia1

```
commit ec9d081b85f7386365428a73896b1d09be7ba917
model: nvidia1
outputs: 2
dataset: genRoad (280727)
command
$ train.py --model=nvidia1 --outdir=../trained_models
loss: 0.0077 - acc: 0.8726 - val_loss: 0.0077 - val_acc: 0.8732

Comment: Network trained with no augmentation or cropping.
Drove off the road.
```

video: `https://youtu.be/ZLhrcyuONj0`

Figure E.1 shows the training loss and accuracy plot for 20201102094552_ nvidia1.h5.

E.5.6 Run 7 - 20201102090041_nvidia2.h5

```
commit 1da10b6745f583e180d5b9c5ba4874847ba8610c
model: nvidia2
outputs: 2
Dataset: genRoad
command
$ train.py --model=nvidia2 --outdir=../trained_models
```

E.5.7 Run 8 - 20201102134802_nvidia2.h5

```
commit 6960f2f5fb50b565c0dfd6c8fe3ac1d283192e69
model nvidia2
outputs 2
dataset log2
command:
```

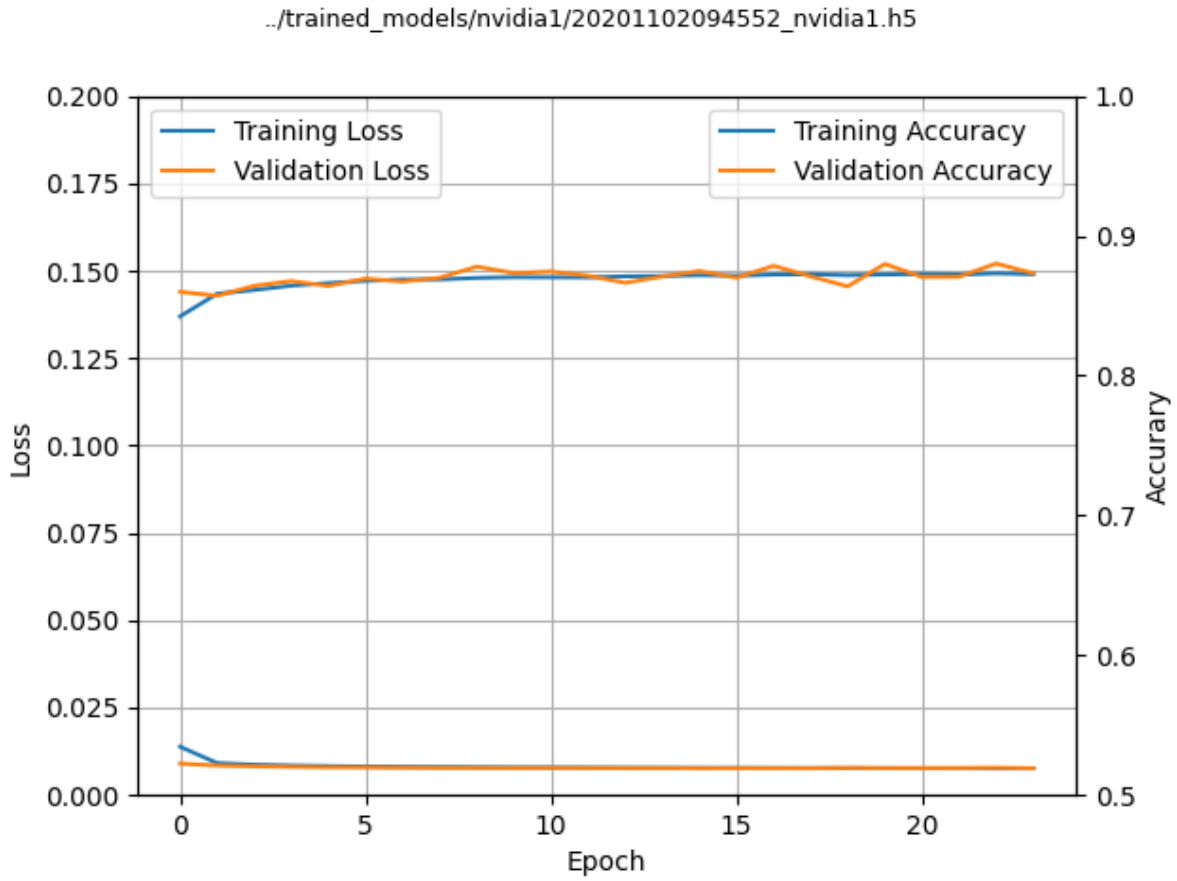

FIGURE E.1: 20201102094552_ nvidia1.h5 testing and validation loss and accuracy plots (run 6)

```
train.py --model=nvidia2 --outdir=../trained_models
```

Figure E.2 shows the training loss and accuracy plot for 20201102134802_ nvidia2.h5.

E.5.8 Run 9 - 20201102210514_nvidia2.h5

```
Running nvidia2 with augmentation
commit 6960f2f5fb50b565c0dfd6c8fe3ac1d283192e69
model nvidia2
outputs 2
dataset log2
command:
train.py --model=nvidia2 --outdir=../trained_models
loss: 0.0345 - acc: 0.7906 - val_loss: 0.0236 - val_acc: 0.8084
Stop! Still running!!! Epoch 60 and still improving. ...
% loss: 0.0193 - acc: 0.8250 - val_loss: 0.0116 - val_acc: 0.8503
% 0.0123 - acc: 0.8501 - val_loss: 0.0091 - val_acc: 0.8594
Ran 78 epochs and may have thrown error:
```

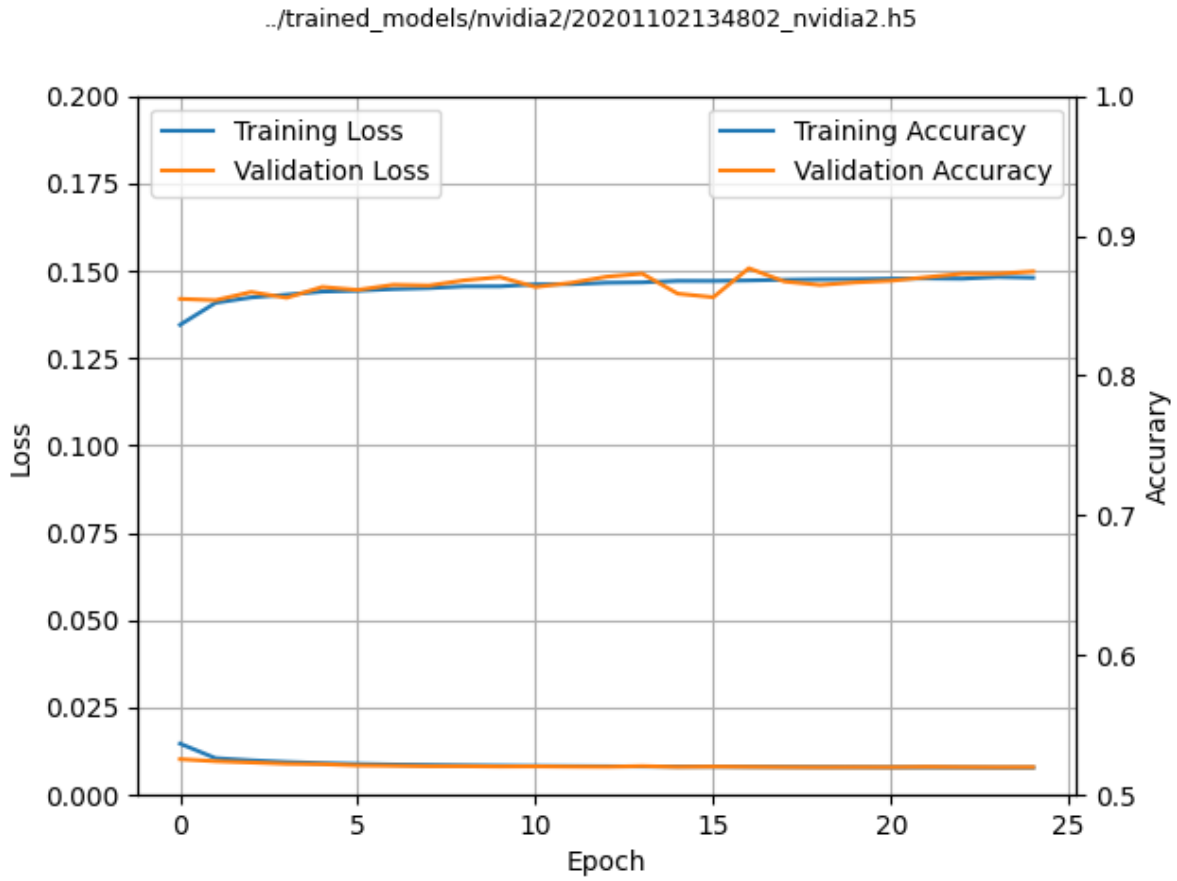

FIGURE E.2: 20201102134802_ nvidia2.h5 testing and validation loss and accuracy plots. Run 8

```
problems with loss graph
Note biggest increase in accuracy during training
```

E.5.9 Run 10 - 20201103211330_nvidia2.h5

```
Run nvidia2 WITHOUT augmentation, just preprocessing
commit 2add77bb60505fe25075f9da55f6465e65cd3825
model nvidia 2
outputs 2
dataset log2
command:
train.py --model=nvidia2 --outdir=../trained_models
loss: 0.0233 - acc: 0.8179 - val_loss: 0.0170 - val_acc: 0.8304
loss: 0.0084 - acc: 0.8656 - val_loss: 0.0086 - val_acc: 0.8655
Epoch 45/100
loss: 0.0083 - acc: 0.8662 - val_loss: 0.0086 - val_acc: 0.8638
problems with loss graph
```

No loss graph, again, no augmentation trained quicker that previous

E.5.10 Run 11

commit 6439a8758d8b46a1cbc3bcefc9db4c15f70820df
model nvidia1
outputs 2
dataset log2
command:
train.py --model=nvidia1 --outdir=../trained_models
Epoch 1/100
1755/1755 [==============================] - 682s 389ms/step - loss: 0.0271
- acc: 0.8049 - val_loss: 0.0172 - val_acc: 0.8351
Epoch 84/100
1755/1755 [==============================] - 559s 318ms/step - loss: 0.0099
- acc: 0.8581 - val_loss: 0.0083 - val_acc: 0.8662
problems with loss graph (probably a bug introduced in adding info to graph?)

E.5.11 Run 12

commit augment.ipynb
model nvidia_baseline
outputs 1
dataset log2 (280746)
command:
train.py --model=nvidia_baseline --outdir=../trained_models

There is an issue with the aspect ratio and cropping, need to investigate with augment.ipynb

Note unity dataset log2 was renamed genRoad as it is obtained from a number of randomly generated Generated Road circuits.

E.5.12 Run 13

commit e14e4bb9bdd0d322867c6cdba706478f662e71f6
model nvidia_baseline
outputs 1
dataset log (45421)cd

```
command:
train.py --model=nvidia_baseline
--outdir=../trained_models
--epochs=1
--inputs=../dataset/unity/log/*.jpg
--aug=True
--preproc=True

Comment: ran ok for one epoch, images sized correctly.
Adding more epochs
```

E.5.13 Run 14

```
commit e14e4bb9bdd0d322867c6cdba706478f662e71f6
model nvidia_baseline
outputs 1
dataset log
command:
train.py --model=nvidia_baseline
--outdir=../trained_models
--epochs=100
--inputs=../dataset/unity/log/*.jpg
--aug=True
--preproc=True

Comment: Started with very low accuracy.
Stopped process as error seems to have gone out of range
Epoch 5/100
283/283 [==============================]
- 107s 377ms/step - loss: nan - acc: 5.5259e-05
- val_loss: nan - val_acc: 1.1004e-04
```

E.5.14 Run 15

```
commit aec3290fdbc83e71e450deef650aa2d51873b886
model nvidia_baseline
outputs 2
dataset log
command:
train.py --model=nvidia_baseline
--outdir=../trained_models
--epochs=100
```

```
--inputs=../dataset/unity/log/*.jpg
--aug=True
--preproc=True
Comment: Started at loss: nan - acc: 0.4305. Two outputs definitely help. Why?
Perhaps because model is not going around track at constant speed?
```

E.5.15 Run 16

```
commit 10a3fc2f6d23e04f604914c1f8420e574a8ce808
model nvidia_baseline
outputs 2
dataset log
command:
train.py --model=nvidia_baseline
--outdir=../trained_models
--epochs=100
--inputs=../dataset/unity/log/*.jpg
--aug=True
--preproc=True
Comment: Added linear activation in final layer, loss returning a reasonable value:
loss: 1.2035 - acc: 0.5404
Turned into nan on 3rd epoch. Accuracy going down, stopping process
```

E.5.16 Run 17

```
Commit: a76169106a9087f5cc7e851fe09294699bb6240c
Model: nvidia_baseline
Outputs: 2
Dataset: genRoad
Command:
train.py --model=nvidia_baseline
--outdir=../trained_models
--epochs=100
--inputs=../dataset/unity/log/*.jpg
--aug=True
--preproc=True

Comment: Removed weight decay
Still loss taking NaN values
Epoch 13/100
281/281 (...) val_loss: nan - val_acc: 0.4939
```

E.5.17 Run 18

```
Commit: baa8f3d066dc37d3c2bb7792fa9db4801824d1bb
Model: nvidia_baseline
Outputs: 2
Dataset: genRoad
Command:
train.py --model=nvidia_baseline
--outdir=../trained_models
--epochs=100
--inputs=../dataset/unity/log/*.jpg
--aug=True
--preproc=True
Comment: Removed dropout from last dense layer. Training loss NaN on first epoch.
Dropout added again, loss back to under 1. Seems to have an influence, trying
multiple dropout removals next.
```

E.5.18 Run 19

```
Commit: 16a6f8ffc6399143e7ad5a6885ee9b44b3ca1dda
Model: nvidia_baseline
Outputs: 2
Dataset: genRoad
Command:
train.py --model=nvidia_baseline
--outdir=../trained_models
--epochs=100
--inputs=../dataset/unity/log/*.jpg
--aug=True
--preproc=True
Comment: Left one dropout (.25) layer, loss still NaN.
```

E.5.19 Run 20

```
Commit: d7e05ad1cdf0fab3a83975be259feb16830b5a38
Rest same as 19
Comment: Removed Dense(1164) layer. Loss is Nan on first epoch.
Stopping run
```

E.5.20 Run 21

```
Commit: d8fb9705dd676554dcf84a213b3c27d0e1f9d0c4
Rest same as 19
Comment: Loss is Nan on first epoch
Epoch 1/100
(...)loss: nan - acc: 0.5017
```

E.5.21 Run 22

```
Commit: ba4432568a589eac5a7d7c4927fa96e2f9e11bd1
Model, Outputs, Dataset and Command: Same as 19
Comment: Using Glorot Uniform (Xavier) kernel initializer.
Loss in Nan on first epoch. Training stopped.
Epoch 1/100
284/284 [==============================] - 101s 357ms/step - loss: nan
- acc: 0.4609
```

E.5.22 Run 23

```
Commit: 5cba2c7b01a160e7053e09220e63b1c575cf51e8
Model, Outputs, Dataset and Command: Same as 19
Comment: All biases initialized to 0.
Loss in Nan on first epoch. Training stopped.
Epoch 1/100
283/283 (...) loss: nan
```

E.5.23 Run 24

```
Commit: 56aea3e16bb0f9db5735dfa536809389f35b12da
Model, Outputs, Dataset and Command: Same as 19
Comment: Removed Dense(10) layer
Loss in Nan on first epoch. Training stopped.
```

E.5.24 Run 25

```
Commit: 6a8ee72727db2508d2ff1ae35d068837a5b524ab
Model, Outputs, Dataset and Command: Same as 19
Comment: Increased dropout to 0.5.
Loss in Nan on second epoch. Training stopped.
Epoch 1/100
285/285 [==============================] - 100s 352ms/step
```

```
- loss: 0.0660 - acc: 0.6910 - val_loss: 0.0299 - val_acc: 0.7809
Epoch 2/100
285/285 [==============================] - 100s 350ms/step - loss: nan
- acc: 0.5000
```

E.5.25 Run 26

```
Commit: e4a287dc816a2a4d2c47893b131c710b2ecb8594
Model, Outputs, Dataset and Command: Same as 19
Comment: Spreading 0.5 dropout between layers (0.1 each).
Loss in Nan on third epoch. Training stopped.
Epoch 3/100
284/284 (...) loss: nan
```

E.5.26 Run 26

```
Commit: 2e5bf1b7002a2ddbbe2fea003fce010627d723e2
Model, Outputs, Dataset and Command: Same as 19
Comment: Changed layer dropout to 0.15.
Loss in Nan on first epoch. Training stopped.
```

E.5.27 Run 27

```
Commit:
Model, Outputs, Dataset and Command: Same as 19
Comment: Changed layer dropout to 0.05.
Loss in Nan on first epoch. Training stopped.
```

E.5.28 Run 28

```
Commit: 9a70ff7e194736a475bdce0b6eddc65aad6ea8c0
Model, Outputs, Dataset and Command: Same as 19
Comment: Changed number of kernels on 2nd Conv layer to 32.
Loss in Nan on first epoch. Training stopped.
```

E.5.29 Run 29

```
Commit:
Model, Outputs, Dataset and Command: Same as 19
Comment: Changed number of kernels on 3nd Conv layer to 64.
Loss in Nan on first epoch. Training stopped.
```

E.5.30 Run 30

```
Commit: 1997b914e829466659a27bfe1b31a6a6374afd36
Model, Outputs, Dataset and Command: Same as 19
Comment: Changed aspect ratio to 160x120
Loss in Nan on 2nd epoch. Training stopped.
Epoch 2/100
283/283 (...) loss: nan - acc: 0.5982
```

E.5.31 Run 31

```
Commit: 1997b914e829466659a27bfe1b31a6a6374afd36
Model, Outputs, Dataset and Command: Same as 19
Comment: Changed aspect ratio to 160x120
Loss in Nan on 2nd epoch. Training stopped.
Epoch 2/100
283/283 (...) loss: nan - acc: 0.5982
```

E.5.32 Run 32 - 20201117154210_nvidia_baseline.h5

```
Commit: 1997b914e829466659a27bfe1b31a6a6374afd36
Model: nvidia_baseline
Outputs: 2
Dataset: smallLoopingCourse (jungle1 renamed) (34443)
Command:
train.py --model=nvidia_baseline
--outdir=../trained_models
--epochs=100
--inputs=../dataset/unity/genTrack/log/*.jpg
--aug=True
--preproc=True
Comment: Loss in Nan on 1st epoch. Training stopped.
This run also produced a usable model.
```

E.5.33 Run 33

```
Commit: 861095ac1a997e0f460aca6ceda93df98b7d0a48
Model, Outputs, Dataset and Command: Same as 19
Comment: Explicitly setting strides=(1,1) is conv layers 4 and 5.
Loss in Nan on 1st epoch. Training stopped.
```

E.5.34 Run 34 - 20201117162326_nvidia_baseline.h5

```
Commit: d0b7160793ee433ed25724d966a7d3bd85ae8ffa
Model, Outputs, Dataset and Command: Same as 19
Comment: Changed batch size to 64
Loss is NaN on 7th epoch
Epoch 7/100
567/567 (...) loss: nan

This run produced a usable model.
```

This run consisted of a sanity check to determine if training would generate a loss value different from NaN (not a number). This was not the case as per results in comments.

E.5.35 Run 35 - 20201120124421_nvidia_baseline.h5

```
Commit: 1f5c64bc219b31cf8654d2cceec910f0dea5ecbf
Model: nvidia_baseline
Outputs: 2
Dataset: log_sample (12894 - small looping track)
Environment: simbox (local)
Command:
--model=nvidia_baseline --outdir=../trained_models --epochs=100
--inputs=../dataset/unity/log_sample/*.jpg --aug=True --preproc=True

Comment: Sanity check run, with the aim of
establishing a repeatable training session, aiming
at debbugging Keras when the need arises, that is
when Keras is suspected of somehow having become
corrupted.

Note before running, all .jpg images were found to have corresponding
steering angles:

$ python ~/git/sdsandbox/src/utils/jsonclean.py \
--inputs=/home/simbox/git/sdsandbox/dataset/unity/log_sample/*.jpg
Files deleted: 0

From the log file 20201120124421\_nvidia\_baseline.log:

Total training time: 0:03:09
Training loss: nan
```

```
Validation loss: nan
Training accuracy: 0.418
Validation accuracy: 0.480

Loss was NaN from 2nd training epoch:
Epoch 2/100
160/160 (...)- loss: nan

When running the model:

sudo ./Unity.AppImage --no-sandbox
And logging the session:
sudo tcpflow -i lo -c port 9091 > /tmp/tcpflow.log
and running predictions
python predict_client.py
--model=../trained_models/nvidia_baseline/20201120124421\_nvidia\_baseline.h5

The model was found to crash into a bollard. A video was generated using
src/utils.MakeVideo.py

https://youtu.be/ZwP53IhBO20
```

Given previous results, this run reverts code to commit 436f635f

E.5.36 Run 36 - 20201120171015_ sanity.h5

Forensics are conducted in this run to find out what code generated model 20201107210627_nvidia1.h5, determining that outliers introduced by a new dataset added to genRoad create problems. The outliers (present in logs_ Thu_ Jul_ _ 9_ 16_ 00_ 15_ 2020) have since been quarantined.

```
Commit: 436f635fa183740832270ea7db6869918a786d55
Model: nvidia1
Outputs: 2
Dataset: log_sample (12894) - "small looping circuit" this is similar to the Generated Track
The difference being the presence of trees, mountains and tall grass.
Command: python train.py --model=sanity --outdir=../trained\_models
Environment: simbox
Comment: This model was trained with the committed code from November 7th, which
produced the first successfull lap drive with not crashes or steering off the track.
There is no record of what commit generated the model, though on video:
https://youtu.be/9z0mMtOnUUc
```

```
One section in the video shows:
python predict_client.py \
--model=../trained_models/nvidia1/20201107210627\_nvidia1.h5
The model name contains date and time created.

The sanity check model trained in under five minutes, and the best model saved
was most likely (subject to double checking Keras source code) generated on the
third epoch.

cat 20201120171015_sanity.log
Model name: ../trained_models/sanity/20201120171015_sanity.h5
Total training time: 0:04:38 (...)

A record of the training output is displayed below:

Epoch 1/100
80/80 [==============================] - 35s 432ms/step - loss: 0.0822
- acc: 0.6293 - val_loss: 0.0300 - val_acc: 0.8109
Epoch 2/100
80/80 [==============================] - 34s 425ms/step - loss: 0.0650
- acc: 0.7127 - val_loss: 0.0264 - val_acc: 0.8129
Epoch 3/100
80/80 [==============================] - 34s 423ms/step - loss: 0.0539
- acc: 0.7429 - val_loss: 0.0276 - val_acc: 0.8133
Epoch 4/100
80/80 [==============================] - 34s 424ms/step - loss: nan
- acc: 0.5398 - val_loss: nan - val_acc: 0.4930
Epoch 5/100
80/80 [==============================] - 34s 427ms/step - loss: nan
- acc: 0.4228 - val_loss: nan - val_acc: 0.4910
Epoch 6/100
80/80 [==============================] - 34s 428ms/step - loss: nan
- acc: 0.4245 - val_loss: nan - val_acc: 0.4963
Epoch 7/100
80/80 [==============================] - 34s 431ms/step - loss: nan
- acc: 0.4216 - val_loss: nan - val_acc: 0.4934
Epoch 8/100
80/80 [==============================] - 34s 428ms/step - loss: nan
- acc: 0.4285 - val_loss: nan - val_acc: 0.4922

Further forensic examination of commits:
```

```
Trying to find what data and code generated model 20201107210627_nvidia1.h5
This was generated on November 7th. There are 4 commits for that day:

src(master)$ git log > gitlog.txt
src(master)$ vim gitlog.txt
/Nov 7
ESC
:q

commit 436f635fa183740832270ea7db6869918a786d55 * commit used for this run
commit 9908eb32975fe33075fefad73f411896456b4a84
commit 1ad187d4bff5b6936c065a1aaa15a654ef4d368c
commit 8ebd02ab9556a0c2869e9b551417a6e28d5f86cf

$ git diff master..436f635 train.py | grep "\-\-inputs"
-    parser.add_argument('--inputs', default='../dataset/unity/genRoad/log/*.jpg',
+    parser.add_argument('--inputs', default='../dataset/unity/log_sample/*.jpg',

$ git diff master..9908eb3297 train.py | grep "\-\-inputs"
-    parser.add_argument('--inputs', default='../dataset/unity/genRoad/log/*.jpg' (...)
+    parser.add_argument('--inputs', default='../dataset/unity/log_sample/*.jpg' (...)

$ git diff master..1ad187d4b train.py | grep "\-\-inputs"
-    parser.add_argument('--inputs', default='../dataset/unity/genRoad/log/*.jpg'
+    parser.add_argument('--inputs', default='../dataset/unity/log2/*.jpg',

$ git diff master..8ebd02ab train.py | grep "\-\-inputs"
-    parser.add_argument('--inputs', default='../dataset/unity/genRoad/log/*.jpg'
+    parser.add_argument('--inputs', default='../dataset/unity/log2/*.jpg',

The plus signal indicates lines added in the commit to the right, minus indicates
what does not exist in the commit to the right (9908eb3297) , where master is
commit 436f635fa (current commit at time or forensic code analysis).

First two commits indicate log_sample dataset was used containing 38684 images
files of
"small looping course" course
/log_sample(master)$ grep -nr jpg . | wc -l
38684
was used,
second two commits indicate log2 dataset (renamed genRoad) containing 280747
image files on the "Generated Road" course.
```

```
/genRoad(master)$ grep -nr jpg . | wc -l
280747 (files)
```

The commits are listed in descending chronological order, that is the first two datasets used to train on that day were log2 (later renamed genRoad), the following two datasets used were log_sample. The commit history suggests one of the two were used to generate the Nov 7 successful model. Since the test was run on "Generated Track" course, and genRoad contains images for "Generated Road" course, the assumption is sample_log (small_looping_circuit) dataset was used.

The sanity check model was then used to run predictions:

```
python predict_client.py
--model=../trained_models/sanity/20201120171015_sanity.h5
(...)
connecting to 127.0.0.1 9091
fps 10.416671685462958
fps 10.405931675595545
fps 11.030877564739708
fps 11.092089084017463
fps 11.466601323987536
fps 11.362325028068517
fps 11.359920970581536
fps 10.861855550070638
fps 10.50113576298416
(...)
```

The tcpflow log was saved to:

```
tcpflow(master)$ ls
20201107210627_sanity_tcpflow.log
```

And a video generated (from /tmp copy) with script src/MakeVideo.py:

```
$ python MakeVideo.py --filename=/tmp/tcpflow.log \
--model=20201120171015_sanity.h5
```

and the video.avi output uploaded to https://youtu.be/JaSkkh-2xtI

Qualitatively (by observation) 20201120171015_sanity.h5 had a better lap than 20201107210627_nvidia1.h5 (https://youtu.be/9z0mMtOnUUc) as it keeps to a single lane and the vehicle wheels never touch the road markings
Noting that the prior predict_client.py fps logs indicate that frames-per-second at varying ratios (e.g. fps 10.50113576298416), while MakeVideo.py records at 11

```
frames per second.
The video generated by MakeVideo.py is going around the track slightly faster
than the SDSandbox simulator steering with 20201120171015_sanity.h5 predictions.
between logged frames (tcpflow) and rendered frames (MakeVideo.py).
The frame rate logged with tcpflow is the frame rate SDSandbox is sending frames over
the TCP network.
```

Figure E.3 shows a still from video `https://youtu.be/v0Otdmtdhnk` (run 96 E.5.96).

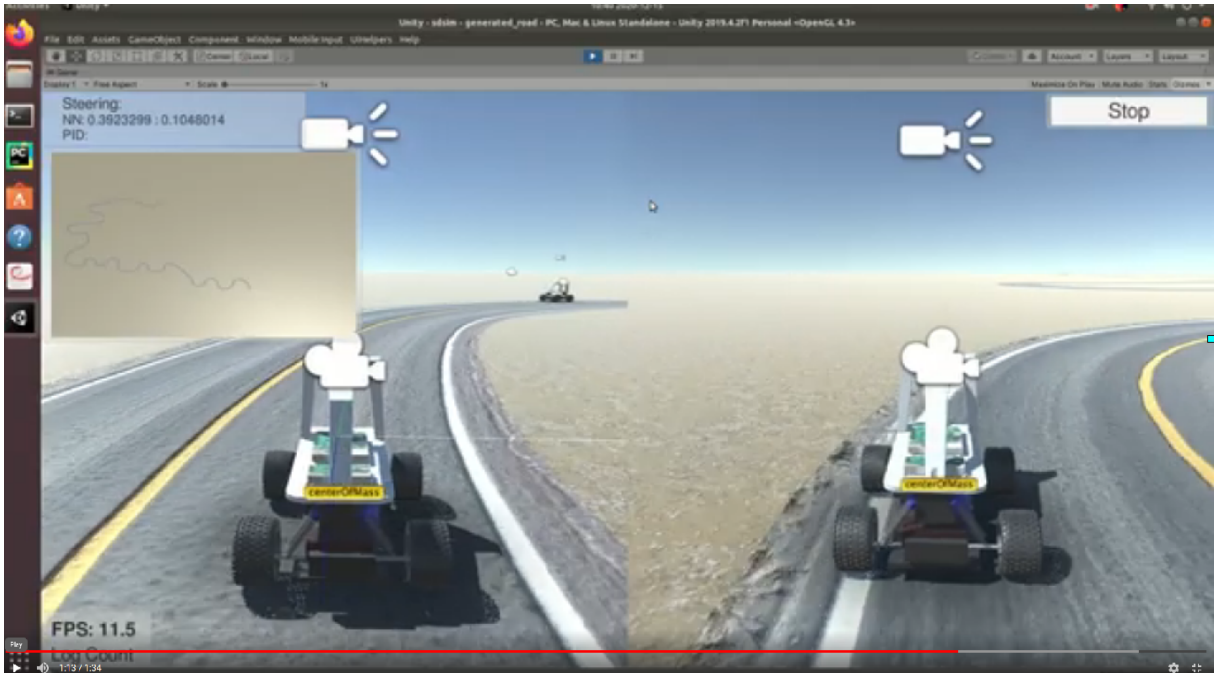

FIGURE E.3: Video still of models 20201120171015_ sanity.h5 (left) and 20201107210627_ nvidia1.h5 (right) driving side by side on a random Generated Road. The simulated car driven by predictions generated by 20201107210627_ sanity.h5 can be seen ahead of simulated car driving by predictions generated by 20201120171015_ sanity.h5 on the left frame. The video shows additional car is added by running prediction engine a second time, where added car appears on right frame.

E.5.37 Run 37 - 20201120184912_sanity.h5

```
Commit: 1ad187d4bff5b6936c065a1aaa15a654ef4d368c
Model: nvidia1
Outputs: 2
Dataset: log2 (genRoad)
Command: python train.py --model=sanity --outdir=../trained\_models
Environment: simbox
Comment: This was the 3rd commit of the day
It seems that this has something to do with the dataset that was
being used. To be confirmed. Now running nvidia1 on genRoad
```

```
both on simbox and camber, loss still converging to 0
and not in NAN range.

Training log:  ../trained_models/sanity/20201120184912\_sanity.log
Total training time: 16:08:01
Training loss: 0.010
Validation loss: 0.008
Training accuracy: 0.858
Validation accuracy: 0.858

This model took over 16 hours to train on 280727 images. It did not perform well.
Could it be due to the outliers?

$ sudo tcpflow -i lo -c port 9091 > /tmp/tcpflow.log
$ python predict_client.py --model=../trained_models/sanity/20201120184912_sanity.h5
$ python MakeVideo.py --filename=/tmp/tcpflow.log --model=20201120184912_sanity.h5

The angles were plotted using notebook GetSteeringAnglesFromtcpflow.ipynb
sa = GetSteeringFromtcpflow('../dataset/unity/genRoad/tcpflow/20201120184912_sanity.log')
sarr = np.asarray(sa)
p = sarr[:,0]
g = sarr[:,1]
plotSteeringAngles(p, g, 25, False, "Generated Road", "20201120184912_sanity.h5")
```

Below are the normalized histogram and bins recovered from tcplow logs for model 20201120184912_sanity.h5 referenced in **??**. The video generated from tcpflow log was uploaded to `https://youtu.be/xGDN8qOnv9M`.

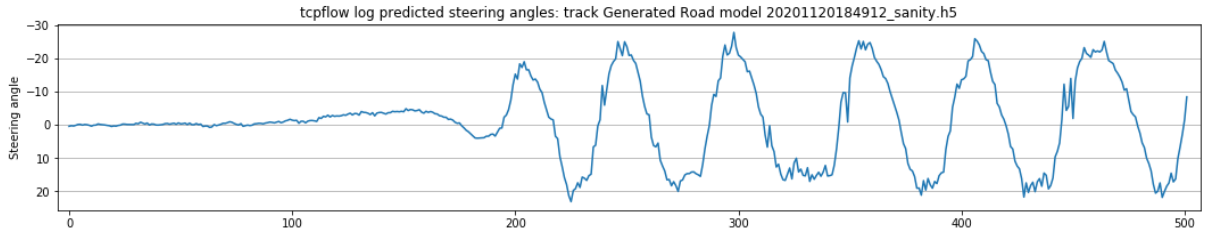

FIGURE E.4: Graph of steering angles recovered from tcpflow log genRoad/tcpflow/20201120184912_sanity_tcpflow.log for model 20201120184912_sanity.h5 driving on Generated Road

E.5.38 Run 38 - 20201123162643_ sanity.h5

```
Commit: 1ad187d4bff5b6936c065a1aaa15a654ef4d368c
```

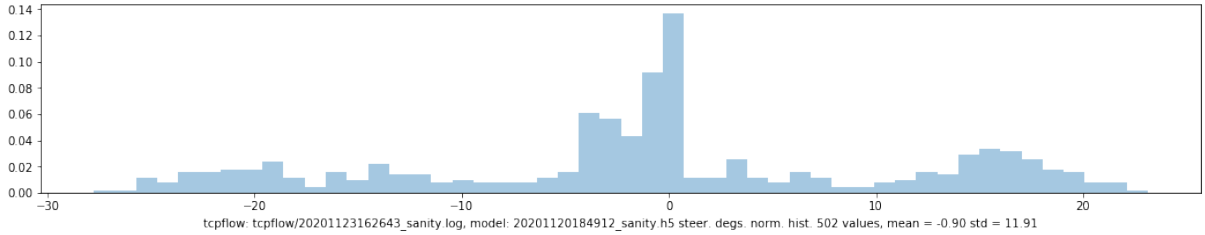

FIGURE E.5: Normalized histogram of tcpflow log genRoad/tcpflow/20201120184912_ sanity _ tcpflow.log for model 20201120184912_ sanity.log driving on Generated Road

```
Model: nvidia1
Outputs: 2
Dataset: genRoad (clean - logs_Thu_Jul__9_16_00_15_2020 quarantined)
Command: python train.py --model=sanity --outdir=../trained\_models --epochs=5
NB Trained for 5 epochs
Environment: simbox
Comment: Rerunning commit 1ad187d4bf with cleaned data. Drove off the road.
$ git checkout nov7 (this branch is commit 1ad187d4bff5b6936c065a1aaa15a654ef4d368c)
NB a simlink was created in unity/log2 pointing to unity/genRoad for the sake of
not changing committed source code.
ln -s ~/git/msc-data/unity/genRoad/ ~/git/msc-data/unity/log2

Epoch 1/5
1616/1616 [==============================] - 689s 427ms/step - loss: 0.0254
- acc: 0.8033 - val_loss: 0.0167 - val_acc: 0.8365
Epoch 2/5
1616/1616 [==============================] - 681s 421ms/step - loss: 0.0186
- acc: 0.8251 - val_loss: 0.0148 - val_acc: 0.8411
Epoch 3/5
1616/1616 [==============================] - 634s 393ms/step - loss: 0.0170
- acc: 0.8288 - val_loss: 0.0139 - val_acc: 0.8428
Epoch 4/5
1616/1616 [==============================] - 635s 393ms/step - loss: 0.0161
- acc: 0.8327 - val_loss: 0.0129 - val_acc: 0.8505
Epoch 5/5
1616/1616 [==============================] - 665s 411ms/step - loss: 0.0154
- acc: 0.8365 - val_loss: 0.0125 - val_acc: 0.8451

$ cat ../trained_models/sanity/20201123162643_sanity.log
Model name: ../trained_models/sanity/20201123162643_sanity.h5
Total training time: 0:55:10
Training loss: 0.015
```

```
Validation loss: 0.013
Training accuracy: 0.836
Validation accuracy: 0.845

The batch size used by generator function was 64:
src(nov7)$ cat train.py | grep batch
def generator(samples, is_training, batch_size=64):

Running the predictions:

$ sudo tcpflow -i lo -c port 9091 > /tmp/tcpflow.log
$ python predict_client.py --model=../trained_models/sanity/20201123162643\_sanity.h5
$ python MakeVideo.py --filename=/tmp/tcpflow.log --model=20201123162643\_sanity.h5
Video uploaded to https://youtu.be/uG2HvAbg2U4

tcpflow log file moved:
cp /tmp/tcpflow.log ../dataset/unity/genRoad/tcpflow/20201123162643_sanity.log

Plotting the bins with GetSteeringAnglesFromtcpflow.ipynb

sa = GetSteeringFromtcpflow('../dataset/unity/genRoad/tcpflow/20201123162643_sanity.log')
sarr = np.asarray(sa)
p = sarr[:,0]
p = sarr[:,0]
plotSteeringAngles(p, g, 25, False, "Generated Road", "20201123162643_sanity.h5")
plotBinsFromArray(p, 25, "20201123162643_sanity.h5", "tcpflow/20201123162643_sanity.log")
```

Steering data recovered from tcpflow log genRoad/tcpflow/20201123162643_sanity.log This was running with commit 1ad187d4bf where frame **is not preprocessed** for prediction (predict_client.py) (Figures E.6 and E.7).

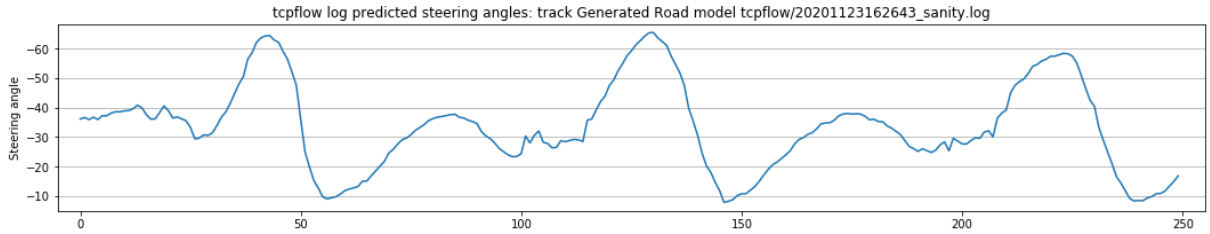

FIGURE E.6: Graph of steering angles recovered from tcpflow log genRoad/tcpflow/20201123162643_sanity.log for model 20201123162643_sanity.h5 driving on Generated Road
. The video can be seen here `https://youtu.be/uG2HvAbg2U4`

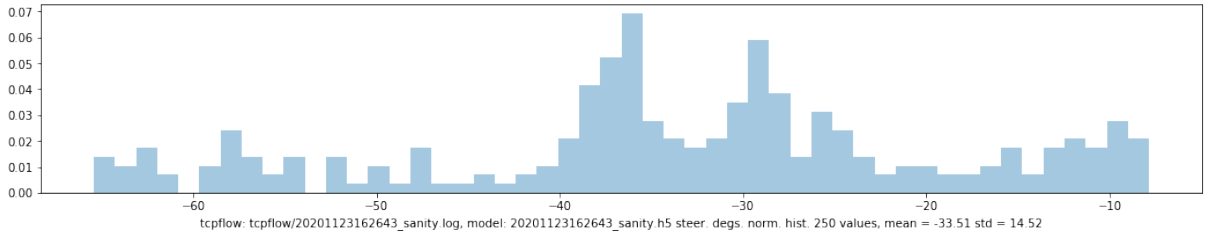

FIGURE E.7: Normalized histogram of tcpflow log genRoad/tcpflow/20201123162643_ sanity.log for model 20201123162643_ sanity.log driving on Generated Road

E.5.39 Run 39 - 20201123162643_sanity.h5

```
Commit: 7f3086490118fec1a20e99e93cc1b853a91272b6
Model:  N/A (trained model)
Outputs:N/A (trained model)
Dataset: N/A (trained model)
Command:
Environment:
Comment: Same as previous run, but on branch 7f308649 (predict_client.py) with
preprocessing

$ sudo tcpflow -i lo -c port 9091 (right angle bracket) /tmp/tcpflow.log
$ python predict_client.py
--model=../trained_models/sanity/20201123162643\_sanity.h5
$ python MakeVideo.py --filename=/tmp/tcpflow.log --model=20201123162643\_sanity.h5

Link: https://youtu.be/377O2E_LwyU

tcpflow log file moved:
cp /tmp/tcpflow.log ../dataset/unity/genRoad/tcpflow/20201123162643_sanity_pp.log

sa = GetSteeringFromtcpflow('../dataset/unity/genRoad/tcpflow/20201123162643_
sanity_pp.log')
sarr = np.asarray(sa)
p = sarr[:,0]
p = sarr[:,0]
plotSteeringAngles(p, g, 25, False, "Generated Road", "20201123162643
_sanity_pp.h5")
plotBinsFromArray(p, 25, "20201123162643_sanity.h5",
"tcpflow/20201123162643_sanity_pp.log")

Video: https://youtu.be/377O2E_LwyU
```

Steering data recovered from tcpflow log genRoad/tcpflow/20201123162643_sanity_pp.log This was running with commit 7f308649 where frame **is preprocessed** for prediction (predict_ client.py) (Figures E.8 and E.9). Note the steering angles are a much tighter range. Bias toward positive 6 is due to road bending right. The bins show a left skewed distribution, due to the right turn. Steering goes a bit crazy but model manages to keep vehicle on the road.

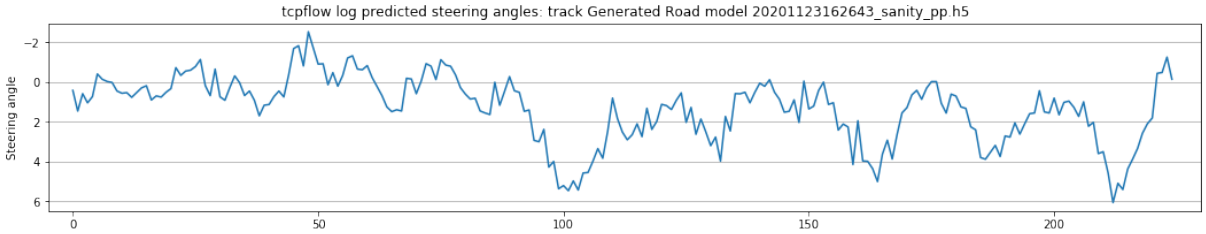

FIGURE E.8: Graph of steering angles recovered from tcpflow log genRoad/tcpflow/20201123162643_ sanity_ pp.log for model 20201123162643_ sanity.h5 driving on Generated Road
. The video can be seen here `https://youtu.be/37702E_LwyU`

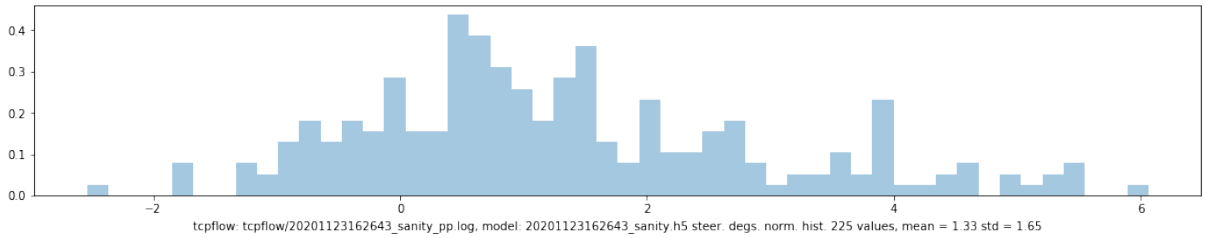

FIGURE E.9: Normalized histogram of tcpflow log genRoad/tcpflow/20201123162643_ sanity_ pp.log for model 20201123162643_sanity.h5 driving on Generated Road

E.5.40 Run 40 - 20201124032017_ nvidia2.h5

20201121090912_ nvidia_ baseline.h5

```
Commit: 7f3086490118fec1a20e99e93cc1b853a91272b6
Model: nvidia2
Outputs: 2
Dataset: N/A
Command: python3 train.py --model=nvidia2 --outdir=../trained_models --epochs=100
Environment: devcloud
Comment: Trained with nvidia2 model. Need to double check input size, as it might
be originally expecting 200x66.
Model did not do well and drove straight off the road. One thing to note, pixels are
being normalized and zero centered:
x = Lambda(lambda x: x/127.5 - 1.0)
Also, one single dropout layer. Need to try with different image size to match
original design.
```

```
$ sudo tcpflow -i lo -c port 9091 (right angle bracket) /tmp/tcpflow.log
$ python predict_client.py
--model=../trained_models/devcloud/nvidia2/20201124032017_nvidia2.h5
$ python MakeVideo.py --filename=/tmp/tcpflow.log --model=20201124032017\_nvidia2.h5

https://youtu.be/RW-luOWQ_qY

tcpflow log file moved:
cp /tmp/tcpflow.log ../dataset/unity/genRoad/tcpflow/20201124032017_nvidia2.log

sa = GetSteeringFromtcpflow('../dataset/unity/genRoad/tcpflow/20201124032017_nvidia2.log)
sarr = np.asarray(sa)
p = sarr[:,0]
p = sarr[:,0]
plotSteeringAngles(p, g, 25, False, "Generated Road", "20201124032017_nvidia2.h5")
plotBinsFromArray(p, 25, "20201124032017_nvidia2.h5", "tcpflow/20201124032017_nvidia2.log")
```

Run 40 started well then pretty much drove straight off the road - need to investigate image geometry

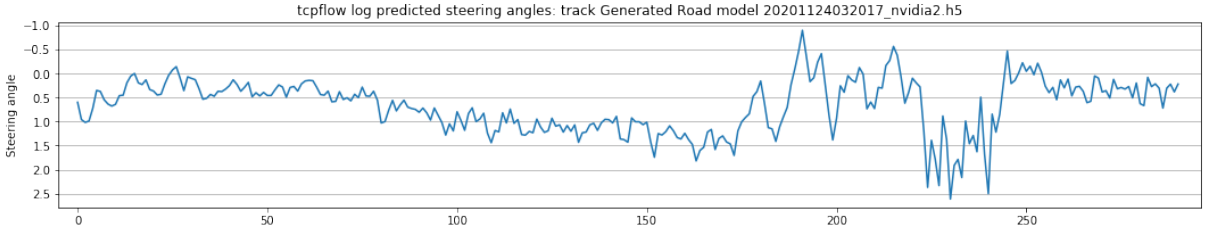

FIGURE E.10: Graph of steering angles recovered from tcpflow log genRoad/tcpflow/20201124032017_ nvidia2.log for model 20201124032017_ nvidia2.h5 driving on Generated Road
. The video can be seen here https://youtu.be/RW-luOWQ_ qY

FIGURE E.11: Normalized histogram of tcpflow log genRoad/tcpflow/20201124032017_ nvidia2.log for model 20201124032017_ nvidia2.h5 driving on Generated Road

E.5.41 Run 41 - 20201120171015_ sanity.h5

```
Commit: f0d2513
Model: nvidia1
Outputs: 2
Dataset: log_sample (small looping circuit)

Command:

$ sudo tcpflow -i lo -c port 9091 > /tmp/tcpflow_20201120171015_sanity.log
$ python predict_client.py --model=../trained_models/sanity/20201120171015_sanity.h5
$ python MakeVideo.py --filename=/tmp/tcpflow_20201120171015_sanity.log
--model=20201120171015_sanity.h5
# mv
$ mv /tmp/tcpflow_20201120171015_sanity.log \
~/git/sdsandbox/trained_models/sanity/tcpflow/

Generating plot:
$ python steerlib.py
$ python
>>> import steerlib as sl
>>> sa = sl.getSteeringFromtcpflow('../../dataset/unity/genRoad/tcpflow/tcpflow_20201
120171015_sanity.log')
>>> sarr = np.asarray(sa)
>>> p = sarr[:,0]

plot = sl.plotSteeringAngles(p, None, 25, True, "Generated Track",
"20201120171015_sanity.h5", 'tcpflow log predicted')
Environment: simbox
Comment: Prediction to generate tcpflow log
video https://youtu.be/LEmZJJzJkEE
```

E.5.42 Run 42 - 20201120171015_ sanity.h5

```
Commit: 444c27f
Model: Outputs: Dataset: Command: Environment: Same as run 41
Comment: Testing conversion of PIL image to numpy array, bypassing saving .jpg to
disk and reading from disk.

git diff ef4b48e..444c27f1 MakeVideo.py
diff --git a/src/utils/MakeVideo.py b/src/utils/MakeVideo.py
index f469471..8abd3c8 100644
```

```
--- a/src/utils/MakeVideo.py
+++ b/src/utils/MakeVideo.py
(...)
                    image = Image.open(BytesIO(base64.b64decode(imgString)))
+                   # try to convert to jpg
+                   image = np.array(image)
(...)
```

Video still (Figure E.12) showing effect of converting PIL image directly to numpy array. Colour of sky is different. No video was uploaded to youtube.

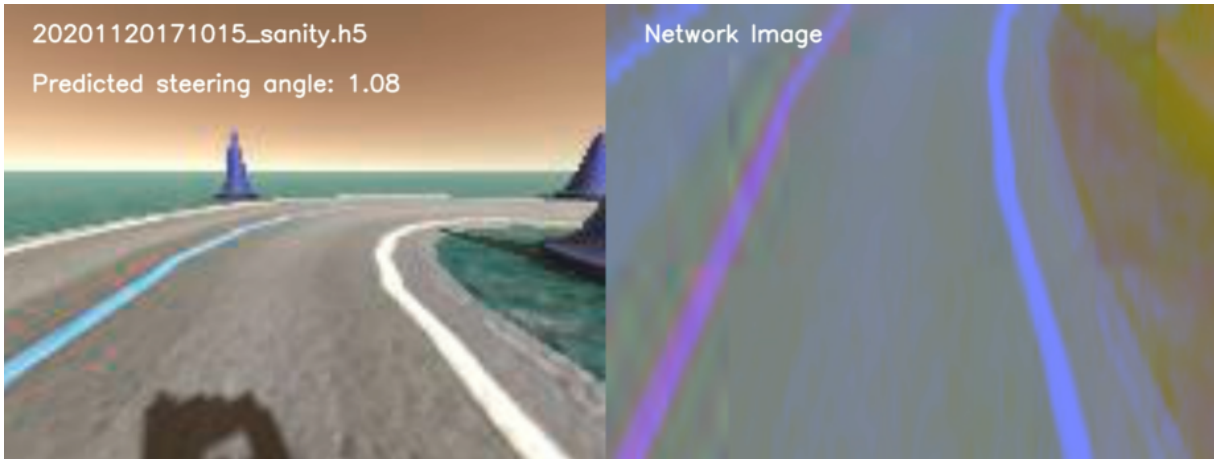

FIGURE E.12: tcpflow log image converted directly to numpy array (left) and image presented to network (right).

E.5.43 Run 43 - Record triple video

On this run we record a video with 3 frames side by side containing the simulator image, simulator image with added rain and the preprocessed image with added rain.

Interesting observation here (pending two tests to be documented), when we change the order of added rain, i.e. added to image before preprocesssing or added to image after preprocessing, this affects the outcome. Conclusion is currently pointing to the fact that moving image space from RGB to YUV is filtering out the rain "noise"

```
Commit: b0a77d2
Model: Outputs: Dataset: Environment: Same as run 41
Command:
$ python predict_client.py --model=../trained_models/sanity/20201120171015_sanity.h5
--rain=torrential --slant=20 --record=true
```

```
Comment: As observed, if noise is added to preprocessed image (at the point of being
presented to network), sim crashes.
```

Video: https://youtu.be/57jwwcjbfdE

E.5.44 Run 44 - Intensity x4

```
Commit: 9696f3cee
Model: Outputs: Dataset: Environment: Same as run 41
Command:
$ python predict_client.py --model=../trained_models/sanity/20201120171015_sanity.h5 \
--rain=heavy --slant=20 --record=True

Comment: Changed luminosity multiply manually
tcpflow file: (TODO plot steering including crash section)
~/git/sdsandbox/trained_models/sanity/tcpflow/20201120171015_sanity_run44.log

Video: https://youtu.be/UBd38Hlfv4w
```

E.5.45 Run 45 - Intensity x8

```
Commit: 20fe20c2e
Model: Outputs: Dataset: Environment: Same as run 41
Command:
$ python predict_client.py --model=../trained_models/sanity/20201120171015_sanity.h5
--rain=heavy --slant=20 --record=True
Comment: Changed intensity manually to 8, vehicle stays on road for 2 laps.
Higher intensity means higher contrast between road and non-road, perhaps creating
a guiding pattern.
It suggests there is some generalisation, as road is very unfamiliar.

tcpflow file:
~/git/sdsandbox/trained_models/sanity/tcpflow/20201120171015_sanity_run45.log

Video https://youtu.be/NerkRCSquiE
```

E.5.46 Run 46 - 20201203164029_ nvidia1.h5

```
Commit: 2d9e6983
```

```
Model: N/A
Outputs: N/A
Dataset: genTrack
Command:
$ python train.py --model=nvidia1 --outdir=../trained_models --epochs=100
--inputs=../dataset/genTrackOneLap/*.jpg --aug=true --preproc=true
--inputs=../dataset/unity/genTrack/*.jpg
Environment: devbox
Comment: Gathered dataset (35967 files) moved to ../dataset/unity/genTrack
ls -R | grep .jpg | wc -l
35967
Laps recorded with settings:
Max Speed 1.963599
Prop: 24
Diff: 5
Steer Max: 25
```

E.5.47 Run 47 - 20201206171648_ nvidia1.h5

```
Commit: 3f0622c8
Model: nvidia1
Outputs: 2
Dataset: genRoad
Command: python train.py
--model=nvidia1
--outdir=../trained_models
--epochs=5
--inputs=../dataset/unity/genRoad/*.jpg
--aug=True
--preproc=True
Environment: simbox
Comment: Created Augmentation class, handling image sizes for different models.
Then ran:
python predict_client.py --model=../trained_models/nvidia1/20201206171648_nvidia1.h5

When running this model (and possibly all previous):

$ python predict_client.py --model=../trained_models/nvidia1/20201206171648_nvidia1.h5

WARNING:tensorflow:Model was constructed with shape (None, 160, 120, 3) for input
Tensor("img_in:0", shape=(None, 160, 120, 3), dtype=float32), but it was called on
an input with incompatible shape (None, 120, 160, 3).
```

As it turns out, it was an assignment mess up. See diff on models.py between runs 48 and 49.

Still the "flipped" input model did manage a stretch of Generated Road and surprisingly manages to get around Generated Track, which is a total fluke, which may prove serendipitous.

TODO ADD TO DISCUSSION (TOTAL FLUKE FLIP)

E.5.48 Run 48 - 20201206211122_ nvidia1.h5

Commit: b5ad97f
Model: nvidia1
Outputs: 2
Dataset: genRoad
Command: python train.py
--model=nvidia1
--outdir=../trained_models
--epochs=5
--inputs=../dataset/unity/genRoad/*.jpg
--aug=True
--preproc=True
Environment: simbox
Comment: Based on previous (Run 47 warning), input to network was changed to col, row, channels. NB This was an oversight. In the next run, col, row remain, the assignment changes (to fix mix-up).

This model did really well on Generated Road but not well at all on Generated Track.

E.5.49 Run 49 - 20201207091932_ nvidia1.h5

Commit: 55cb00b
Model: nvidia1
Outputs: 2
Dataset: genTrack
Command:
$ python train.py
--model=nvidia1
--outdir=../trained_models
--epochs=5
--inputs=../dataset/unity/genTrack/*.jpg

```
--aug=True
--preproc=True

Environment: simbox
Comment: Trained for 5 epochs, maximum validation accuracy 0.7644. This model
drives both on Generated Road and Generated Track. This model took 1m23s to
train.

$ cat ../trained_models/nvidia1/20201207091932_nvidia1.log
Model name: ../trained_models/nvidia1/20201207091932_nvidia1.h5
Total training time: 0:01:23
Training loss: 0.021
Validation loss: 0.015
Training accuracy: 0.751
Validation accuracy: 0.757

Recording with no rain:
python predict_client.py
--model=../trained_models/nvidia1/20201207091932_nvidia1.h5 --record=True
https://youtu.be/S-PBNgJ_mdY

tcpflow log:
../trained_models/nvidia1/tcpflow/20201207091932_nvidia1_tcpflow.log
```

Video of image plus network image - no rain - Generated Road: `https://youtu.be/S-PBNgJ_mdY`.

Figure E.13 shows the training loss and accuracy plot for 20201207091932_ nvidia1.h5. The x axis divisions should be interpreted as 0.0 values obtained at the end of first training epoch. 1.0 as the values obtained at the end of second training epoch, and so forth.

Figure E.14 shows a slight positive value (road turning right) to start with, then a negative value (road turning left) steering angle.

E.5.50 Run 50 - 20201207111940_ nvidia2.h5

```
Commit: f9325ff
Model: nvidia2
Outputs: 2
Dataset: genTrack
Command: python train.py
--model=nvidia2
--outdir=../trained_models
```

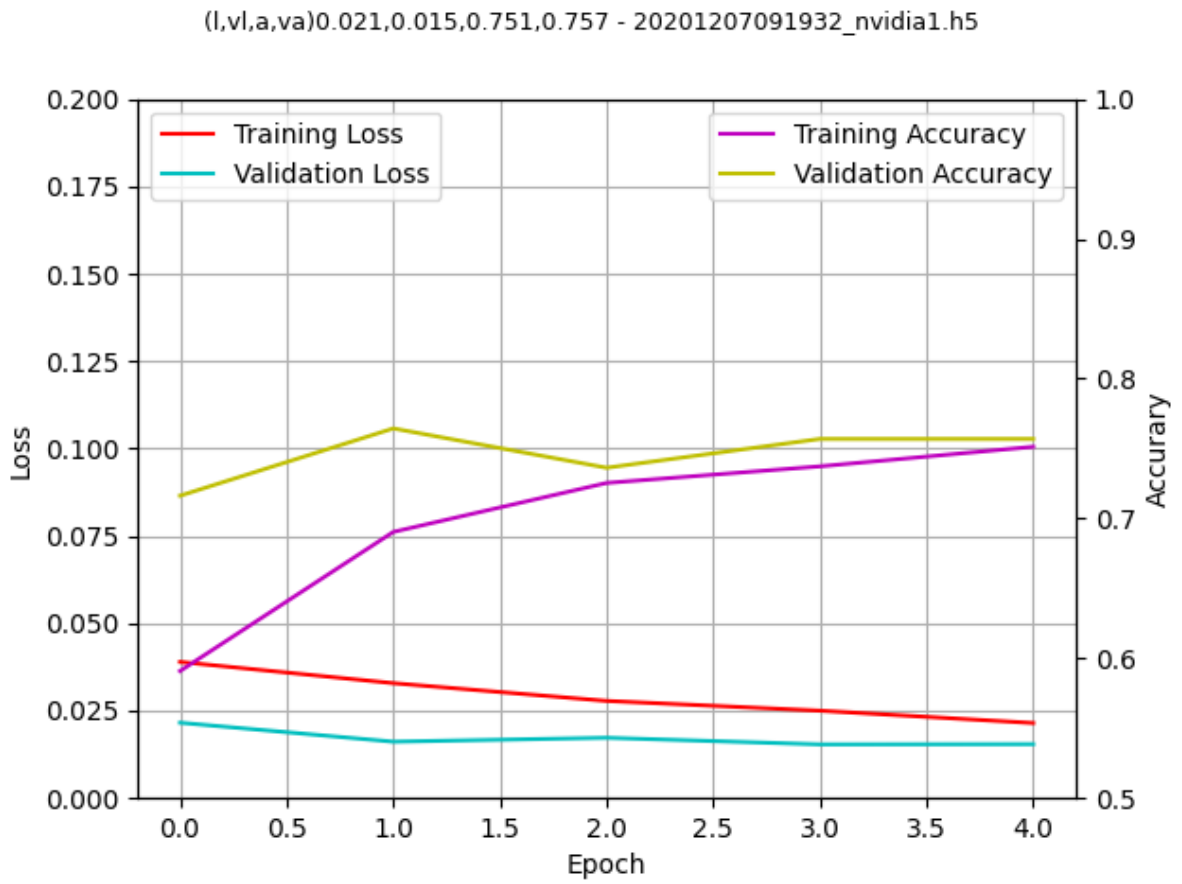

FIGURE E.13: 20201207091932_ nvidia1.h5 testing and validation loss and accuracy plots

```
--epochs=5
--inputs=../dataset/unity/genTrack/*.jpg
--aug=True
--preproc=True

Environment: simbox
Comment: Model trained with 0.5 dropout in a single layer. nvidia1 is trained
with 5 dropout layers at 0.1 each.

Running the model:
$ cd ~/Downloads
$ ./startunity.sh
$ sudo tcpflow -i lo -c port 9091 > /tmp/tcpflow.log
python predict_client.py
--model=../trained_models/nvidia2/20201207111940_nvidia2.h5
--modelname=nvidia2
--record=True
```

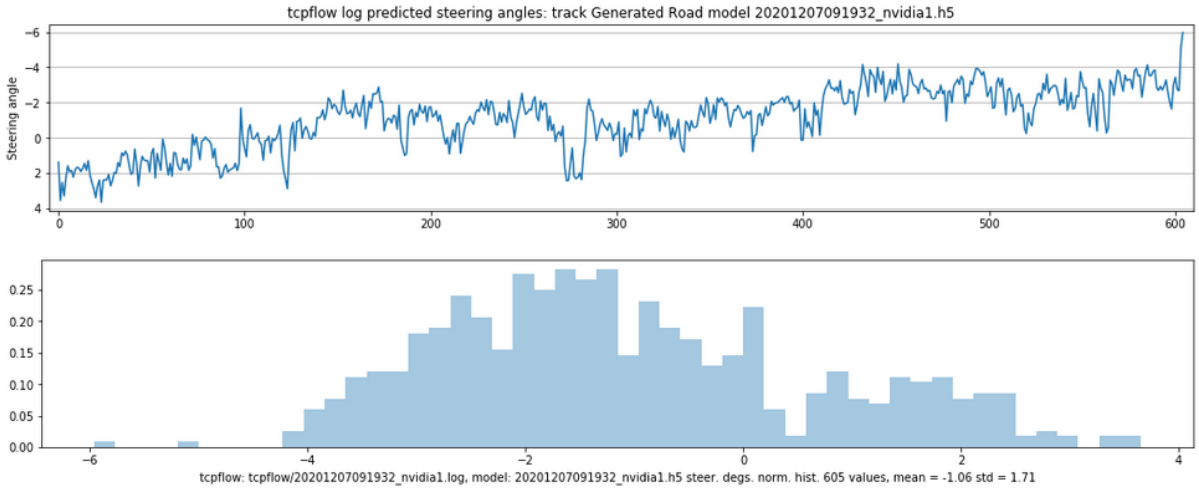

FIGURE E.14: tcpflow steering angle log for model 20201207091932_ nvidia1.h5 trained over 5 epochs on Generated Track, self-driving in a section of Generated Road

`Video: https://youtu.be/b6IIoHuiUQ8`

Figure E.15 shows a tcpflow of the nvidia2 model driving off the road by over-steering to the left after the first right turn as can be seen in video `https://youtu.be/b6IIoHuiUQ8`

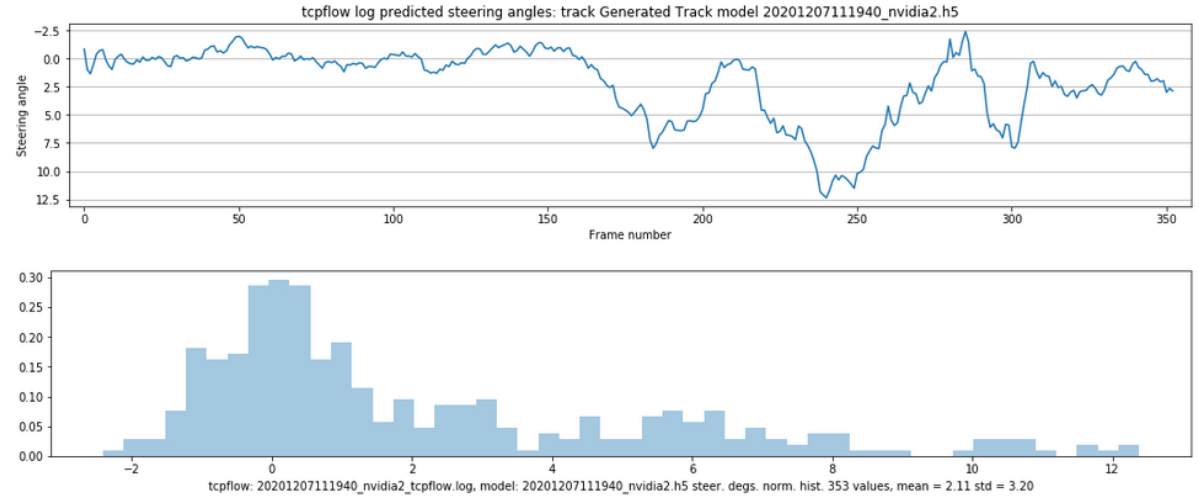

FIGURE E.15: tcpflow steering angle log for model 20201207111940_ nvidia2.h5 trained over 5 epochs on Generated Roard, self-driving off the road after first right turn.

E.5.51 Run 51 - 20201207124146_ nvidia2.h5

`Commit: bb66ed7`

`Comment: Same as run 50, with dropout set to 0.25`

```
$ python predict_client.py --model=../trained_models/nvidia2/20201207124146_nvidia2.h5
--modelname=nvidia2 --record=True

Video: https://youtu.be/-lDaiodokxw
```

Figure E.16 shows a tcpflow log of the nvidia2 model driving off the road by over-steering to the left off the first right turn as can be seen in video `https://youtu.be/-lDaiodokxw`. The model was trained with a single 0.25 dropout layer

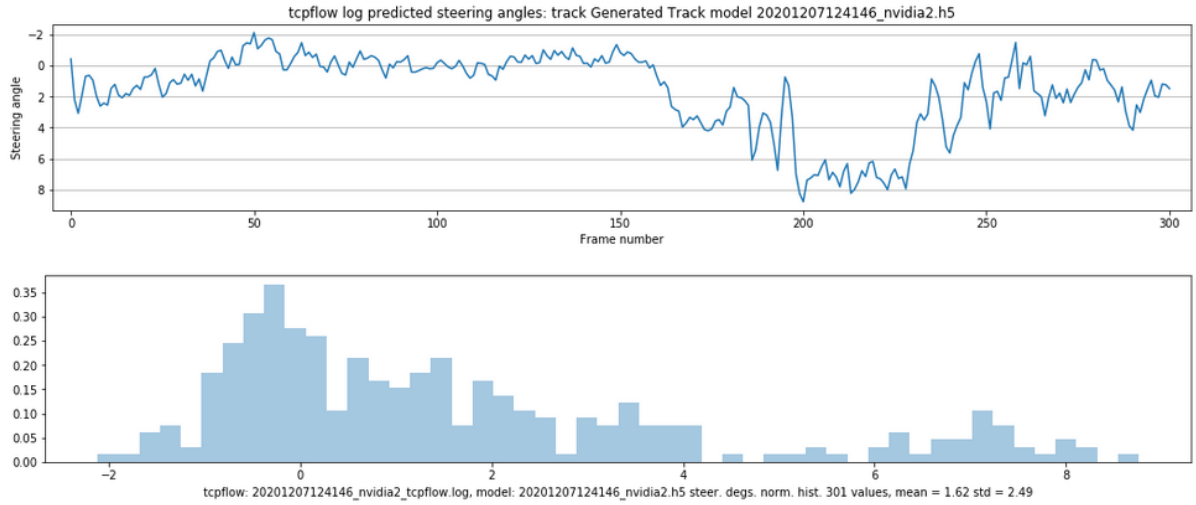

FIGURE E.16: tcpflow steering angle log for model 20201207124146_ nvidia2.h5 trained over 5 epochs on Generated Track, with droupout set to 0.25, self-driving off the road after first right turn.

E.5.52 Run 52 - 20201207132429_ nvidia2.h5

```
Commit: cd3aa3a
Comment: Same as 51, with dropout set to 0.1

$ python predict_client.py
--model=../trained_models/nvidia2/20201207132429_nvidia2.h5
--modelname=nvidia2 --record=True

https://youtu.be/I6aB5RxYPsg
```

Another case where nvidia2 drives off the road on the first right turn (Figure E.17). `https://youtu.be/I6aB5RxYPsg`.

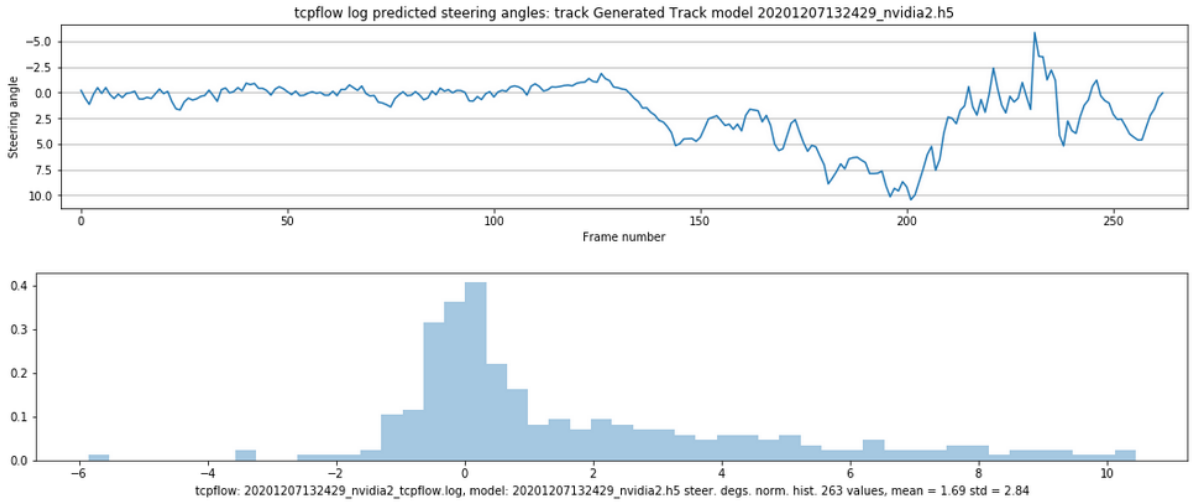

FIGURE E.17: tcpflow steering angle log for model 220201207132429_ nvidia2.h5 trained over 5 epochs on Generated Road, with dropout set to 0.1, self-driving off the road after first right turn.

E.5.53 Run 53 - 20201207133600_ nvidia2.h5

```
Commit: 3e3ff1a
Comment: Same as 52 except last dense layer (10 units) removed
$ git diff --name-only cd3aa3a..3e3ff1a
src/GetSteeringAnglesFromtcpflow.ipynb
src/models.py

$ git diff cd3aa3a..3e3ff1a models.py
diff --git a/src/models.py b/src/models.py
index 0ef889e..8efe891 100644
--- a/src/models.py
+++ b/src/models.py
@@ -165,7 +165,7 @@ def nvidia_model2(num_outputs):
     # x = Dropout(drop)(x)
     x = Dense(50, activation='elu')(x)
     # x = Dropout(drop)(x)
-    x = Dense(10, activation='elu')(x) # Added in Naoki's model
+    # x = Dense(10, activation='elu')(x) # Added in Naoki's model
```

Slightly better but still drives off road past the second left bollard.

The model:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
```

```
img_in (InputLayer)          [(None, 66, 200, 3)]      0
_________________________________________________________________
lambda (Lambda)              (None, 66, 200, 3)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 31, 98, 24)        1824
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 14, 47, 36)        21636
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 5, 22, 48)         43248
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 3, 20, 64)         27712
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 1, 18, 64)         36928
_________________________________________________________________
dropout (Dropout)            (None, 1, 18, 64)         0
_________________________________________________________________
flattened (Flatten)          (None, 1152)              0
_________________________________________________________________
dense (Dense)                (None, 100)               115300
_________________________________________________________________
dense_1 (Dense)              (None, 50)                5050
_________________________________________________________________
steering_throttle (Dense)    (None, 2)                 102
=================================================================
Total params: 251,800
Trainable params: 251,800
Non-trainable params: 0
_________________________________________________________________
[(None, 66, 200, 3)]
```

Video: `https://youtu.be/DPxFLukN0Ws`

E.5.54 Run 54 - 20201207141017_ nvidia2.h5

```
Commit: a39008c
Comment: Same as run 53 except dropout layers added as per nvidia1. Note
zero-centered pixel values in git diff below:

$ git diff 3e3ff1a..a39008c models.py
diff --git a/src/models.py b/src/models.py
index 8efe891..28e6d62 100644
--- a/src/models.py
```

```
+++ b/src/models.py
@@ -149,13 +149,13 @@ def nvidia_model2(num_outputs):
     x = Lambda(lambda x: x/127.5 - 1.0)(x) # normalize and re-center
     # x = Lambda(lambda x: x / 255.0)(x)
     x = Conv2D(24, (5, 5), strides=(2, 2), activation='elu', name="conv2d_1")(x)
-    # x = Dropout(drop)(x)
+    x = Dropout(drop)(x)
     x = Conv2D(36, (5, 5), strides=(2, 2), activation='elu', name="conv2d_2")(x)
-    #x = Dropout(drop)(x)
+    x = Dropout(drop)(x)
     x = Conv2D(48, (5, 5), strides=(2, 2), activation='elu', name="conv2d_3")(x)
-    #x = Dropout(drop)(x)
+    x = Dropout(drop)(x)
     x = Conv2D(64, (3, 3), activation='elu', name="conv2d_4")(x) # default strides=(1,1)
-    #x = Dropout(drop)(x)
+    x = Dropout(drop)(x)
     x = Conv2D(64, (3, 3), activation='elu', name="conv2d_5")(x)
     x = Dropout(drop)(x)
```

```
Veers off road a little bit after run 53.
```

Video: `https://youtu.be/SQQJacV622E`

Network:

```
import models
mymodel = models.nvidia_model2(2)
models.show_model_summary(mymodel)
Model: "model_1"
Layer (type)                 Output Shape              Param #
=================================================================
img_in (InputLayer)          [(None, 66, 200, 3)]      0
_________________________________________________________________
lambda (Lambda)              (None, 66, 200, 3)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 31, 98, 24)        1824
_________________________________________________________________
dropout (Dropout)            (None, 31, 98, 24)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 14, 47, 36)        21636
_________________________________________________________________
dropout_1 (Dropout)          (None, 14, 47, 36)        0
_________________________________________________________________
```

```
conv2d_3 (Conv2D)            (None, 5, 22, 48)         43248
_________________________________________________________________
dropout_2 (Dropout)          (None, 5, 22, 48)         0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 3, 20, 64)         27712
_________________________________________________________________
dropout_3 (Dropout)          (None, 3, 20, 64)         0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 1, 18, 64)         36928
_________________________________________________________________
dropout_4 (Dropout)          (None, 1, 18, 64)         0
_________________________________________________________________
flattened (Flatten)          (None, 1152)              0
_________________________________________________________________
dense (Dense)                (None, 100)               115300
_________________________________________________________________
dense_1 (Dense)              (None, 50)                5050
_________________________________________________________________
steering_throttle (Dense)    (None, 2)                 102
=================================================================
Total params: 251,800
Trainable params: 251,800
Non-trainable params: 0
_________________________________________________________________
[(None, 66, 200, 3)]
(None, 66, 200, 3)
(None, 31, 98, 24)
(None, 31, 98, 24)
(None, 14, 47, 36)
(None, 14, 47, 36)
(None, 5, 22, 48)
(None, 5, 22, 48)
(None, 3, 20, 64)
(None, 3, 20, 64)
(None, 1, 18, 64)
(None, 1, 18, 64)
(None, 1152)
(None, 100)
(None, 50)
(None, 2)
```

E.5.55 Run 55 - 20201207142129_ nvidia2.h5

`Commit: 28ea337`

`Comment: Same as 54, except aligned number of nvidia2 feature maps with nvidia1`

Another crash (`https://youtu.be/3ipZZgGtJUA`). Given that nvidia1 () run 52 did so well and every subsequent nvidia2 has crashed. Let's have a look at the actual crop we are presenting to both networks (Figure E.18):

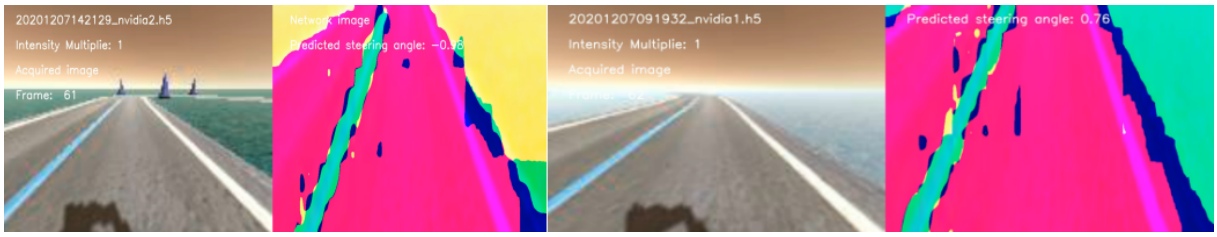

FIGURE E.18: Network images presented to networks nvidia2 (20201207142129_ nvidia2.h5) network (second from left) and nvidia1 (20201207091932_ nvidia1.h5), first on right

What is most noticeable is that the nvidia1 network image seems to somehow augment the road information, by making it cover more of the image. If we look at the actual dimension crops up to commit 28ea337 on master branch, in conf.py:

```
# NVIDIA1
NVIDIA1 = 'nvidia1' # a.k.a. TawnNet
nvidia1_img_dims = [160,120,3,160,120,60,-25]
# NVIDIA2
NVIDIA2 = 'nvidia2' # a.k.a. NaokiNet
nvidia2_img_dims = [320,160,3,200,66,70,-35]
```

where the last two digits on the list can be understood as the amount of top and bottom crop. In the case of nvidia1, for a total height of 120, a crop of index 60 till index (120 + (-25)) 95 (pixels) (total crop height = 35 pixels) is being made. In the case of nvidia2, for a total height of 160 a crop of index 70 until index (160 + (-35)) 125 (total crop height = 55 pixels) is being made. In the case of nvidia1, the crop, as a proportion of the original image height is $35/120 = 0.21$. In the case of nvidia1, the crop, as a proportion of the original image height is $55/160 = 0.34$. The bottom crop is the extent required to remove the vehicle shadow (if any) from the frame, so does not change. To change the top crop index (the conf.py IMG_ TOP_ CROP_ IDX index value of nvidia2_ img_ dims) such that the proportion of both cropped images presented to networks nvidia1 and nvidia2 is the same, the top crop value of nvidia2 must change to approximately $x/160 = 0.21, x = 34, tc = 160 - 35 - 34 = 91$, where tc is the top crop value, that is, setting value of nvidia2_ img_ dims to

```
# NVIDIA2
NVIDIA2 = 'nvidia2' # a.k.a. NaokiNet
```

```
nvidia2_img_dims = [320,160,3,200,66,91,-35]
```

Figure E.19 shows from left to right, the nvidia1 cropped image, followed by the nvidia2 cropped images with top crop value set to 70, 81 and 91 pixels.

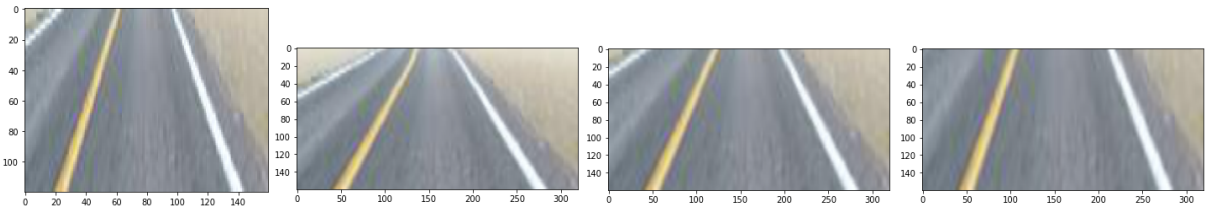

FIGURE E.19: Left to right, nvidia1 network crop, nvidia2 network crops at top crop set to 70, 81 and 91 pixels respectively)

The network architecture for run 55, aligned with nvidia1 convolutional feature maps, is 24, 32, 64, 64 and 64 feature maps in the convolutional layers (conv2d_ 1 through conv2d_ 5):

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
img_in (InputLayer)          [(None, 66, 200, 3)]      0
_________________________________________________________________
lambda (Lambda)              (None, 66, 200, 3)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 31, 98, 24)        1824
_________________________________________________________________
dropout (Dropout)            (None, 31, 98, 24)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 14, 47, 32)        19232
_________________________________________________________________
dropout_1 (Dropout)          (None, 14, 47, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 5, 22, 64)         51264
_________________________________________________________________
dropout_2 (Dropout)          (None, 5, 22, 64)         0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 3, 20, 64)         36928
_________________________________________________________________
dropout_3 (Dropout)          (None, 3, 20, 64)         0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 1, 18, 64)         36928
_________________________________________________________________
dropout_4 (Dropout)          (None, 1, 18, 64)         0
_________________________________________________________________
```

```
flattened (Flatten)          (None, 1152)              0
_________________________________________________________________
dense (Dense)                (None, 100)               115300
_________________________________________________________________
dense_1 (Dense)              (None, 50)                5050
_________________________________________________________________
steering_throttle (Dense)    (None, 2)                 102
=================================================================
Total params: 266,628
Trainable params: 266,628
Non-trainable params: 0
_________________________________________________________________
[(None, 66, 200, 3)]
(None, 66, 200, 3)
(None, 31, 98, 24)
(None, 31, 98, 24)
(None, 14, 47, 32)
(None, 14, 47, 32)
(None, 5, 22, 64)
(None, 5, 22, 64)
(None, 3, 20, 64)
(None, 3, 20, 64)
(None, 1, 18, 64)
(None, 1, 18, 64)
(None, 1152)
(None, 100)
(None, 50)
(None, 2)
```

E.5.56 Run 56 - 20201207170938_ nvidia2.h5

```
Commit: 8f163e9
Model: nvidia2
Outputs: 2
Dataset: genTrack
Command: Assumed to be the same as previous, which can be backtracked to run 50:
Command: python train.py
--model=nvidia2
--outdir=../trained_models
--epochs=5
--inputs=../dataset/unity/genTrack/*.jpg
--aug=True
```

```
--preproc=True

Environment: simbox
Comment: Changed top crop for nvidia2

$ git diff 28ea337..8f163e9 conf.py
diff --git a/src/conf.py b/src/conf.py
index 0ccb633..0b97b8e 100644
--- a/src/conf.py
+++ b/src/conf.py
@@ -42,10 +42,10 @@ NVIDIA1 = 'nvidia1' # a.k.a. TawnNet
 nvidia1_img_dims = [160,120,3,160,120,60,-25]
 # NVIDIA2
 NVIDIA2 = 'nvidia2' # a.k.a. NaokiNet
-nvidia2_img_dims = [320,160,3,200,66,70,-35]
+nvidia2_img_dims = [320,160,3,200,66,81,-35]
 # NVIDIA_BASELINE
 NVIDIA_BASELINE = 'nvidia_baseline' # a.k.a. NaokiNet
-nvidia_baseline_img_dims = [320,160,3,200,66,70,-35]
+nvidia_baseline_img_dims = [320,160,3,200,66,91,-35]
```

Another crash (https://youtu.be/I3gkoO1yWkk), changed top crop to 81 in conf.py.

E.5.57 Run 57 - 20201207175205_ nvidia2.h5

```
Commit: 485d845
Model: nvidia2
Outputs: 2
Dataset: genTrack
Command:
python train.py
--model=nvidia2
--outdir=../trained_models
--epochs=5
--inputs=../dataset/unity/genTrack/*.jpg
--aug=True
--preproc=True

Environment: simbox
Comment: Same as run 56, done as sanity check. Might have messed up the
predict_client.py parameters in 56. This model almost goes off the track on the first
```

```
corner but actually gets around the track and works with a 81 top crop.

python predict_client.py
--model=../trained_models/nvidia2/20201207175205_nvidia2.h5
--modelname=nvidia2 --record=True --modelname=nvidia2
```

This actually managed to get around the track (`https://youtu.be/Vqd2W39CwdA`. Investigating the logs for the run 57 model:

```
/trained_models/nvidia2(master)$ ls -l *20201207175205*
-rw-rw-r-- 1 simbox simbox   41146 Dec  7 17:53 20201207175205_nvidia2_accuracy.png
-rw-rw-r-- 1 simbox simbox 3278432 Dec  7 17:53 20201207175205_nvidia2.h5
-rw-rw-r-- 1 simbox simbox     258 Dec  7 17:53 20201207175205_nvidia2.history
-rw-rw-r-- 1 simbox simbox     195 Dec  7 17:53 20201207175205_nvidia2.log

$ cat 20201207175205_nvidia2.log
Model name: ../trained_models/nvidia2/20201207175205_nvidia2.h5
Total training time: 0:01:41
Training loss: 0.028
Validation loss: 0.014
Training accuracy: 0.716
Validation accuracy: 0.768
```

And the equivalent for run 56:

```
/trained_models/nvidia2(master)$ ls -l *20201207170938*
-rw-rw-r-- 1 simbox simbox   42312 Dec  7 17:11 20201207170938_nvidia2_accuracy.png
-rw-rw-r-- 1 simbox simbox 3278432 Dec  7 17:11 20201207170938_nvidia2.h5
-rw-rw-r-- 1 simbox simbox     258 Dec  7 17:11 20201207170938_nvidia2.history
-rw-rw-r-- 1 simbox simbox     195 Dec  7 17:11 20201207170938_nvidia2.log

/trained_models/nvidia2(master)$ cat 20201207170938_nvidia2.log
Model name: ../trained_models/nvidia2/20201207170938_nvidia2.h5
Total training time: 0:01:40
Training loss: 0.031
Validation loss: 0.016
Training accuracy: 0.689
Validation accuracy: 0.772
```

Looking at the training history for run 56 (Figure E.21) and run 57 (Figure E.20):

As a final sanity check, a rerun of the inference time drive is done for both models (Run 56 and Run 57, generating the video only from tcpflow data:

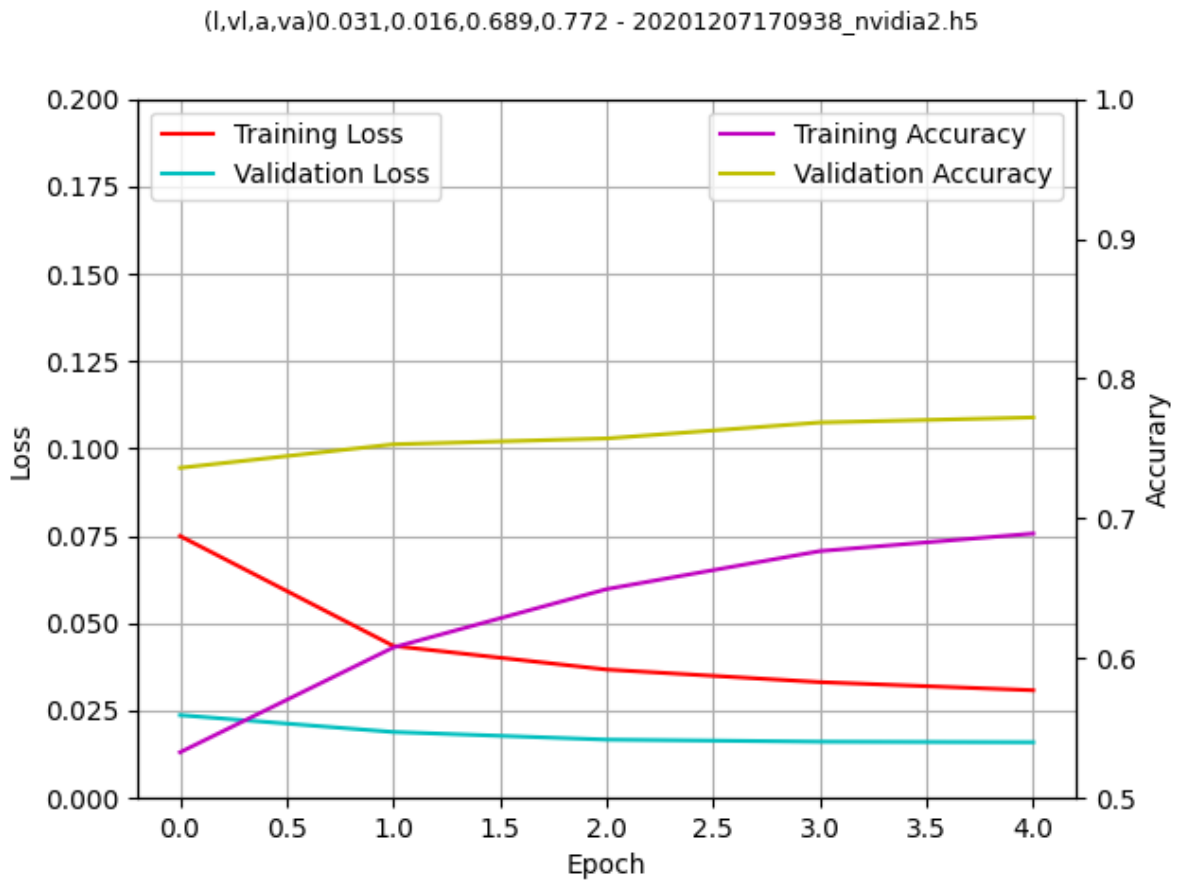

FIGURE E.20: nvidia2 20201207170938_ nvidia2.h5 (Run 56) model training accuracy for 5 epochs. Refer to E.21 for explanation of plot title

```
# Run 56
# record tcpflow log
$ sudo tcpflow -i lo -c port 9091 >
../trained_models/nvidia2/tcpflow/20201207170938_Rerun_56_nvidia2.log
# predict
$ python predict_client.py
--model=../trained_models/nvidia2/20201207170938_nvidia2.h5 --modelname=nvidia2
# make video - had to copy augmentation.py and conf.py to utils (at commit c4158dd), where Ma
# lives
$ python MakeVideo.py
--filename=../../trained_models/nvidia2/tcpflow/20201207170938_Rerun_56_nvidia2.log
--model="20201207170938_nvidia2.h5"
```

This produced video `https://youtu.be/XGA2oL8QVms` which matches run 56.

Now the same for run 57:

```
# Run 57
# record tcpflow log
```

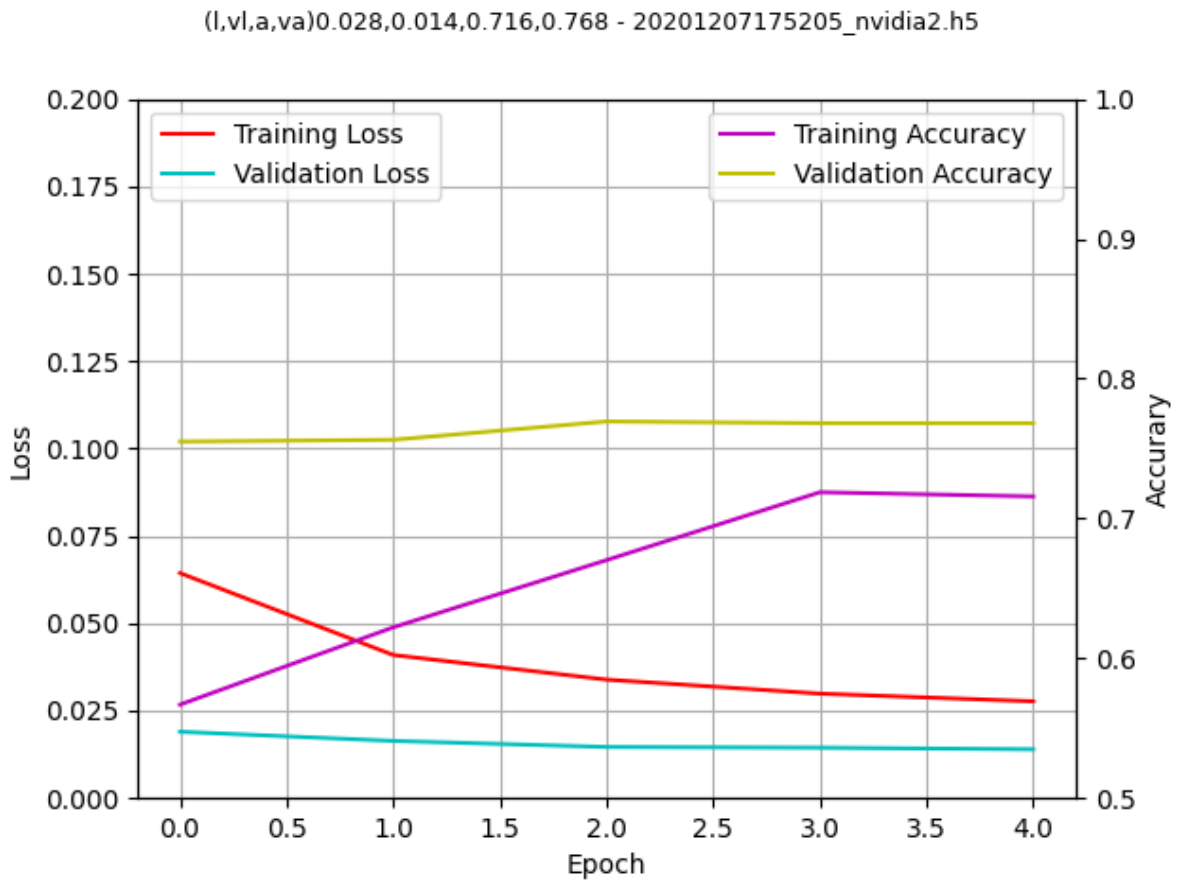

FIGURE E.21: nvidia2 20201207175205_ nvidia2.h5 (Run 57) model training accuracy for 5 epochs showing in the title the training loss (l), validation loss (vl), training accuracy (a) and validation accuracy (va) recorded for the last (5th - zero indexed) epoch (0.028, 0.014, 0.716 and 0.768, respectively.

```
$ sudo tcpflow -i lo -c port 9091 >
../trained_models/nvidia2/tcpflow/20201207175205_Rerun_57_nvidia2.log
# predict
$ python predict_client.py
--model=../trained_models/nvidia2/20201207175205_nvidia2.h5 --modelname=nvidia2
# make video - had to copy augmentation.py and conf.py to utils, where MakeVideo.py now
# lives
$ python MakeVideo.py
--filename=../../trained_models/nvidia2/tcpflow/20201207175205_Rerun_57_nvidia2.log
--model="20201207170938_nvidia2.h5"
```

This produced video `https://youtu.be/cUDiz7v_o3Y` which matches run 57. The result is mystifying, as it succeeds and precedes a succession of failed nvidia2 architecture runs until the (in hindsight) best "wet" performing model produced in Run 62 E.5.62). One possibility is that the last 10-unit hidden layer had been commented in for run 57, then commented out, with no commit to show the change. The 10-unit hidden layer made "a", perhaps "the" difference in run 62 E.5.62.

E.5.58 Run 58 - 20201207184329_ nvidia2.h5

```
Commit: 196f820
Model: nvidia2
Outputs: 2
Dataset: genTrack
Command:
python train.py
--model=nvidia2
--outdir=../trained_models
--epochs=5
--inputs=../dataset/unity/genTrack/*.jpg
--aug=True
--preproc=True
Environment: simbox
Comment: top crop set to 77.
Model drove off the road.

Inference time prediction:

$ python predict_client.py ../trained_models/nvidia2/20201207184329_nvidia2.h5
--modelname=nvidia2 --record=True --modelname=nvidia2
```

Top crop set to 77, drove off the road (`https://youtu.be/jHWX7bi5Okw`).

E.5.59 Run 58 - 20201207185525_ nvidia2.h5

```
Commit: 0cc41e7
Model: nvidia2
Outputs: 2
Dataset: genTrack
Command:
python train.py
--model=nvidia2
--outdir=../trained_models
--epochs=5
--inputs=../dataset/unity/genTrack/*.jpg
--aug=True
--preproc=True
Environment: simbox

Comment: top crop set to 83. Drove off the road
```

```
$ python predict_client.py
--model=../trained_models/nvidia2/20201207185525_nvidia2.h5 --modelname=nvidia2
--record=True --modelname=nvidia2
```

Video: `https://youtu.be/dkV0Tbfwr9Q`

E.5.60 Run 60 - 20201207190447_ nvidia2.h5

```
Commit: f1a5263
Model:  Same as 59
Outputs:  Same as 59
Dataset:  Same as 59
Command: Same as 59

Environment: simbox
Comment: Crash - another simulated car drives off the road.

Prediction:
$ python predict_client.py
--model=../trained_models/nvidia2/20201207190447_nvidia2.h5 --modelname=nvidia2
--record=True --modelname=nvidia2
```

E.5.61 Run 61 - 20201207192309_ nvidia2.h5

```
Commit: ae4742b
Model:  Same as 59
Outputs:  Same as 59
Dataset:  Same as 59
Command: Same as 59
Comment: Not zero centering pixel values, just normalizing. Crashed.

Investigating the actual changes for the nvidia2 model:

$ git diff --name-only f1a5263..ae4742b
src/conf.py
src/models.py

$ git diff f1a5263..ae4742b conf.py
diff --git a/src/conf.py b/src/conf.py
index e0efddd..d64e228 100644
```

```
--- a/src/conf.py
+++ b/src/conf.py
@@ -42,7 +42,7 @@ NVIDIA1 = 'nvidia1' # a.k.a. TawnNet
 nvidia1_img_dims = [160,120,3,160,120,60,-25]
 # NVIDIA2
 NVIDIA2 = 'nvidia2' # a.k.a. NaokiNet
-nvidia2_img_dims = [320,160,3,200,66,83,-35]
+nvidia2_img_dims = [320,160,3,200,66,81,-35]

$ git diff f1a5263..ae4742b models.py
diff --git a/src/models.py b/src/models.py
index adfada8..2539322 100644
--- a/src/models.py
+++ b/src/models.py
@@ -141,21 +141,21 @@ def nvidia_model2(num_outputs):
     row, col, ch = conf.nvidia2_img_dims[conf.IMG_HEIGHT_NET_IDX],
     conf.nvidia2_img_dims[conf.IMG_WIDTH_NET_IDX], \
                     conf.nvidia2_img_dims[conf.IMG_DEPTH_IDX]

-    drop = 0.5
+    drop = 0.1

     img_in = Input(shape=(row, col, ch), name='img_in')
     x = img_in
     # x = Cropping2D(cropping=((10,0), (0,0)))(x) #trim 10 pixels off top
-    x = Lambda(lambda x: x/127.5 - 1.0)(x) # normalize and re-center
-    # x = Lambda(lambda x: x / 255.0)(x)
+    # x = Lambda(lambda x: x/127.5 - 1.0)(x) # normalize and re-center
+    x = Lambda(lambda x: x / 255.0)(x)
     x = Conv2D(24, (5, 5), strides=(2, 2), activation='elu', name="conv2d_1")(x)
-    # x = Dropout(drop)(x)
-    x = Conv2D(36, (5, 5), strides=(2, 2), activation='elu', name="conv2d_2")(x)
-    # x = Dropout(drop)(x)
+    x = Dropout(drop)(x)
+    x = Conv2D(32, (5, 5), strides=(2, 2), activation='elu', name="conv2d_2")(x)
+    x = Dropout(drop)(x)
     x = Conv2D(48, (5, 5), strides=(2, 2), activation='elu', name="conv2d_3")(x)
-    # x = Dropout(drop)(x)
+    x = Dropout(drop)(x)
     x = Conv2D(64, (3, 3), activation='elu', name="conv2d_4")(x) # default strides=(1,1)
-    # x = Dropout(drop)(x)
+    x = Dropout(drop)(x)
```

```
    x = Conv2D(64, (3, 3), activation='elu', name="conv2d_5")(x)
    x = Dropout(drop)(x)
```

Top crop is set at 81, multiple dropout layers with dropout set at 0.1.

https://youtu.be/5VnNHD0wgFc

E.5.62 Run62 - 20201207192948_ nvidia2.h5

Commit: df953d2
Model: Same as 59
Outputs: Same as 59
Dataset: Same as 59
Command: Same as 59
Environment: same as 59
Comment: Restore last 10 unit fully connected layer. This one drives well around Generated Track. This last layer, perhaps together with non-zero centered pixel values and multi-dropout layers may have made the difference. Or perhaps just the final 10 hidden units? Will probably not have time to test so leaving as recommendation for future work i.e. do last ten hidden units before output neurons work with zero-centered normalised inputs? Note, answered this question in run 70.

```
$ cat nvidia2/20201207192948_nvidia2.log
Model name: ../trained_models/nvidia2/20201207192948_nvidia2.h5
Total training time: 0:01:39
Training loss: 0.023
Validation loss: 0.015
Training accuracy: 0.747
Validation accuracy: 0.773
```

To examine the model:

```
$ git checkout -b looksee df953d2
$ cat models.py
```

It shows the last ten unit hidden layers and the pixel values not being centered (run 70).

```
$ python predict_client.py
--model=../trained_models/nvidia2/20201207192948_nvidia2.h5 --modelname=nvidia2
--record=True --modelname=nvidia2
```

Video: `https://youtu.be/7-Paz1_PKAo`. Figure E.22 shows the training loss and accuracy plot for 20201207192948_ nvidia2.h5. The x axis divisions should be interpreted as 0.0 values obtained at the end of first training epoch. 1.0 as the values obtained at the end of second training epoch, and so forth.

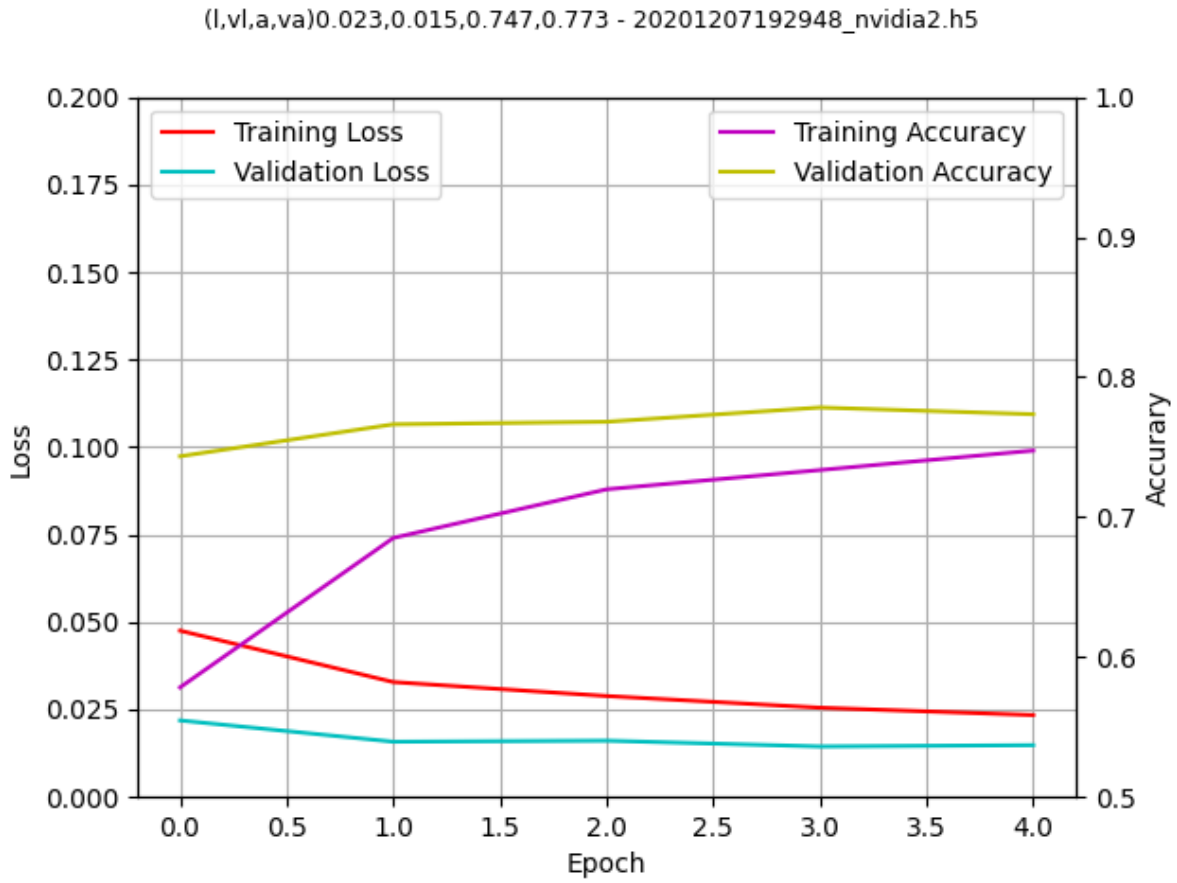

FIGURE E.22: 20201207192948_ nvidia2.h5 testing and validation loss and accuracy plots

E.5.63 Run 63 - 20201207193607_ nvidia2.h5

```
Commit: 3f65201
Model:  Same as 59
Outputs:  Same as 59
Dataset:  Same as 59
Command: Same as 59
Environment: same as 59
Comment: Only one dropout layer as per original NaokiNet model. Seemed to be
getting around the road ok, until it drove off the road. Otherwise, Very smooth.
Comparable to run 62.

$ python predict_client.py
```

```
--model=../trained_models/nvidia2/20201207193607_nvidia2.h5 --modelname=nvidia2
--record=True --modelname=nvidia2
```

Video: `https://youtu.be/p2Y6FZX4hNA`

E.5.64 Run 64 - 20201207194331_ nvidia2.h5

```
commit: 3cc0a1bfb8
Model:  Same as 59
Outputs:  Same as 59
Dataset:  Same as 59
Command: Same as 59
Environment: same as 59
Comment: No Dropout layers. This model although trained on Generated Track data,
drives well on Generated Road circuit.

$ python predict_client.py --model=../trained_models/nvidia2/20201207194331_nvidia2.h5
--modelname=nvidia2 --record=True --modelname=nvidia2

TODO - time allowing, run predictions on Generated Track and compare steering
for networks trained with none, one, and several dropout layers (runs 63, 64
and 49). Or move this paragraph to reflections section.
```

Video (`https://youtu.be/N9xyRizMJvI`) showing the first few hundred frames of model nvidia2 20201207194331_ nvidia2.h5 on the randomly Generated Road circuit.

Figure E.23 shows the training loss and accuracy plot for 20201207194331_ nvidia2.h.

E.5.65 Run 65 - 20201207194331_ nvidia2.h5

```
Comment: Duplicate entry, same as 64.
```

E.5.66 Run 66 - 20201207193607_ nvidia2.h5

```
Comment: Model 63 (only one dropout layer) on Generated Road. Also a bit wobbly but
manages to stay on the road for a while (drove off road in the end).
```

This is a duplicate entry to run 63 - video is `https://youtu.be/p2Y6FZX4hNA` The previously attributed video log `https://youtu.be/WA2uLtRn3Z0` refers to an incorrectly labelled upload. Another upload was made for run 66 `https://youtu.be/v34aJKjeiNs` which is in fact a duplicate of run 63, bar an extra second's duration in the video length. For previous edit (with

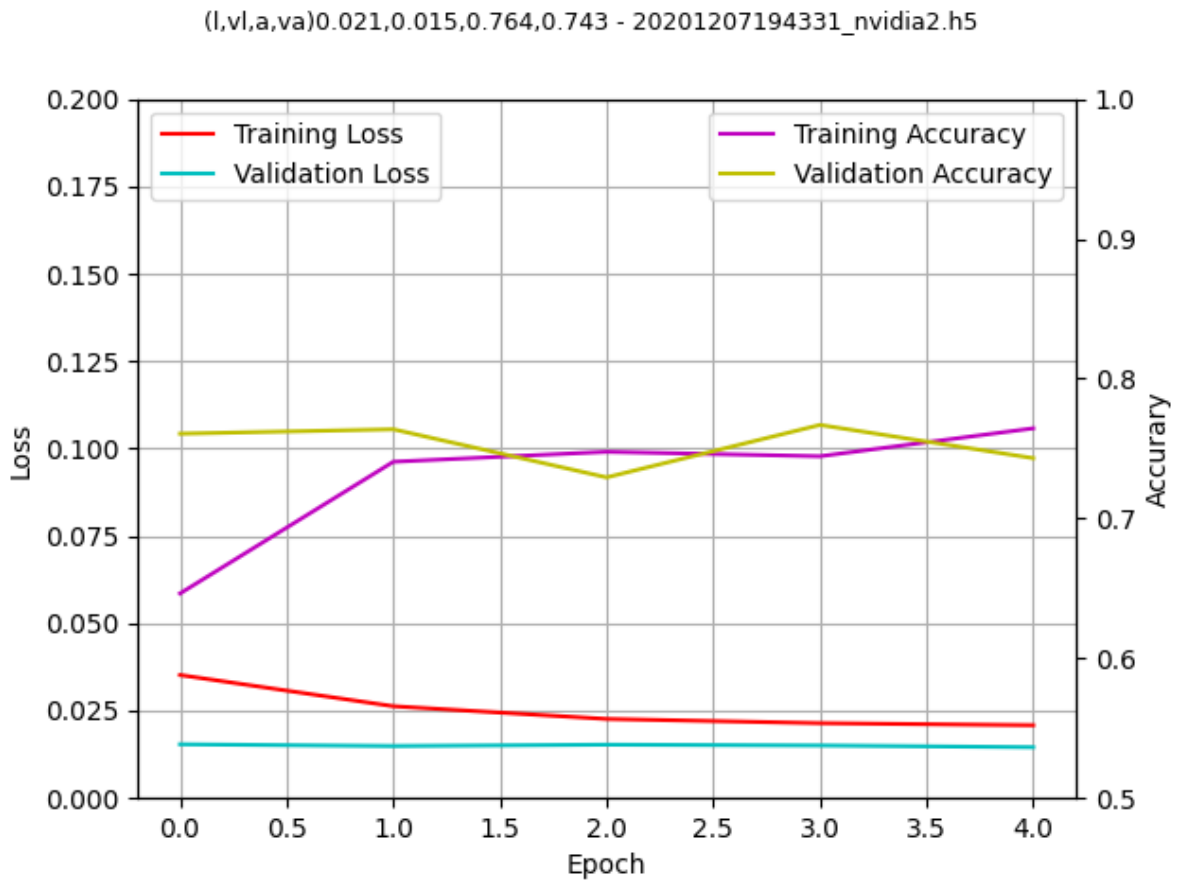

FIGURE E.23: 20201207194331_ nvidia2.h5 testing and validation loss and accuracy plots.

errors) see diff for this file between commit d68c4be5 and previous commit 2f93c98 on submitted repository msc-dissertation-latex.

E.5.67 Run 67 - 20201207195804_ nvidia2.h5

```
Commit: 957d14c
Model:  Same as 59
Outputs:  Same as 59
Dataset:  Same as 59
Command: Same as 59
Environment: same as 59
Comment: zero centered pixel values, drove off the road. This seems to be
the change that most impacts model performance, followed by image size.

$ git diff 3cc0a1bfb8..957d14c models.py
diff --git a/src/models.py b/src/models.py
index 7d4618e..fc143e2 100644
--- a/src/models.py
```

```
+++ b/src/models.py
@@ -146,8 +146,8 @@ def nvidia_model2(num_outputs):
     img_in = Input(shape=(row, col, ch), name='img_in')
     x = img_in
     # x = Cropping2D(cropping=((10,0), (0,0)))(x) #trim 10 pixels off top
-    # x = Lambda(lambda x: x/127.5 - 1.0)(x) # normalize and re-center
-    x = Lambda(lambda x: x / 255.0)(x)
+    x = Lambda(lambda x: x/127.5 - 1.0)(x) # normalize and re-center
+    # x = Lambda(lambda x: x / 255.0)(x)

This nvidia2 model does have the last dense(10) hidden layer:

$ git checkout -b looksee 957d14c
Switched to a new branch 'looksee'
$ cat models.py | grep Dense\(10
(...)
    x = Dense(10, activation='elu')(x) # Added in Naoki's model
(...)

Also worth noting, nvidia2 (NaokiNet) uses elu as activation function for all except
output layer.
Prediction:
$ python predict_client.py --model=../trained_models/nvidia2/20201207195804_nvidia2.h5
--modelname=nvidia2 --record=True --modelname=nvidia2
```

Video: `https://youtu.be/ZSgKMGL7azY`

Figure E.24 shows the training loss and accuracy plot for 20201207195804_ nvidia2.h.

E.5.68 Run 68 - 20201207201157_ nvidia_ baseline.h5

```
Commit: b1af57c
Model: nvidia_baseline
Outputs: 2
Dataset: genTrack
Command:
python train.py
--model=nvidia_baseline
--outdir=../trained_models
--epochs=5
--inputs=../dataset/unity/genTrack/*.jpg
--aug=True
```

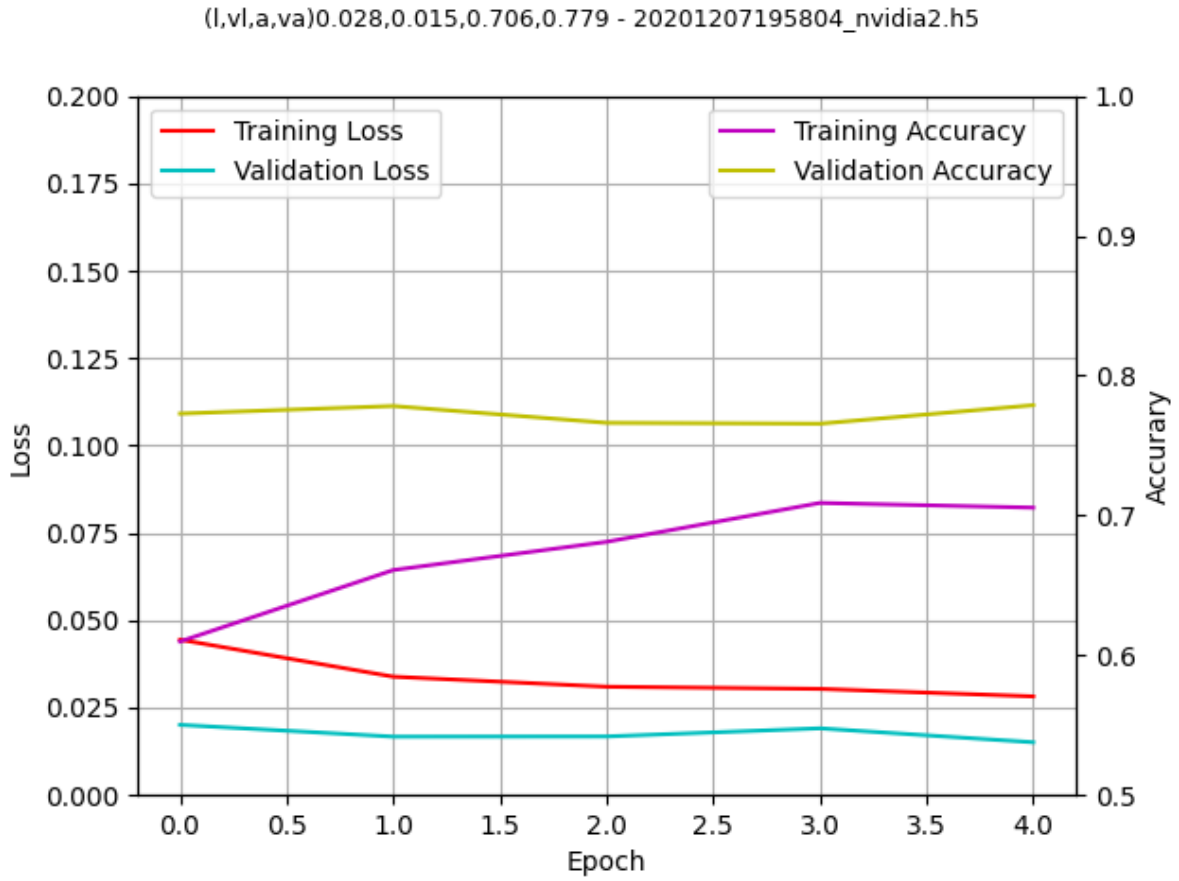

FIGURE E.24: 20201207195804_ nvidia2.h5 testing and validation loss and accuracy plots for run 67

```
--preproc=True
Environment: simbox
Comment: Drove off the road. Looks like geometry of input image must be known.
That is to say, the size of actual acquired image and consequentially, the size of top
and bottom crop, and the size of image (after being cropped) presented to network.
```

Video: `https://youtu.be/RxcC9tHCVUo` Note this model was actually trained with nvidia1 geometry.

E.5.69 Run 69 - 20201102090041_ nvidia2.h5

```
Commit: 4831a1b
Model: nvidia2
Outputs: 2
Dataset: genTrack
Command:
python train.py
--model=nvidia2
```

```
--outdir=../trained_models
--epochs=5
--inputs=../dataset/unity/genTrack/*.jpg
--aug=True
--preproc=True
Environment: simbox
Comment: set second convolutional layer to 32 feature maps failed to predict.

    ValueError: Input 0 of layer dense is incompatible with the layer: expected axis -1 of
    input shape to have value 6656 but received input with shape [None, 1152]

Was still zero centering (sanity checking)

$ python predict_client.py --model=../trained_models/nvidia2/20201102090041_nvidia2.h5
--record=True --modelname=nvidia2

Not sure what happened here. Investigating:

$ git diff --name-only b1af57c..4831a1b
src/conf.py
src/models.py

Changes related to nvidia_baseline.

Reran prediction:

$ python predict_client.py --model=../trained_models/nvidia2/20201102090041_nvidia2.h5
--record=True --modelname=nvidia2

Got error:
WARNING:tensorflow:Model was constructed with shape (None, 120, 160, 3) for input
Tensor("img_in:0", shape=(None, 120, 160, 3), dtype=float32), but it was called on an
input with incompatible shape (None, 66, 200, 3).

as 200x66 is the actual expected network size:
# in conf.py
# NVIDIA2
NVIDIA2 = 'nvidia2' # a.k.a. NaokiNet
nvidia2_img_dims = [320,160,3,200,66,81,-35]

There might have been an issue fixed with no commits to show the fix, changing the
prediction workflow, such that image is resized to expected network dimensions,
```

```
based on parameters in conf.py.
```

Figure E.25 shows the training loss and accuracy plot for 20201207192948_ nvidia2.h5.

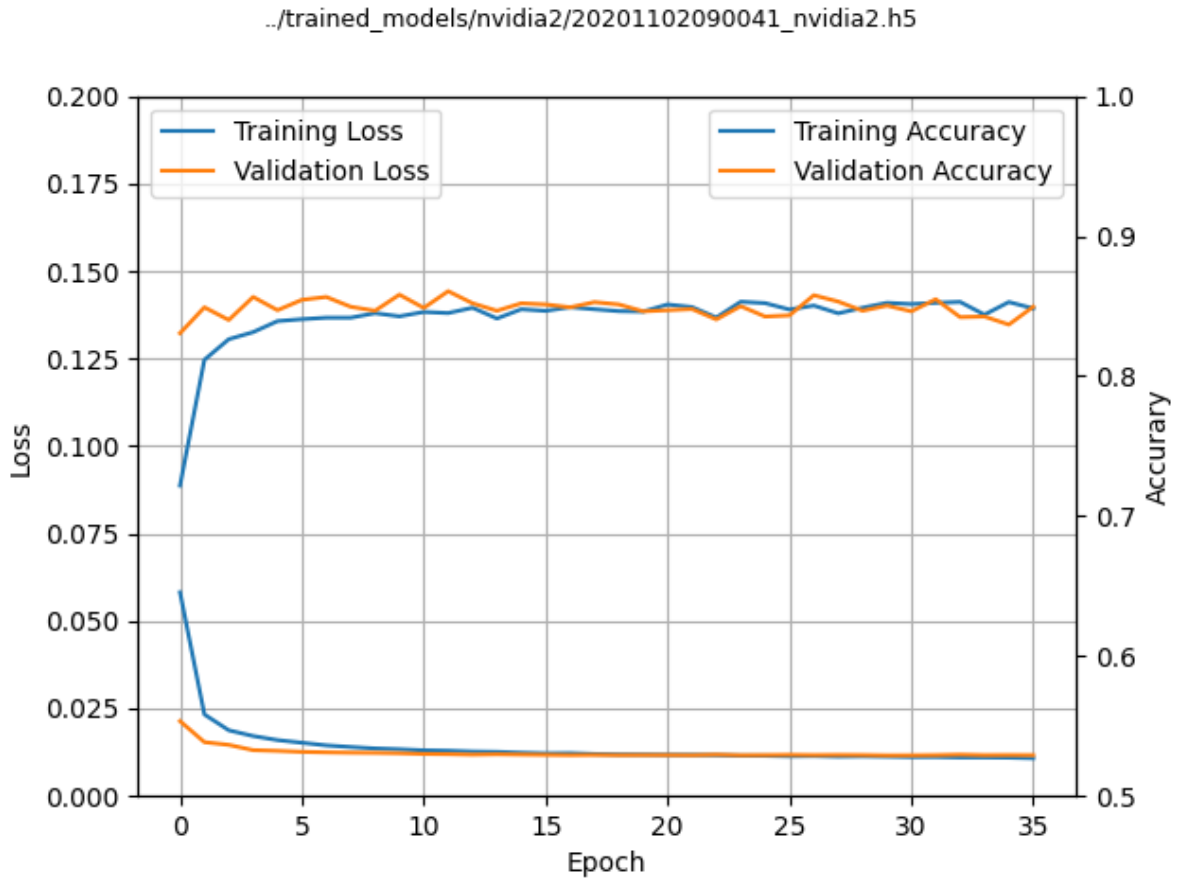

FIGURE E.25: 20201102090041_ nvidia2.h5 testing and validation loss and accuracy plots.

E.5.70 Run 70 - 20201207203451_ nvidia2.h5

```
Commit: 5e313d6
Model: nvidia2
Outputs: 2
Dataset: genTrack
Command:
python train.py
--model=nvidia2
--outdir=../trained_models
--epochs=5
--inputs=../dataset/unity/genTrack/*.jpg
--aug=True
--preproc=True
```

```
Environment: simbox
Comment: Weird how a change in feature map size changes model performance. It is still
going ok but not the best.

$ python predict_client.py --model=../trained_models/nvidia2/20201207203451_nvidia2.h5
--record=True --modelname=nvidia2

Video not uploaded, rerunning:

sudo tcpflow -i lo -c port 9091 >
../trained_models/nvidia2/tcpflow/20201207203451_nvidia2_rerun_70.log

$ python predict_client.py --model=../trained_models/nvidia2/20201207203451_nvidia2.h5
--modelname=nvidia2

$ python MakeVideo.py
--filename=../../trained_models/nvidia2/tcpflow/20201207203451_nvidia2_rerun_70.log
--model=20201120171015_sanity.h5

Actually, two things changed in code:

$ git diff 4831a1b..5e313d6 models.py
diff --git a/src/models.py b/src/models.py
index 6e82c2b..a749a2b 100644
--- a/src/models.py
+++ b/src/models.py
@@ -148,11 +148,11 @@ def nvidia_model2(num_outputs):
     img_in = Input(shape=(row, col, ch), name='img_in')
     x = img_in
     # x = Cropping2D(cropping=((10,0), (0,0)))(x) #trim 10 pixels off top
-    x = Lambda(lambda x: x/127.5 - 1.0)(x) # normalize and re-center
-    # x = Lambda(lambda x: x / 255.0)(x)
+    # x = Lambda(lambda x: x/127.5 - 1.0)(x) # normalize and re-center
+    x = Lambda(lambda x: x / 255.0)(x)
     x = Conv2D(24, (5, 5), strides=(2, 2), activation='elu', name="conv2d_1")(x)
     #x = Dropout(drop)(x)
-    x = Conv2D(36, (5, 5), strides=(2, 2), activation='elu', name="conv2d_2")(x)
+    x = Conv2D(32, (5, 5), strides=(2, 2), activation='elu', name="conv2d_2")(x)

Size of second set of feature maps changed from 36 to 32 (Naoki Shibuya's original
NaokiNet feature map size for second convolutional layer was 36.
```

https://github.com/naokishibuya/car-behavioral-cloning/blob/master/model.py
And data is being centered. This is significant. In run 62 the question was asked:
will a model with centered and normalized pixel values also perform well, when
the the "missing" NaokiNet original design 10-unit hidden layer had been added?
Then answer is, yes, but not as well as non-centered (run 62). The difference being
run 70 nearly drives off the road on first left turn, then rights itself and completes
the circuit on the opposite (left) lane, whereas run 62 completes the entire circuit
on the right hand lane.

Video: https://youtu.be/zaeQnImVTsk

E.5.71 Run 71 -

```
Commit: bc76eb3
Model: alexnet
Outputs: 2
Dataset: genTrack
Command:
python train.py
--model=alexnet
--outdir=../trained_models
--epochs=5
--inputs=../dataset/unity/genTrack/*.jpg
--aug=True
--preproc=True

Environment: simbox
Comment: Error - Input to reshape is a tensor with 495616 values, but the
requested shape requires a multiple of 6336. Need to sort out shapes.
```

E.5.72 Run 72 - 20201207091932_ nvidia1.h5

```
Commit: bc76eb3
Model: nvidia1
Outputs: 2
Dataset: genTrack
Command:
sudo tcpflow -i lo -c port 9091 >
/tmp/20201207091932_nvidia1_light_rain_mult_1_tcpflow.log

$ python predict_client.py
--model=../trained_models/nvidia1/20201207091932_nvidia1.h5 --modelname=nvidia1
```

```
--rain='light' --slant=0 --record=True

Environment: simbox
Comment: light rain. Drove off the road on next to last right turn.
```

Video: https://youtu.be/ulMMgQyMwNI

E.5.73 Run 73 - 20201207091932_ nvidia1.h5

```
Commit: bc76eb3
Model: nvidia1
Outputs: 2
Dataset: genTrack
Command:
sudo tcpflow -i lo -c port 9091 >
/tmp/20201207091932_nvidia1_heavy_10_mult_1_tcpflow.log

$ python predict_client.py
--model=../trained_models/nvidia1/20201207091932_nvidia1.h5 --modelname=nvidia1
--rain='heavy' --slant=10 --record=True

Environment: simbox
Comment: heavy rain 10 degree slant. Drove off the road on same spot as run
72.
```

Video: https://youtu.be/Yff1s3RYFSg

E.5.74 Run 74 - 20201207091932_ nvidia1.h5

```
Commit:
Model:
Outputs:
Dataset:
Command:
sudo tcpflow -i lo -c port 9091 >
/tmp/20201207091932_nvidia1_torrential_20__mult_1_tcpflow.log

$ python predict_client.py
--model=../trained_models/nvidia1/20201207091932_nvidia1.h5 --modelname=nvidia1
--rain='torrential' --slant=20 --record=True

Environment: simbox
```

```
Comment: Torrential rain, 20 degree slant. Drove off the road on same spot as
runs 72 and 73.
```

Video: https://youtu.be/Hn4fXr1a89I. Sky was not changed for runs 72, 73 and 74, nvidia1, intensity multiplie (sic) 1. As the car drove off the road on all 3 runs, this is assumed to not have made much difference in model performance. Also, sky is changed for a more realistic effect of rain. The actual brighter multiplies in following experiments were suggested by supervisor as a means of recreating the effect of reflected light on wet roads.

E.5.75 Run 75 - 20201207091932_ nvidia1.h5

```
Commit: 4d2c67e
Model: nvidia1
Outputs: 2
Dataset: genTrack
Command:
sudo tcpflow -i lo -c port 9091 > /tmp/20201207091932_nvidia1_light_rain_mult_4_tcpflow.log

$ python predict_client.py --model=../trained_models/nvidia1/20201207091932_nvidia1.h5
--modelname=nvidia1 --rain='light' --slant=0 --record=True

Environment: simbox
Comment: Light rain, intensity multiply 4. Drove off the road on first right turn.
```

Video: https://youtu.be/qdTA5ho5VOE

E.5.76 Run 76 - 20201207091932_ nvidia1.h5

```
Commit: 4d2c67e
Model: nvidia1
Outputs: 2
Dataset: genTrack
Command:
sudo tcpflow -i lo -c port 9091 > /tmp/20201207091932_nvidia1_heavy_10_mult_4_tcpflow.log

$ python predict_client.py --model=../trained_models/nvidia1/20201207091932_nvidia1.h5
--modelname=nvidia1 --rain='heavy' --slant=10 --record=True

Environment: simbox
Comment: Heavy rain, slant -+10, intensity multiply 4. Drove off the road on same
spot as run 75 and hit the side of bollard.
```

Video: https://youtu.be/sKyoke3IO84

E.5.77 Run 77 - 20201207091932_ nvidia1.h5

```
Commit: 4d2c67e
Model: nvidia1
Outputs: 2
Dataset: genTrack
Command:
sudo tcpflow -i lo -c port 9091 >
/tmp/20201207091932_nvidia1_torrential_20_mult_4_tcpflow.log

$ python predict_client.py --model=../trained_models/nvidia1/20201207091932_nvidia1.h5
--modelname=nvidia1 --rain='torrential' --slant=20 --record=True

Environment: simbox
Comment: Torrential rain, slant -+20, intensity multiply 4. Same as 76, and
crashed straight into the bollard.
```

Video: https://youtu.be/mDjtnnVZdic

E.5.78 Run 78 - 20201207091932_ nvidia1.h5

```
Commit: 8d3745c
Outputs: 2
Dataset: genTrack
Command:
sudo tcpflow -i lo -c port 9091 >
/tmp/20201207091932_nvidia1_light_rain_mult_8_tcpflow.log

$ python predict_client.py --model=../trained_models/nvidia1/20201207091932_nvidia1.h5
--modelname=nvidia1 --rain='light' --slant=0 --record=True

Environment: simbox
Comment: Light rain, intensity multiply 8. Managed to get around, swapping lanes.
```

Video: https://youtu.be/31Dt8RafE8o

E.5.79 Run 79 - 20201207091932_ nvidia1.h5

```
Commit: 8d3745c
Outputs: 2
Dataset: genTrack
Command:
```

```
sudo tcpflow -i lo -c port 9091 > /tmp/20201207091932_nvidia1_heavy_10_mult_8_tcpflow.log

$ python predict_client.py --model=../trained_models/nvidia1/20201207091932_nvidia1.h5
--modelname=nvidia1 --rain='heavy' --slant=20 --record=True

Environment: simbox
Comment: Heavy rain, +-10 degree slant, intensity multiply 8. Managed to get around,
swapping lanes.
```

Video: https://youtu.be/RwWftJeJagY

E.5.80 Run 80 - 20201207091932_ nvidia1.h5

```
Commit: 8d3745c
Outputs: 2
Dataset: genTrack
Command:
sudo tcpflow -i lo -c port 9091 >
/tmp/20201207091932_nvidia1_torrential_20_mult_8_tcpflow.log

$ python predict_client.py --model=../trained_models/nvidia1/20201207091932_nvidia1.h5
--modelname=nvidia1 --rain='torrential' --slant=20 --record=True

Environment: simbox
Comment: Torrential rain, +-20 degree slant, intensity multiply 8. Managed to get
around, swapping lanes.
```

Video https://youtu.be/ta40jlqdZ04.

E.5.81 Run 81 - 20201207091932_ nvidia1.h5

```
Commit: 356fe8f
Outputs: 2
Dataset: genTrack
Command:
sudo tcpflow -i lo -c port 9091 > /tmp/20201207091932_nvidia1_no_rain_tcpflow.log

$ python predict_client.py
--model=../trained_models/nvidia1/20201207091932_nvidia1.h5 --modelname=nvidia1
--record=True
Environment: simbox
Comment: Run to create a tcpflow a no rain file (best steering lap for nvidia1
```

```
$ python MakeVideo.py --filename=../../trained_models/nvidia1/tcpflow/20201207091932_nvidia1_
tcpflow.log
--model=20201207091932_nvidia1.h5
```

For run 81, no recording was done as the interest was in the tcpflow data for the best nvidia1 model. CORRENTION. Recording now 2020.12.18 done. Video: `https://youtu.be/Tzo3zw-b_qk`

E.5.82 Run 82 - 20201207192948_ nvidia2.h5

```
Commit: 156e1a3f3bb007cbf7090402a9c276c6a9cc835a
Model: nvidia2
Outputs: 2
Dataset: genTrack
Command:
python predict_client.py
--model=../trained_models/nvidia2/20201207192948_nvidia2.h5 --modelname=nvidia2
--record=True
Environment: simbox
Comment: No rain lap for best nvidia2 model
```

Video: `https://youtu.be/iePt2HJPXP0`. For nvidia2, a recording was done, which matches the "line", that is the path, taken on run 62, as it is the same model.

E.5.83 Run 83 - 20201207192948_ nvidia2.h5

nvidia2 actually managed to get around

```
Commit: 156e1a3f3bb007cbf7090402a9c276c6a9cc835a
Model: nvidia2
Outputs: 2
Dataset: genTrack
Command:
$ sudo tcpflow -i lo -c port 9091 >
/tmp/20201207192948_nvidia2_light_rain_mult_1_tcpflow.log

$ python predict_client.py
--model=../trained_models/nvidia2/20201207192948_nvidia2.h5 --modelname=nvidia2
--record=True --rain='light' slant=0
```

```
Environment: simbox
Comment: light rain lap for best nvidia2 model
```

Video: `https://youtu.be/d-yY9M2fO4U`. The nvidia2 model managed to drive around the track (nvidia 1 crashed). The model however does change lanes.

E.5.84 Run 84 - 20201207192948_ nvidia2.h5

nvidia2 actually managed to get around again, practically same path

```
Commit: 156e1a3f3bb007cbf7090402a9c276c6a9cc835a
Model: nvidia2
Outputs: 2
Dataset: genTrack
Command:
$ sudo tcpflow -i lo -c port 9091 >
/tmp/20201207192948_nvidia2_heavy_10_mult_1_tcpflow.log

$ python predict_client.py
--model=../trained_models/nvidia2/20201207192948_nvidia2.h5 --modelname=nvidia2
--record=True --rain='heavy' slant=10
Environment: simbox
Comment: Heavy rain slant 10 lap for best nvidia2 model
```

Video: `https://youtu.be/3AhcVqdV6KM`

E.5.85 Run 85 - 20201207192948_ nvidia2.h5

```
Commit: 156e1a3f3bb007cbf7090402a9c276c6a9cc835a
Model: nvidia2
Outputs: 2
Dataset: genTrack
Command:
$ sudo tcpflow -i lo -c port 9091 >
/tmp/20201207192948_nvidia2_torrential_20_mult_1_tcpflow.log

$ python predict_client.py
--model=../trained_models/nvidia2/20201207192948_nvidia2.h5 --modelname=nvidia2
--record=True --rain='torrential' slant=20
Environment: simbox
Comment: Torrential rain slant 20 lap for best nvidia2 model
```

Video: `https://youtu.be/zk6XalCoOrw`. Runs 83, 84 and 85 (mult 1) take identical lines with same lane changes during the lap.

E.5.86 Run 86 - 20201207192948_ nvidia2.h5

Same batch of 3 with multiplier set to 4

```
Commit: c501da2
Model: nvidia2
Outputs: 2
Dataset: genTrack
Command:
$ sudo tcpflow -i lo -c port 9091 >
/tmp/20201207192948_nvidia2_light_rain_mult_4_tcpflow.log

$ python predict_client.py
--model=../trained_models/nvidia2/20201207192948_nvidia2.h5 --modelname=nvidia2
--record=True --rain='light' slant=0
Environment: simbox
Comment: light rain lap mult 4 for best nvidia2 model
```

Video: `https://youtu.be/nmD8wnVtdoo`

E.5.87 Run 87 - 20201207192948_ nvidia2.h5

Same batch of 3 with multiplier set to 4

```
Commit: c501da2
Model: nvidia2
Outputs: 2
Dataset: genTrack
Command:
$ sudo tcpflow -i lo -c port 9091 >
/tmp/20201207192948_nvidia2_heavy_10_mult_4_tcpflow.log

$ python predict_client.py
--model=../trained_models/nvidia2/20201207192948_nvidia2.h5 --modelname=nvidia2
--record=True --rain='heavy' slant=10
Environment: simbox
Comment: heavy rain slant 10 mult 4
```

Video: `https://youtu.be/LTTNYyDFFDk`

E.5.88 Run 88 - 20201207192948_ nvidia2.h5

Same batch of 3 with multiplier set to 4

```
Commit: c501da2
Model: nvidia2
Outputs: 2
Dataset: genTrack
Command:
$ sudo tcpflow -i lo -c port 9091 >
/tmp/20201207192948_nvidia2_torrential_20_mult_4_tcpflow.log

$ python predict_client.py
--model=../trained_models/nvidia2/20201207192948_nvidia2.h5 --modelname=nvidia2
--record=True --rain='torrential' slant=20
Environment: simbox
Comment: torrential rain slant 20 mult 4
```

Video: `https://youtu.be/jBB4q1JK3oA`

E.5.89 Run 89 - 20201207192948_ nvidia2.h5

Same batch of 3 with multiplier set to 8

```
Commit: fbd3448
Model: nvidia2
Outputs: 2
Dataset: genTrack
Command:
$ sudo tcpflow -i lo -c port 9091 >
/tmp/20201207192948_nvidia2_light_rain_mult_8_tcpflow.log

$ python predict_client.py
--model=../trained_models/nvidia2/20201207192948_nvidia2.h5 --modelname=nvidia2
--record=True --rain='light' slant=0
Environment: simbox
Comment: light rain slant 0 mult 8
```

Video: `https://youtu.be/qY8eL-9V-hM`

Figure E.26 shows 3 stills taken for corresponding videos of network nvidia2 20201207192948_ nvidia2.h5 (best performing model in the rain), for runs 83 (E.5.83), 86 (E.5.86) and 89 (E.5.89), containing from left to right, the acquired image as produced by the Unity3D game engine (SDSandbox), the same image with added light rain and the processed image as presented to

the network, which then predicts a steering angle. The first row for intensity multiply 1, the second for 4 and the third for 8. There are no lane markings in the last row, suggesting the network has learned about road boundaries and is able to generalise, given the image being presented (no lane markings) had never been seen before.

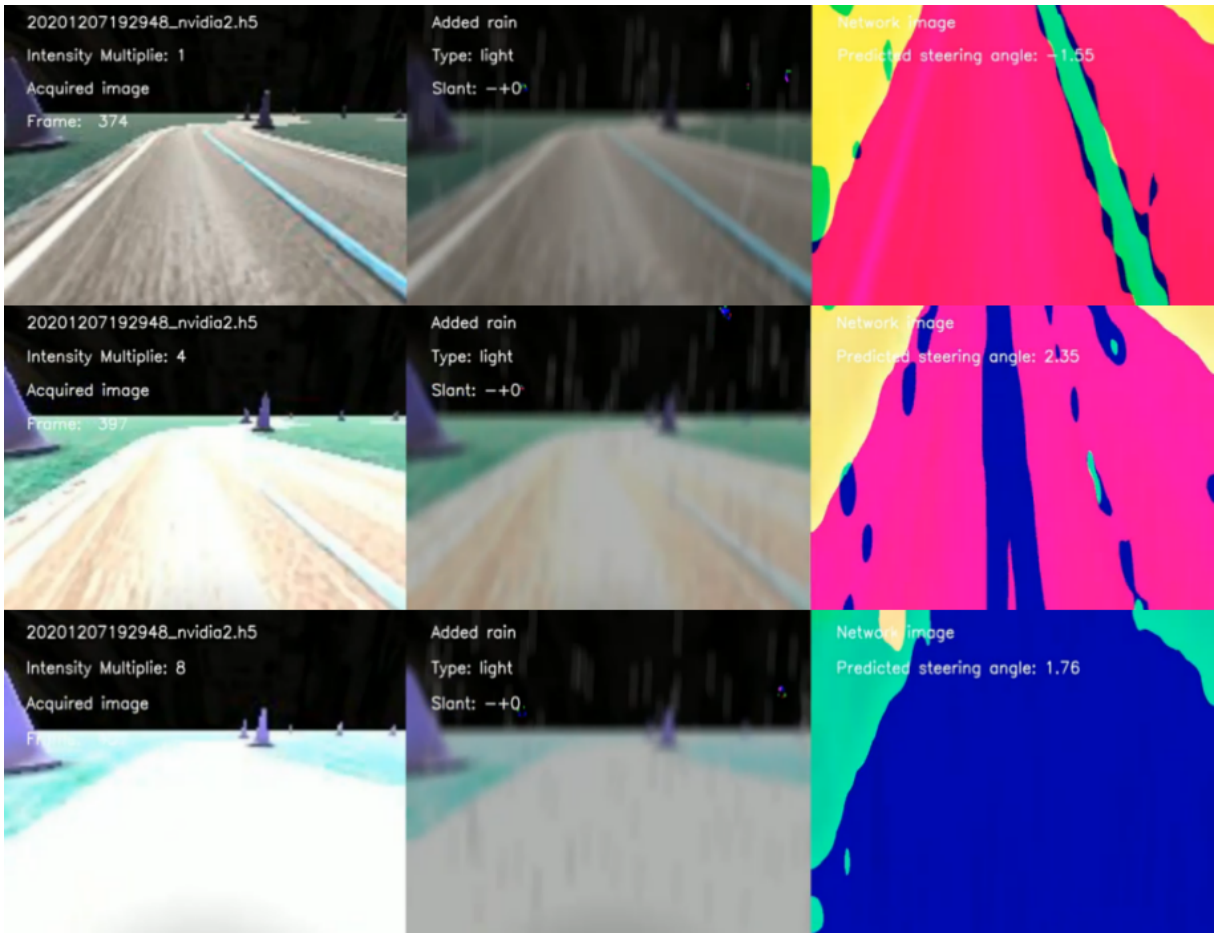

FIGURE E.26: Best performing 20201207192948_ nvidia2 model video stills, rows from top to bottom generated in runs 83 (E.5.83), 86 (E.5.86) and 89 (E.5.89. The data generated in each run corresponding to orange lines in Figures 4.20, 4.21 and 4.22.

E.5.90 Run 90 - 20201207192948_ nvidia2.h5

Same batch of 3 with multiplier set to 8

```
Commit: fbd3448
Model: nvidia2
Outputs: 2
Dataset: genTrack
Command:
$ sudo tcpflow -i lo -c port 9091 >
/tmp/20201207192948_nvidia2_heavy_10_mult_8_tcpflow.log
```

```
$ python predict_client.py
--model=../trained_models/nvidia2/20201207192948_nvidia2.h5 --modelname=nvidia2
--record=True --rain='heavy' slant=10
Environment: simbox
Comment: heavy rain slant 10 mult 8
```

Video: `https://youtu.be/1PpfHLwOx9M`

Figure E.27 shows 3 stills taken for corresponding videos of network nvidia2 20201207192948_ nvidia2.h5 (best performing model in the rain), for runs 84 (E.5.84), 87 (E.5.87) and 90 (E.5.90), with heavy rain added.

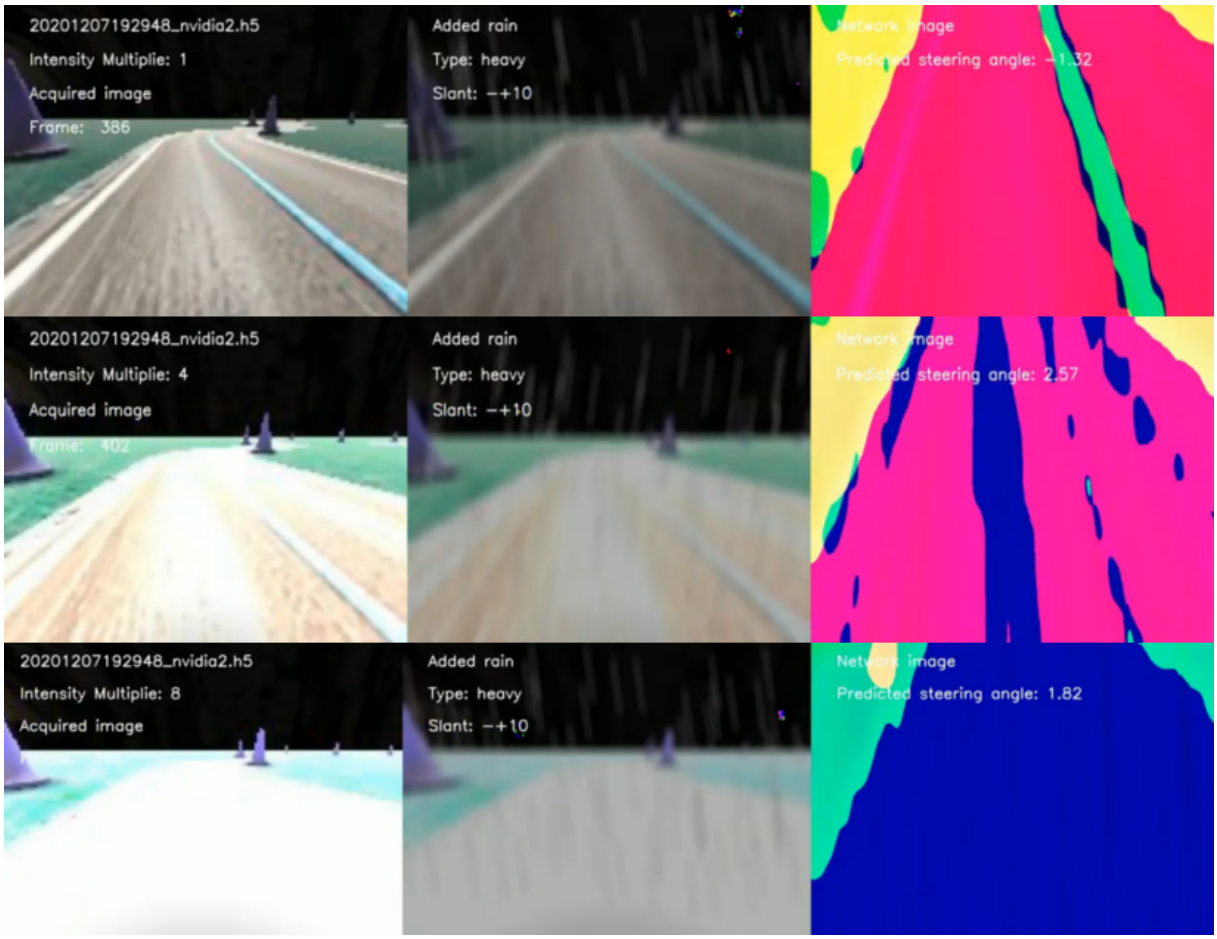

FIGURE E.27: Intensity multipliers 1, 4 and 8 for heavy rain

E.5.91 Run 91 - 20201207192948_ nvidia2.h5

Same batch of 3 with multiplier set to 8

```
Commit: fbd3448
Model: nvidia2
Outputs: 2
Dataset: genTrack
```

```
Command:
$ sudo tcpflow -i lo -c port 9091 >
/tmp/20201207192948_nvidia2_torrential_20_mult_8_tcpflow.log

$ python predict_client.py
--model=../trained_models/nvidia2/20201207192948_nvidia2.h5 --modelname=nvidia2
--record=True --rain='torrential' slant=20
Environment: simbox
Comment: Torrential rain slant 20 mult 8
```

Video: `https://youtu.be/W1eRN5DWPXw`

Figure E.28 shows 3 stills taken for corresponding videos of network nvidia2 20201207192948_ nvidia2.h5 (best performing model in the rain), for runs 85 (E.5.85), 88 (E.5.88) and 91 (E.5.91), with torrential rain added.

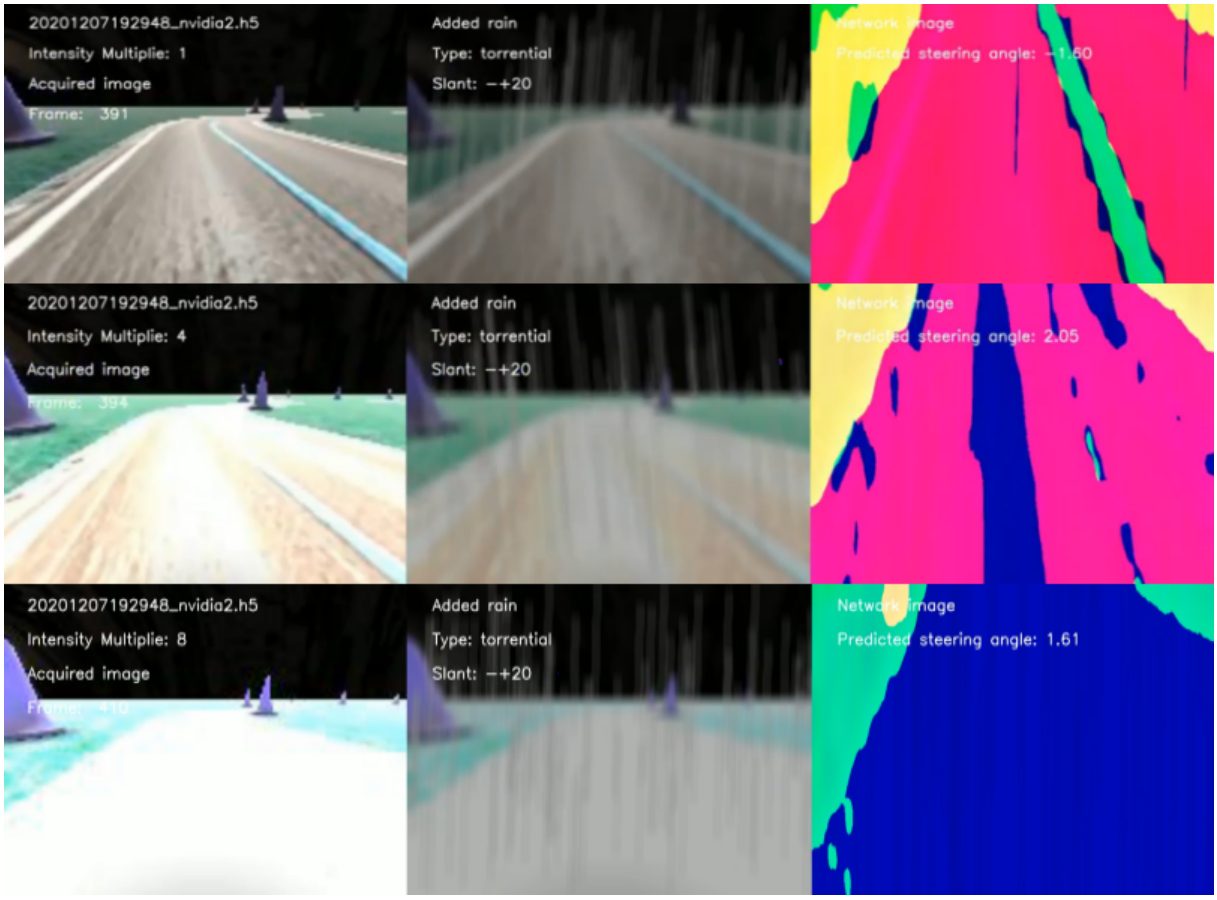

FIGURE E.28: Intensity multipliers 1, 4 and 8 for torrential rain

E.5.92 Run 92 - 20201209001926_ nvidia1.h5

```
Commit: 20201209001926_nvidia1.h5
```

```
Model: nvidia1
Outputs: 2
Dataset: genRoad
Command:
python train.py
--model=nvidia1
--outdir=../trained_models
--epochs=100
--inputs=../dataset/unity/genRoad/*.jpg
--aug=True
--preproc=True

Epoch 42/100
3229/3229 [==== (...)

Total training time: 6:35:44
Training loss: nan
Validation loss: nan
Training accuracy: 0.206
Validation accuracy: 0.204

Environment: simbox
Comment: Very wobbly on the road

python predict_client.py
--model=../trained_models/nvidia1/20201209001926_nvidia1.h5 --modelname=nvidia1
```

Starts well then wobbles and goes off road (`https://youtu.be/K5FmPq0_0dE`)

```
Epoch 4/5
3232/3232 [==============================] - 560s 173ms/step - loss: 0.0142 -
acc: 0.8396 - val_loss: 0.0106 - val_acc: 0.8559
Epoch 5/5
3232/3232 [==============================] - 560s 173ms/step - loss: 0.0136 -
acc: 0.8422 - val_loss: 0.0101 - val_acc: 0.8494
```

E.5.93 Run 93 - 20201209221402_ nvidia1.h5

```
Commit: 173d74b
Model: nvidia1
Outputs: 2
```

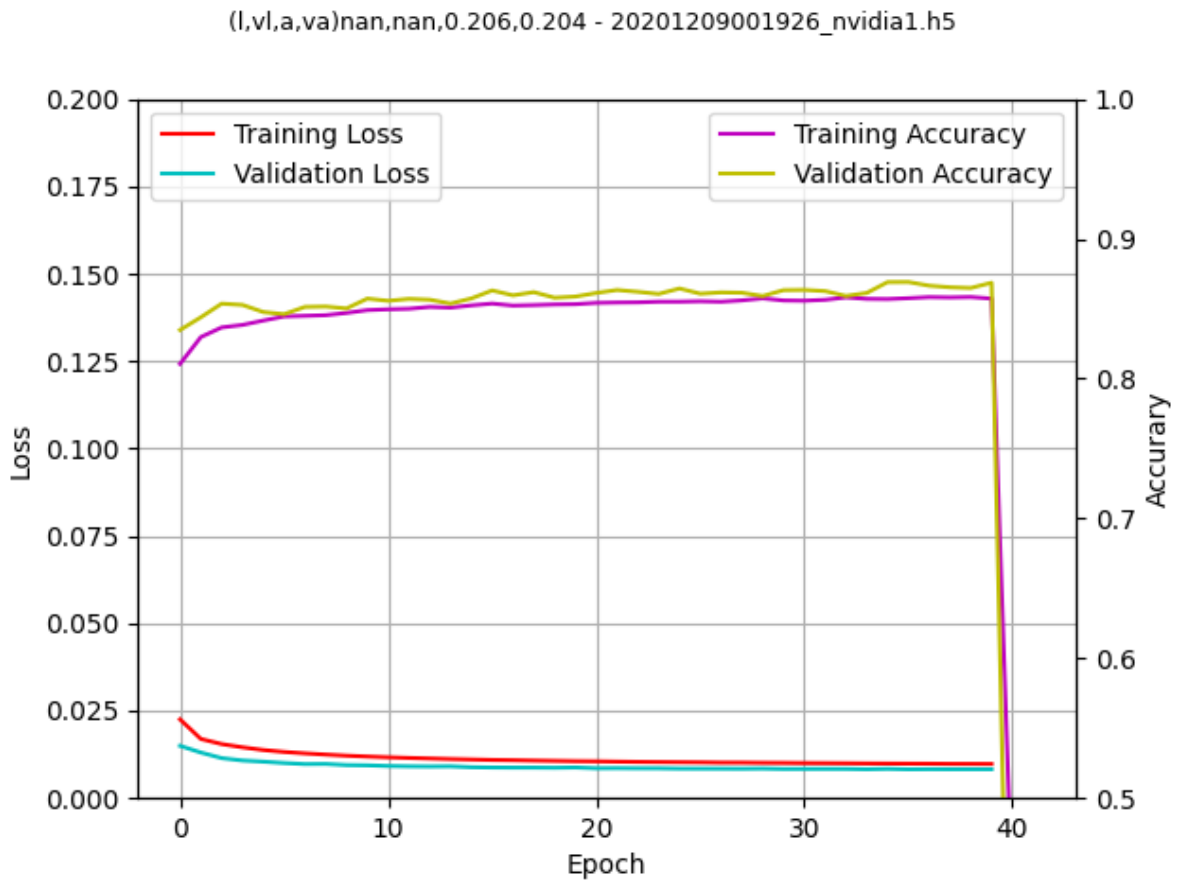

FIGURE E.29: nvidia1 trained on genRoad data for 100 epochs - stopped at 42 (best model saved)

```
Dataset: genRoad
Command:
python train.py
--model=nvidia1
--outdir=../trained_models
--epochs=5
--inputs=../dataset/unity/genRoad/*.jpg
--aug=True
--preproc=True
Environment: simbox
Comment: Same results training for 5 epochs (run 93) and 100 epochs (run 92).

sudo tcpflow -i lo -c port 9091 >
../trained_models/nvidia1/tcpflow/20201209221402_nvidia1_genRoad_dry.log

python3 predict_client.py
--model=../trained_models/nvidia1/20201209221402_nvidia1.h5 --record=True
```

Video: `https://youtu.be/L6I84lpoAyI`. Starts straight, then wobbles. Figure E.30 shows the erratic steering starting after frame 100.

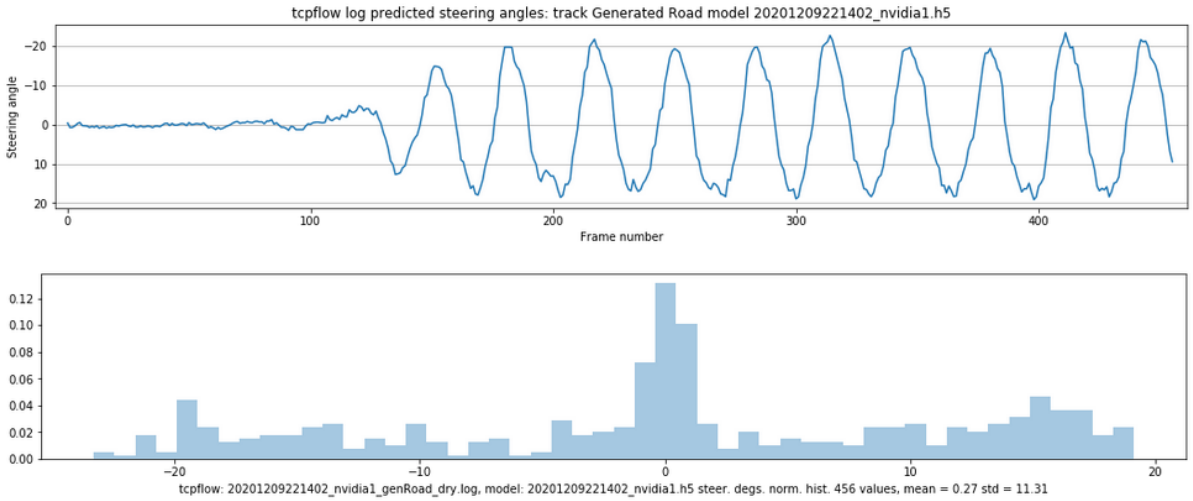

FIGURE E.30: $20201209221402_nvidia1.h5 trained on genRoad data for 5 epochs - same results as trained with 100 epochs$

E.5.94 Run 94 - 20201207192948_ nvidia2.h5

```
Commit: b0b10e3
Model: nvidia2
Outputs: 2
Dataset: genTrack
Command: Same as 59
Environment: simbox
Comment: Running a prediction on genRoad data with model trained on genTrack data.
Drives quite smoothly. Recommendation for future work: investigate why this is the
case, model trained on genTrack drives well on genRoad.

sudo tcpflow -i lo -c port 9091 >
../trained_models/nvidia2/tcpflow/20201207192948_nvidia2_dry_genRoad.log

python3 predict_client.py --model=../trained_models/nvidia2/20201207192948_nvidia2.h5
--record=True --modelname=nvidia2
```

Video: `https://youtu.be/EhTCJ_e-RpM`.

Figure E.31 shows the smoother steering starting centered around 0 degrees, as well as as more even spread in the histogram.

Figure E.32 shows a randomly Generated Road used for runs 93, 94 and 95, where the first few hundred frames go to near the top left of the circuit (long right 180 degree turn) shown

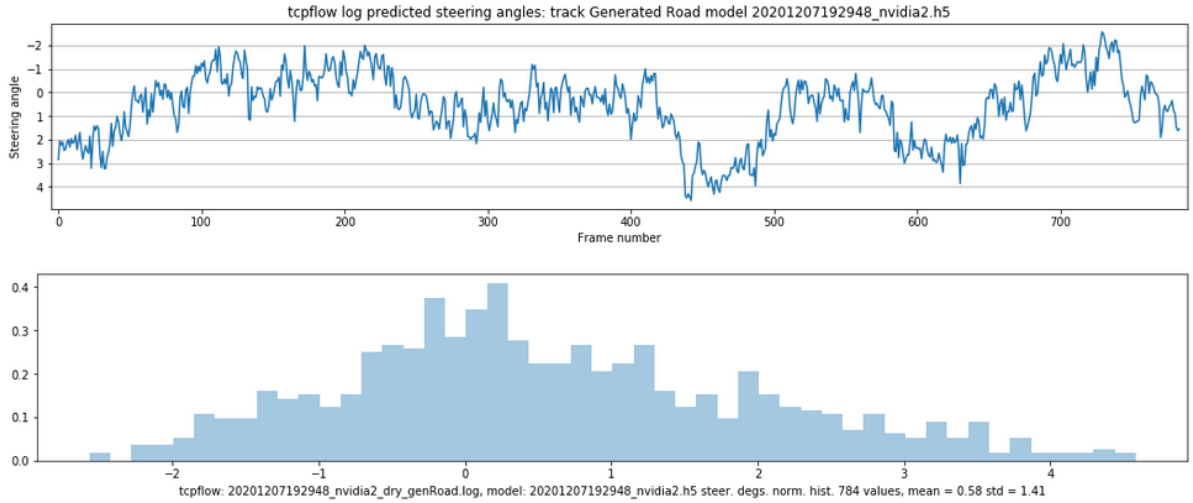

FIGURE E.31: nvidia2 trained on genTrack dataset (from the Generated Track circuit) driving well on the randomly generated Generated Road circuit shown in figure E.32

in detail on the right hand image. The predicted steering angles shown in Figure E.31 for the well performing nvidia2 20201207192948_ nvidia2.h5 model, show the majority of values in the positive range, as the simulated vehicle follows the road and takes a right turn to do so.

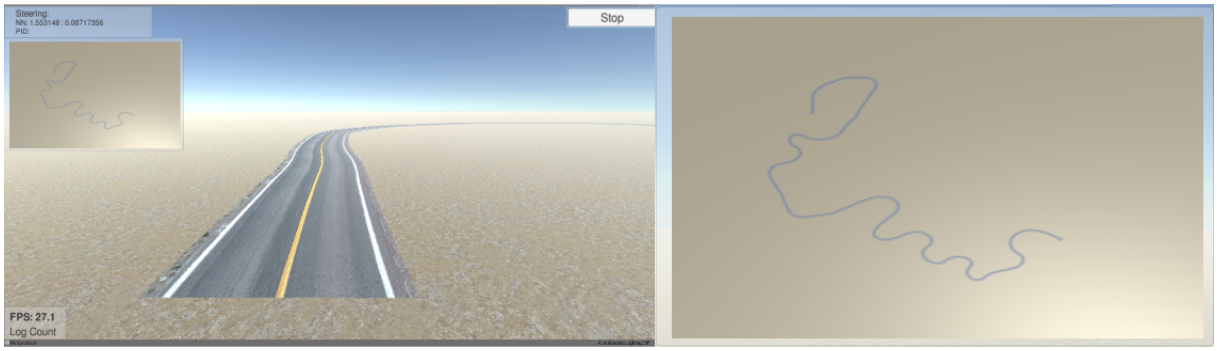

FIGURE E.32: The randomly generated Generated Road circuit used in runs 93 and 94. Left image is a view of the simulator as presented on computer desktop, the right image is the augmented detail showing the Generated Road circuit, the same inset on left image.

E.5.95 Run 95 - 20201207192948_ nvidia2.h5

```
Commit: c4158dd
Model: nvidia2
Outputs: 2
Dataset: genTrack
Command: same as 59
Environment: simbox
Comment: One of the best nvidia2 models self-driving the entirety of a randomly
generated road.
```

Video: `https://youtu.be/z9nILq9dQfI`.

Figure E.33 showing steering angles and normalized histogram plots. Run 95 (16m14s duration) was much longer than run 94 (1m11s duration) which is apparent in the histogram distribution, as more data is present (about 800 data points in run 94 and 11000 data points in run 95), so is the distribution smoother.

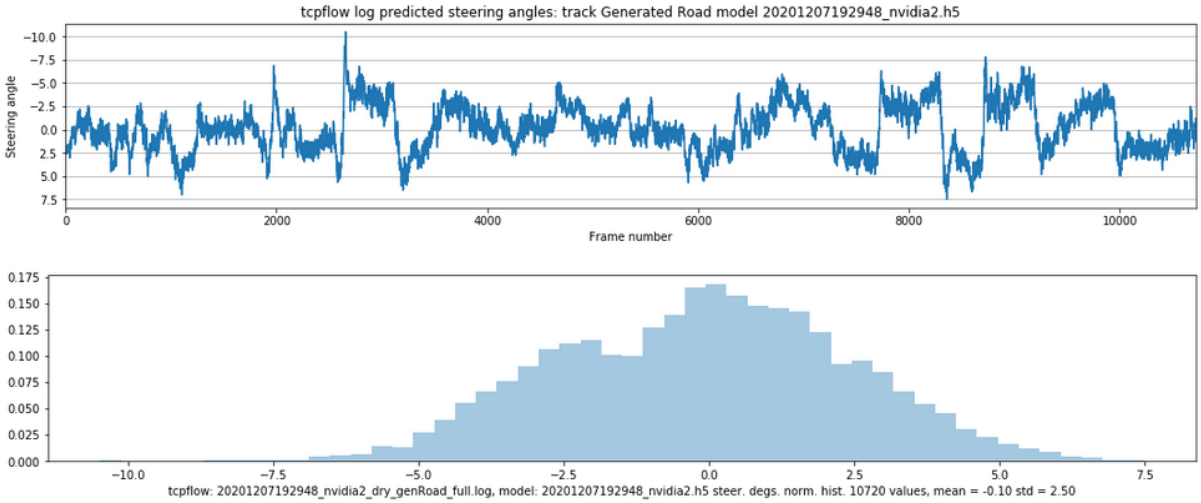

FIGURE E.33: Steering angle and normalized histogram plots of the 20201207192948_ nvidia2.h5 model driving the full length of Generated Track presented in figure E.32

E.5.96 Run 96 - Two models on Generated Road

```
Commit: ae0afcc1c1
Model: 20201120171015_sanity.h5 and 20201107210627_nvidia1.h5
Outputs: 2
Dataset: genTrack
Command:
$ python predict_client.py --model=../trained_models/sanity/20201120171015_sanity.h5
$ python predict_client.py --model=../trained_models/nvidia1/20201107210627_nvidia1.h5
Environment: Simbox
Comment: Recorded with Kazam
```

Video: `https://youtu.be/v0Otdmtdhnk`

E.5.97 Run 97 - Four models on Generated Track

```
Commit: ae0afcc1c1
Model: 20201207192948_nvidia2.h5 and 20201207091932_nvidia1.h5
```

```
Outputs: 2
Dataset: genTrack
Command:
$ python predict_client.py
--model=../trained_models/nvidia2/20201207192948_nvidia2.h5 --modelname=nvidia2
$ python predict_client.py
--model=../trained_models/nvidia2/20201207192948_nvidia2.h5 --modelname=nvidia2
--rain=light --slant=0
$ python predict_client.py
--model=../trained_models/nvidia1/20201207091932_nvidia1.h5 --modelname=nvidia1
--rain=light --slant=0
$ python predict_client.py
--model=../trained_models/nvidia2/20201207192948_nvidia2.h5 --modelname=nvidia2
--rain=torrential --slant=20
Environment: Simbox
Comment: Best nvidia1 (x1) and nvidia2 (x3) driving around Generated Track,
with different levels of rain (not shown on video). Recorded with Kazam.
```

Figure E.34 shows a still from video `https://youtu.be/ayESXH9zZdM` record with Kazam. Bottom left is 20201207192948_ nvidia2.h5 with no rain, bottom right 20201207192948_ nvidia2.h5 light rain zero slant. Top left is 20201207091932_ nvidia1.h5 light rain zero slant , top right is 20201207192948_ nvidia2.h5 torrential rain -+20 degree slant.

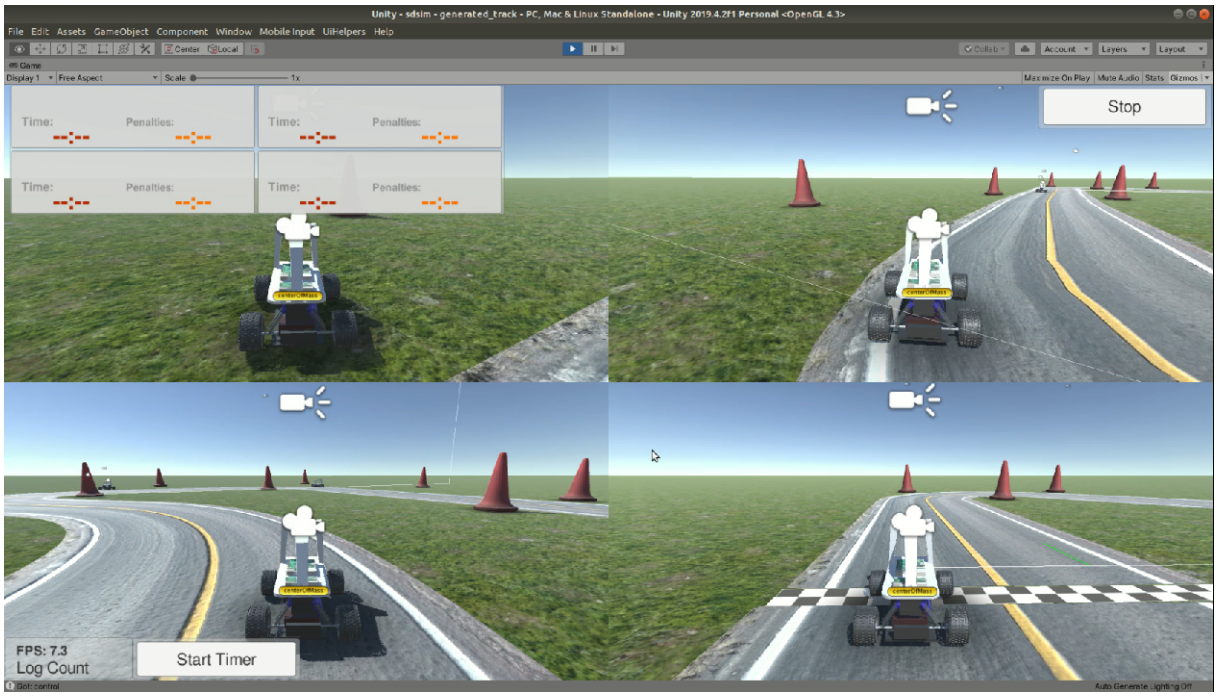

FIGURE E.34: Video still from Kazam recording for four-way SDSandbox prediction for nvidia2 with no rain (bottom left), nvidia2 with light rain (bottom right), nvidia1 with light rain (top left, crash on third right turn) and nvidia2 torrential rain (top right)

E.5.98 Run 98 - 20201121090912_ nvidia_ baseline.h5

```
Commit: Not recorded
Model: nvidia_baseline
Outputs: 2
Dataset: not recorded
Command: not recorded
Environment: camber
Comment: Added after the training run for Figure reference. This model was trained
on camber on November 21

Training log:

Model name: ../trained_models//nvidia_baseline/20201121090912_nvidia_baseline.h5
Total training time: 0:05:25
Training loss: 0.066
Validation loss: 0.064
Training accuracy: 0.524
Validation accuracy: 0.556

Unusable model due to weights not being loaded on local workstation due to Python/
Keras/Tensorflow version incompatibility.
```

E.6 Outputs

This section is a partial list models, videos and tcpflow log outputs generated for this project.

E.6.1 Models

Keras .h5 pickled models (73 total) save in /trained_ models directory:

```
# To obtain the list:
/trained_models(master)$ ls -R | grep .h5 > models.txt
The string between "_" and ".h5" indicates the sub-directory of /trained_models where
model is stored, e.g. 20201124030257_nvidia1.h5 is stored in /trained_models/nvidia1,
20201102090041_nvidia2.h5 is stored in /trained_models/nvidia2.

20201124030257_nvidia1.h5
20201124032017_nvidia2.h5
20201031222825_nvidia.h5
20201101164532_nvidia.h5
20201102070829_nvidia.h5
```

20201102075100_nvidia1.h5
20201102081239_nvidia1.h5
20201102094552_nvidia1.h5
20201105230035_nvidia1.h5
20201107144927_nvidia1.h5
20201107164328_nvidia1.h5
20201107164514_nvidia1.h5
20201107181406_nvidia1.h5
20201107210627_nvidia1.h5
20201117160222_nvidia1.h5
20201117161503_nvidia1.h5
20201117162041_nvidia1.h5
20201117165411_nvidia1.h5
20201117173543_nvidia1.h5
20201117175030_nvidia1.h5
20201123225559_nvidia1.h5
20201124235331_nvidia1.h5
20201206171648_nvidia1.h5
20201206211122_nvidia1.h5
20201207091932_nvidia1.h5
20201209001926_nvidia1.h5
20201209221402_nvidia1.h5
20201116234135_nvidia1_jungle.h5
20201102090041_nvidia2.h5
20201102134802_nvidia2.h5
20201102210514_nvidia2.h5
20201103211330_nvidia2.h5
20201207111940_nvidia2.h5
20201207124146_nvidia2.h5
20201207132429_nvidia2.h5
20201207133600_nvidia2.h5
20201207141017_nvidia2.h5
20201207142129_nvidia2.h5
20201207170336_nvidia2.h5
20201207170938_nvidia2.h5
20201207173730_nvidia2.h5
20201207175205_nvidia2.h5
20201207184329_nvidia2.h5
20201207185525_nvidia2.h5
20201207190447_nvidia2.h5
20201207191850_nvidia2.h5
20201207192309_nvidia2.h5

```
20201207192948_nvidia2.h5
20201207193607_nvidia2.h5
20201207194331_nvidia2.h5
20201207195804_nvidia2.h5
20201207202449_nvidia2.h5
20201207202950_nvidia2.h5
20201207203451_nvidia2.h5
20201207205741_nvidia2.h5
20201116222734_nvidia_baseline.h5
20201116223255_nvidia_baseline.h5
20201116223556_nvidia_baseline.h5
20201117131144_nvidia_baseline.h5
20201117132314_nvidia_baseline.h5
20201117134127_nvidia_baseline.h5
20201117134837_nvidia_baseline.h5
20201117145744_nvidia_baseline.h5
20201117151149_nvidia_baseline.h5
20201117151825_nvidia_baseline.h5
20201117154210_nvidia_baseline.h5
20201117162326_nvidia_baseline.h5
20201117174303_nvidia_baseline.h5
20201120124421_nvidia_baseline.h5
20201207201157_nvidia_baseline.h5
20201120171015_sanity.h5
20201120184912_sanity.h5
20201123162643_sanity.h5
```

E.6.2 YouTube videos

List of 61 videos uploaded to youtube:

```
# Obtained exporting playlist
https://www.youtube.com/playlist?list=PLHdK4Nj2PBN7cGyPO_kNSp5nmdMbGk68j
# with
https://www.tunemymusic.com/YouTube-to-File.php
# csv list then parsed with utils/parse_playlist.py
$ parse_playlist.py > playlist.txt
# then list was edited to remove deleted videos entries (labelled as such).
# A google search for each string returns the video.

Rerun 70 - 20201207203451 nvidia2 rerun 70
Run 95 - 20201207192948 nvidia2 dry genRoad full
```

Rerun 57 - 20201207175205 nvidia2 run 57 tcpflow sanity check
Run 56 sanity check - 20201207170938 nvidia2 Run 56 tcpflow sanity check
Run 72 - 20201207091932 nvidia1 light rain mult 1 h5
Run 63 - 20201207193607 nvidia2 h5 - only one dropout layers
Run 64 - 20201207194331 nvidia2 h5 - no dropout layers
Run 94 - 20201207192948 nvidia2 dry genRoad (model trained on genTrack)
Run 93 - 20201209221402 nvidia1 genRoad dry h5 - wobbly
Run 92 - 20201209001926 nvidia1 100 epochs genRoad h5
Run 91 - 20201207192948_nvidia2.h5 torrential rain +-20 degree slant intensity multiplie 8
Run 90 - 20201207192948 nvidia2 heavy 10 mult 8 h5
Run 89 - 20201207192948 nvidia2 light rain mult 8 h5
Run 87 - 20201207192948 nvidia2 heavy 10 mult 4 h5
Run 88 - 20201207192948 nvidia2 torrential 20 mult 4 h5
Run 85 - 20201207192948 nvidia2 torrential 20 mult 1 h5
Run 86 - 20201207192948 nvidia2 light rain mult 4 h5
Run 84 - 20201207192948 nvidia2 heavy 10 mult 1 h5
Run 83 - 20201207192948 nvidia2 light rain mult 1 h5
Run 82 - 20201207192948 nvidia2 no rain h5
Run 80 - 20201207091932 nvidia1 torrential 20 mult 8 h5
Run 78 20201207091932_nvidia1_light_rain_mult_8.h5
Run 79 20201207091932 nvidia1 heavy 10 mult 8 h5
Run 74 - 20201207091932 nvidia1 torrential 20 mult 1 h5
Run 73 20201207091932 nvidia1 heavy 10 mult 1 h5
Run 77 20201207091932_nvidia1_torrential_20_mult_4.h5
Run 76 20201207091932 nvidia1 heavy 10 mult 4 h5
Run 75 - 20201207091932 nvidia1 light rain mult 4 h5
Run 66 (route 66 ?) 20201207193607 nvidia2 h5 model trained on GenTrack driving on GenRoad
Run 62 20201207192948 nvidia2 h5 restored last 10 units fully connected layer
Run 68 20201207201157 nvidia baseline.h5 - drove off the road
Run 67 20201207195804 nvidia2 h5
Run 57 20201207175205 nvidia2 h5
Run 66 - model 20201207193607 nvidia2 h5 (one dropout) on Generated Road (INCORRECT UPLOAD)
Run 61 20201207191850 nvidia2 h5 - not zero centering pixel values or cropping off the top.
Run 58 20201207184329 nvidia2 h5
Run 59 20201207185525 nvidia2 h5
Run 56 20201207170938 nvidia2 h5
"Run 55 20201207142129 nvidia2 h5
Run 54 20201207141017 nvidia2 h5

```
Run 53 20201207133600 nvidia2 h5 same as 52 with last 10 unit layer removed
Run 52 20201207132429 nvidia2 h5 dropout set to 0.1
Run 51 20201207124146 nvidia2 h5 trained with 0.25 droupout on single layer
20201207111940 nvidia2 h5 nvidia2 (NaokiNet) trained with a single 0.5 dropout layer
goes off road
20201207091932 nvidia1 h5 - trained over 5 epochs in 1m32s
"Run 45 - 20201120171015_ sanity.h5 in heavy rain
"Run 44 - 20201120171015_ sanity.h5 in heavy rain
Run 43 - 20201120171015_ sanity.h5 in torrential rain drives off the road
20201120171015_sanity.h5 simulator and network images
"SDSandbox ""Drive w Rec"" mode"
20201124032017_nvidia2.h5 trained on Intel DevCloud
Model 20201123162643\_sanity.h5 - predictions run with image preprocessing
20201123162643_sanity.h5 keras/tensorflow model
20201120184912_sanity.h5 keras/tensorflow model - oversteering (Outliers? Batch Size?)
Outliers!
20201120171015_sanity.h5
20201120171015 sanity h5 - Sanity check
Tensorflow AV x Unity
20201107210627_nvidia1.h5 keras/tensorflow model
20201120124421 nvidia baseline model predictions
20201102094552 nvidia1 h5 run
20201107210627 nvidia1 h5 run
```

E.6.3 tcpflow logs

List of 30 tcpflow logs recorded for models nvidia1 and nvidia2.

```
# obtained by listing all tcpflow logs
/trained_models(master)$ ls nvidia1/tcpflow > tcpflow_logs.txt
/trained_models(master)$ ls nvidia2/tcpflow >> tcpflow_logs.txt

20201207091932_nvidia1_heavy_10__mult_1_tcpflow.log
20201207091932_nvidia1_heavy_10_mult_4_tcpflow.log
20201207091932_nvidia1_heavy_10_mult_8_tcpflow.log
20201207091932_nvidia1_light_rain_mult_1_tcpflow.log
20201207091932_nvidia1_light_rain_mult_4_tcpflow.log
20201207091932_nvidia1_light_rain_mult_8_tcpflow.log
20201207091932_nvidia1_no_rain_tcpflow.log
20201207091932_nvidia1_tcpflow.log
20201207091932_nvidia1_torrential_20__mult_1_tcpflow.log
20201207091932_nvidia1_torrential_20_mult_4_tcpflow.log
```

20201207091932_nvidia1_torrential_20_mult_8_tcpflow.log
20201209221402_nvidia1_genRoad_dry.log
20201207111940_nvidia2_tcpflow.log
20201207124146_nvidia2_tcpflow.log
20201207132429_nvidia2_tcpflow.log
20201207170938_Rerun_56_nvidia2.log
20201207175205_Rerun_57_nvidia2.log
20201207192948_nvidia2_dry_genRoad_full.log
20201207192948_nvidia2_dry_genRoad.log
20201207192948_nvidia2_heavy_10_mult_1_tcpflow.log
20201207192948_nvidia2_heavy_10_mult_4_tcpflow.log
20201207192948_nvidia2_heavy_10_mult_8_tcpflow.log
20201207192948_nvidia2_light_rain_mult_1_tcpflow.log
20201207192948_nvidia2_light_rain_mult_4_tcpflow.log
20201207192948_nvidia2_light_rain_mult_8_tcpflow.log
20201207192948_nvidia2_no_rain_tcpflow.log
20201207192948_nvidia2_torrential_20_mult_1_tcpflow.log
20201207192948_nvidia2_torrential_20_mult_4_tcpflow.log
20201207192948_nvidia2_torrential_20_mult_8_tcpflow.log
20201207203451_nvidia2_rerun_70.log

E.7 Download links

This section contain download links for data made available to examiners.

- Dataset - `https://bit.ly/3alDWvs`
- Source code - `https://bit.ly/2KwnNsl`
- Trained models - `https://bit.ly/34raiBh`

F Source Code Listing

This appendix contains the original and modified-from-original source code listings used in this project.

Listing of original (and were noted modified from other source) code used in this project

```
#####################################################
# 1. augmentation.py
# Note: code modified from original in
# https://github.com/naokishibuya/car-behavioral-cloning/blob/master/utils.py
# Available for audit in audit_files/naoki from sharepoint link
#####################################################


import cv2, os
import numpy as np
import matplotlib.image as mpimg
import conf

IMAGE_HEIGHT, IMAGE_WIDTH, IMAGE_CHANNELS = conf.image_height, conf.image_width, conf.image_
depth
INPUT_SHAPE = (IMAGE_HEIGHT, IMAGE_WIDTH, IMAGE_CHANNELS)
IMAGE_HEIGHT_NET, IMAGE_WIDTH_NET =  conf.image_height_net, conf.image_width_net

def load_image(image_path):
    """
    Load RGB images from a file
    """
    return mpimg.imread(image_path)


def crop(image):
    """
    Crop the image (removing the sky at the top and the car front at the bottom)
    """
    # this breaks nvidia_baseline
    return image[60:-25, :, :] # remove the sky and the car front


def resize(image):
    """
    Resize the image to the input shape used by the network model
    """
    return cv2.resize(image, (IMAGE_WIDTH_NET, IMAGE_HEIGHT_NET), cv2.INTER_AREA)


def rgb2yuv(image):
    """
    Convert the image from RGB to YUV (This is what the NVIDIA model does)
    """
    return cv2.cvtColor(image, cv2.COLOR_RGB2YUV)


def preprocess(image):
    """
    Combine all preprocess functions into one
    """
    image = crop(image)
```

```
    image = resize(image)
    image = rgb2yuv(image)
    return image

'''
# only one camera in our case
def choose_image(data_dir, center, left, right, steering_angle):
    """
    Randomly choose an image from the center, left or right, and adjust
    the steering angle.
    """
    choice = np.random.choice(3)
    if choice == 0:
        return load_image(data_dir, left), steering_angle + 0.2
    elif choice == 1:
        return load_image(data_dir, right), steering_angle - 0.2
    return load_image(data_dir, center), steering_angle
'''


def random_flip(image, steering_angle):
    """
    Randomly flipt the image left <-> right, and adjust the steering angle.
    """
    if np.random.rand() < 0.5:
        image = cv2.flip(image, 1)
        steering_angle = -steering_angle
    return image, steering_angle


def random_translate(image, steering_angle, range_x, range_y):
    """
    Randomly shift the image vertically and horizontally (translation).
    """
    trans_x = range_x * (np.random.rand() - 0.5)
    trans_y = range_y * (np.random.rand() - 0.5)
    steering_angle += trans_x * 0.002
    trans_m = np.float32([[1, 0, trans_x], [0, 1, trans_y]])
    height, width = image.shape[:2]
    image = cv2.warpAffine(image, trans_m, (width, height))
    return image, steering_angle


def random_shadow(image):
    """
    Generates and adds random shadow
    """
    # (x1, y1) and (x2, y2) forms a line
    # xm, ym gives all the locations of the image
    x1, y1 = IMAGE_WIDTH * np.random.rand(), 0
    x2, y2 = IMAGE_WIDTH * np.random.rand(), IMAGE_HEIGHT
    xm, ym = np.mgrid[0:IMAGE_HEIGHT, 0:IMAGE_WIDTH]

    # mathematically speaking, we want to set 1 below the line and zero otherwise
    # Our coordinate is up side down.  So, the above the line:
```

```
    # (ym-y1)/(xm-x1) > (y2-y1)/(x2-x1)
    # as x2 == x1 causes zero-division problem, we'll write it in the below form:
    # (ym-y1)*(x2-x1) - (y2-y1)*(xm-x1) > 0
    mask = np.zeros_like(image[:, :, 1])
    mask[(ym - y1) * (x2 - x1) - (y2 - y1) * (xm - x1) > 0] = 1

    # choose which side should have shadow and adjust saturation
    cond = mask == np.random.randint(2)
    s_ratio = np.random.uniform(low=0.2, high=0.5)

    # adjust Saturation in HLS(Hue, Light, Saturation)
    hls = cv2.cvtColor(image, cv2.COLOR_RGB2HLS)
    hls[:, :, 1][cond] = hls[:, :, 1][cond] * s_ratio
    return cv2.cvtColor(hls, cv2.COLOR_HLS2RGB)


def random_brightness(image):
    """
    Randomly adjust brightness of the image.
    """
    # HSV (Hue, Saturation, Value) is also called HSB ('B' for Brightness).
    hsv = cv2.cvtColor(image, cv2.COLOR_RGB2HSV)
    ratio = 1.0 + 0.4 * (np.random.rand() - 0.5)
    hsv[:,:,2] =  hsv[:,:,2] * ratio
    return cv2.cvtColor(hsv, cv2.COLOR_HSV2RGB)

def augment(image, steering_angle, range_x=100, range_y=10):
    """
    Generate an augumented image and adjust steering angle.
    (The steering angle is associated with the center image)
    """
    # resize according to expected input shape e.g. AlexNet 224x224, Udacity 320x160, Unity 160x120, etc
    # set in conf.py
    image = cv2.resize(image, (IMAGE_WIDTH, IMAGE_HEIGHT), cv2.INTER_AREA)
    #image, steering_angle = choose_image(data_dir, center, left, right, steering_angle)
    image, steering_angle = random_flip(image, steering_angle)
    image, steering_angle = random_translate(image, steering_angle, range_x, range_y)
    image = random_shadow(image)
    image = random_brightness(image)
    return image, steering_angle


######################################################
# 2. Augmentation.py
# Note: code modified from original in
# https://github.com/naokishibuya/car-behavioral-cloning/blob/master/utils.py
# Available for audit in audit_files/naoki from sharepoint link
######################################################


import cv2, os
import numpy as np
import matplotlib.image as mpimg
import conf
```

```
class Augmentation():
    """
    Augmentation methods
    """
    img_dims = []

    def __init__(self, model):
        """
        Set image dimensions for model
        Inputs
            model: string, model name
        """
        self.img_dims = self.get_image_dimensions(model)

    def get_image_dimensions(self, model):
        """
        Get the required dimensions for image model, used for resizing and cropping images
        Inputs
            model: string, name of network model
        Output
            int: IMAGE_WIDTH, IMAGE_HEIGHT, IMAGE_WIDTH_NET, IMAGE_HEIGHT_NET, TOP_CROP,
BOTTOM_CROP
        """
        if (model == conf.ALEXNET):
            return conf.alexnet_img_dims
        elif (model == conf.NVIDIA1):
            return conf.nvidia1_img_dims
        elif (model == conf.NVIDIA2):
            return conf.nvidia2_img_dims
        elif (model == conf.NVIDIA_BASELINE):
            return conf.nvidia_baseline_img_dims
        else:
            # default to nvidia1
            return conf.nvidia1_img_dims

    def load_image(self, image_path):
        """
        Load RGB images from a file
        """
        return mpimg.imread(image_path)

    def crop(self, image):
        """
        Crop the image (removing the sky at the top and the car front at the bottom)
        """
        # this breaks nvidia_baseline
        # return image[60:-25, :, :] # remove the sky and the car front
        return image[self.img_dims[conf.IMG_TOP_CROP_IDX]:self.img_dims[conf.IMG_BOTTOM_CROP_IDX],
:,
              :]  # remove the sky and the car front

    def resize(self, image):
```

```
        """
        Resize the image to the input shape used by the network model
        """
        return cv2.resize(image, (self.img_dims[conf.IMG_WIDTH_NET_IDX], self.img_dims[conf.IMG_HEIGHT_
NET_IDX]),
                          cv2.INTER_AREA)

    def resize_expected(self, image):
        """
        Resize the image to the expected original shape
        """
        return cv2.resize(image, (self.img_dims[conf.IMG_WIDTH_IDX], self.img_dims[conf.IMG_HEIGHT_IDX]),
 cv2.INTER_AREA)

    def rgb2yuv(self, image):
        """
        Convert the image from RGB to YUV (This is what the NVIDIA model does)
        """
        return cv2.cvtColor(image, cv2.COLOR_RGB2YUV)

    def preprocess(self, image):
        """
        Combine all preprocess functions into one
        """
        image = self.crop(image)
        image = self.resize(image)
        image = self.rgb2yuv(image)
        return image

    def random_flip(self, image, steering_angle):
        """
        Randomly flipt the image left <-> right, and adjust the steering angle.
        """
        if np.random.rand() < 0.5:
            image = cv2.flip(image, 1)
            steering_angle = -steering_angle
        return image, steering_angle

    def random_translate(self, image, steering_angle, range_x, range_y):
        """
        Randomly shift the image vertically and horizontally (translation).
        """
        trans_x = range_x * (np.random.rand() - 0.5)
        trans_y = range_y * (np.random.rand() - 0.5)
        steering_angle += trans_x * 0.002
        trans_m = np.float32([[1, 0, trans_x], [0, 1, trans_y]])
        height, width = image.shape[:2]
        image = cv2.warpAffine(image, trans_m, (width, height))
        return image, steering_angle

    def random_shadow(self, image):
        """
        Generates and adds random shadow
        """
```

```
        # (x1, y1) and (x2, y2) forms a line
        # xm, ym gives all the locations of the image
        x1, y1 = self.img_dims[conf.IMG_WIDTH_IDX] * np.random.rand(), 0
        x2, y2 = self.img_dims[conf.IMG_WIDTH_IDX] * np.random.rand(), self.img_dims[conf.IMG_HEIGHT_ID
X]
        xm, ym = np.mgrid[0:self.img_dims[conf.IMG_HEIGHT_IDX], 0:self.img_dims[conf.IMG_WIDTH_IDX]]

        # mathematically speaking, we want to set 1 below the line and zero otherwise
        # Our coordinate is up side down.  So, the above the line:
        # (ym-y1)/(xm-x1) > (y2-y1)/(x2-x1)
        # as x2 == x1 causes zero-division problem, we'll write it in the below form:
        # (ym-y1)*(x2-x1) - (y2-y1)*(xm-x1) > 0
        mask = np.zeros_like(image[:, :, 1])
        mask[(ym - y1) * (x2 - x1) - (y2 - y1) * (xm - x1) > 0] = 1

        # choose which side should have shadow and adjust saturation
        cond = mask == np.random.randint(2)
        s_ratio = np.random.uniform(low=0.2, high=0.5)

        # adjust Saturation in HLS(Hue, Light, Saturation)
        hls = cv2.cvtColor(image, cv2.COLOR_RGB2HLS)
        hls[:, :, 1][cond] = hls[:, :, 1][cond] * s_ratio
        return cv2.cvtColor(hls, cv2.COLOR_HLS2RGB)

    def random_brightness(self, image):
        """
        Randomly adjust brightness of the image.
        """
        # HSV (Hue, Saturation, Value) is also called HSB ('B' for Brightness).
        hsv = cv2.cvtColor(image, cv2.COLOR_RGB2HSV)
        ratio = 1.0 + 0.4 * (np.random.rand() - 0.5)
        hsv[:, :, 2] = hsv[:, :, 2] * ratio
        return cv2.cvtColor(hsv, cv2.COLOR_HSV2RGB)

    def augment(self, image, steering_angle, range_x=100, range_y=10):
        """
        Generate an augumented image and adjust steering angle.
        (The steering angle is associated with the center image)
        """
        # resize according to expected input shape e.g. AlexNet 224x224, Udacity 320x160, Unity 160x120, etc
        # set in conf.py
        #image = cv2.resize(image, (self.img_dims[conf.IMG_WIDTH_IDX], self.img_dims[conf.IMG_HEIGHT_ID
X]),
        #                   cv2.INTER_AREA)
        image = self.resize_expected(image)
        # image, steering_angle = choose_image(data_dir, center, left, right, steering_angle)
        image, steering_angle = self.random_flip(image, steering_angle)
        image, steering_angle = self.random_translate(image, steering_angle, range_x, range_y)
        image = self.random_shadow(image)
        image = self.random_brightness(image)
        return image, steering_angle


##########################################################
```

```
# 3. augment.ipynb.py
#####################################################


#!/usr/bin/env python
# coding: utf-8

# In[1]:


# start jupyter notebooks from prompt to load all required libraries
# jupyter notebook
import cv2


# In[3]:


import matplotlib.image as mpimg
fp = '/home/simbox/git/msc-data/unity/log2/logs_Fri_Jul_10_09_16_18_2020/10000_cam-image_array_.jpg'
# fp = '/home/simbox/Downloads/IMG/center_2020_11_10_22_02_48_622.jpg'
img = mpimg.imread(fp)
# adapt to naoki net, we have 160w x 120h, first scale to 200
img = cv2.resize(img, (320,160), cv2.INTER_AREA)
import matplotlib.pyplot as plt
plt.imshow(img)
#plt.imshow(img)\n",
# plt.imshow(img[61:-25, :, :]) # image is 120h160w3d: 60:-25 ~ start at h pixel index 60, end at index (120) - 25
                        # equivalent to img[60:95, :, :]
                        # plain english: remove sky and car shadow
# print(img[59:-25, :, :].shape)
    # img[50:-25, :, :].shape"


# In[8]:


# first resize
image = load_image(fp)
image = cv2.resize(image, (320, 160), cv2.INTER_AREA)
image = crop(image)
plt.imshow(image)


# In[33]:


image = load_image(fp)
image = cv2.resize(image, (320, 160), cv2.INTER_AREA)
steering_angle = 0.07
# image, steering_angle = augment(image, steering_angle)
# image = crop(image)
# image = crop(image)
# image = resize(image)
```

```
# image = rgb2yuv(image)
# print(steering_angle)
plt.imshow(image)



# In[2]:


import cv2, os
import numpy as np
import matplotlib.image as mpimg

# Udacity
IMAGE_HEIGHT, IMAGE_WIDTH, IMAGE_CHANNELS = 160, 320, 3
# Alexnet
#IMAGE_HEIGHT, IMAGE_WIDTH, IMAGE_CHANNELS = 224, 224, 3
# IMAGE_HEIGHT, IMAGE_WIDTH, IMAGE_CHANNELS = 66, 200, 3
# IMAGE_HEIGHT, IMAGE_WIDTH, IMAGE_CHANNELS = 120, 160, 3
INPUT_SHAPE = (IMAGE_HEIGHT, IMAGE_WIDTH, IMAGE_CHANNELS)
# Dimensions expected by network
# Udacity
IMAGE_WIDTH_NET, IMAGE_HEIGHT_NET = 200, 66
# Alexnet
# IMAGE_WIDTH_NET, IMAGE_HEIGHT_NET = 224, 224



def load_image(image_path):
    """
    Load RGB images from a file
    """
    return mpimg.imread(image_path)


def crop(image):
    """
    Crop the image (removing the sky at the top and the car front at the bottom)
    """
    # unity
    # return image[60:-25, :, :] # remove the sky and the car front
    # alexnet
    return image[109:-40, :, :] # remove the sky and the car front


def resize(image):
    """
    Resize the image to the input shape used by the network model
    """
    return cv2.resize(image, (IMAGE_WIDTH_NET, IMAGE_HEIGHT_NET), cv2.INTER_AREA)


def rgb2yuv(image):
    """
```

```
    Convert the image from RGB to YUV (This is what the NVIDIA model does)
    """
    return cv2.cvtColor(image, cv2.COLOR_RGB2YUV)


def preprocess(image):
    """
    Combine all preprocess functions into one
    """
    image = crop(image)
    image = resize(image)
    image = rgb2yuv(image)
    return image


def choose_image(data_dir, center, left, right, steering_angle):
    """
    Randomly choose an image from the center, left or right, and adjust
    the steering angle.
    """
    choice = np.random.choice(3)
    if choice == 0:
        return load_image(data_dir, left), steering_angle + 0.2
    elif choice == 1:
        return load_image(data_dir, right), steering_angle - 0.2
    return load_image(data_dir, center), steering_angle


def random_flip(image, steering_angle):
    """
    Randomly flipt the image left <-> right, and adjust the steering angle.
    """
    if np.random.rand() < 0.5:
        image = cv2.flip(image, 1)
        steering_angle = -steering_angle
    return image, steering_angle


def random_translate(image, steering_angle, range_x, range_y):
    """
    Randomly shift the image virtially and horizontally (translation).
    """
    trans_x = range_x * (np.random.rand() - 0.5)
    trans_y = range_y * (np.random.rand() - 0.5)
    steering_angle += trans_x * 0.002
    trans_m = np.float32([[1, 0, trans_x], [0, 1, trans_y]])
    height, width = image.shape[:2]
    image = cv2.warpAffine(image, trans_m, (width, height))
    return image, steering_angle


def random_shadow(image):
    """
    Generates and adds random shadow
```

```
    """
    # (x1, y1) and (x2, y2) forms a line
    # xm, ym gives all the locations of the image
    x1, y1 = IMAGE_WIDTH * np.random.rand(), 0
    x2, y2 = IMAGE_WIDTH * np.random.rand(), IMAGE_HEIGHT
    # could this be a bug?
    xm, ym = np.mgrid[0:IMAGE_HEIGHT, 0:IMAGE_WIDTH]
    # xm, ym = np.mgrid[0:IMAGE_WIDTH, 0:IMAGE_HEIGHT]

    # mathematically speaking, we want to set 1 below the line and zero otherwise
    # Our coordinate is up side down.  So, the above the line:
    # (ym-y1)/(xm-x1) > (y2-y1)/(x2-x1)
    # as x2 == x1 causes zero-division problem, we'll write it in the below form:
    # (ym-y1)*(x2-x1) - (y2-y1)*(xm-x1) > 0
    mask = np.zeros_like(image[:, :, 1])
    mask[(ym - y1) * (x2 - x1) - (y2 - y1) * (xm - x1) > 0] = 1

    # choose which side should have shadow and adjust saturation
    cond = mask == np.random.randint(2)
    s_ratio = np.random.uniform(low=0.2, high=0.5)

    # adjust Saturation in HLS(Hue, Light, Saturation)
    hls = cv2.cvtColor(image, cv2.COLOR_RGB2HLS)
    hls[:, :, 1][cond] = hls[:, :, 1][cond] * s_ratio
    return cv2.cvtColor(hls, cv2.COLOR_HLS2RGB)


def random_brightness(image):
    """
    Randomly adjust brightness of the image.
    """
    # HSV (Hue, Saturation, Value) is also called HSB ('B' for Brightness).
    hsv = cv2.cvtColor(image, cv2.COLOR_RGB2HSV)
    ratio = 1.0 + 0.4 * (np.random.rand() - 0.5)
    hsv[:,:,2] =  hsv[:,:,2] * ratio
    return cv2.cvtColor(hsv, cv2.COLOR_HSV2RGB)


def augment(image, steering_angle, range_x=100, range_y=10):
    """
    Generate an augumented image and adjust steering angle.
    (The steering angle is associated with the center image)
    """
    # resize - we start with assumed image capture size
    image = cv2.resize(image, (320,160), cv2.INTER_AREA)
    # image = cv2.resize(image, (IMAGE_WIDTH, IMAGE_HEIGHT), cv2.INTER_AREA)
    image, steering_angle = random_flip(image, steering_angle)
    image, steering_angle = random_translate(image, steering_angle, range_x, range_y)
    image = random_shadow(image)
    image = random_brightness(image)
    return image, steering_angle


def batch_generator(data_dir, image_paths, steering_angles, batch_size, is_training):
```

```
    """
    Generate training image give image paths and associated steering angles
    """
    images = np.empty([batch_size, IMAGE_HEIGHT, IMAGE_WIDTH, IMAGE_CHANNELS])
    steers = np.empty(batch_size)
    while True:
        i = 0
        for index in np.random.permutation(image_paths.shape[0]):
            center, left, right = image_paths[index]
            steering_angle = steering_angles[index]
            # argumentation
            if is_training and np.random.rand() < 0.6:
                image, steering_angle = augument(data_dir, center, left, right, steering_angle)
            else:
                image = load_image(data_dir, center)
            # add the image and steering angle to the batch
            images[i] = preprocess(image)
            steers[i] = steering_angle
            i += 1
            if i == batch_size:
                break
        yield images, steers


# In[11]:


# cropping test nvidia1

def crop(image, top_crop, bot_crop):
    """
    Crop the image (removing the sky at the top and the car front at the bottom)
    """
    # unity
    return image[top_crop:bot_crop, :, :] # remove the sky and the car front
    # alexnet
    # return image[100:-50, :, :] # remove the sky and the car front

import matplotlib.pyplot as plt
IMAGE_WIDTH, IMAGE_HEIGHT = 160, 120
fp = '/home/simbox/git/msc-data/unity/log2/logs_Fri_Jul_10_09_16_18_2020/10000_cam-image_array_.jpg'
image = load_image(fp)
image_resized = cv2.resize(image, (IMAGE_WIDTH, IMAGE_HEIGHT), cv2.INTER_AREA)
# plt.imshow(image)
plt.imshow(image_resized)
print(image_resized.shape)
print("Expected (120, 160, 3) nvidia1")


# In[12]:


image_nvidia1_crop = crop(image_resized, 60, -25)
# cropping test
```

```
print("nvidia1 crop (35, 160, 3)")
print(image_nvidia1_crop.shape)
plt.imshow(image_nvidia1_crop)


# In[14]:


image_nvidia_crop_resized = cv2.resize(image_nvidia1_crop, (IMAGE_WIDTH, IMAGE_HEIGHT), cv2.INTER_
AREA)
print(image_nvidia_crop_resized.shape)
print("nvidia1 resized")
plt.imshow(image_nvidia_crop_resized)


# In[25]:


# cropping test nvidia2

import matplotlib.pyplot as plt
IMAGE_WIDTH, IMAGE_HEIGHT = 320, 160
image_resized = cv2.resize(image, (IMAGE_WIDTH, IMAGE_HEIGHT), cv2.INTER_AREA)
# plt.imshow(image)
plt.imshow(image_resized)
print(image_resized.shape)
print("Expected (160, 320, 3) nvidia1")


# In[26]:


image_nvidia2_crop_70_35 = crop(image_resized, 70, -35)
# cropping test
print("nvidia2 70 -35 crop (55, 320, 3)")
print(image_nvidia2_crop_70_35.shape)
plt.imshow(image_nvidia2_crop_70_35)


# In[27]:


image_nvidia_crop_70_35_resized = cv2.resize(image_nvidia2_crop_70_35, (IMAGE_WIDTH, IMAGE_HEIGHT
), cv2.INTER_AREA)
print(image_nvidia_crop_70_35_resized.shape)
print("nvidia1 crop 70 -35 resized")
plt.imshow(image_nvidia_crop_70_35_resized)


# In[20]:


image_nvidia2_crop_91_35 = crop(image_resized, 91, -35)
# cropping test
```

```
print("nvidia2 91 -35 crop (35, 160, 3)")
print(image_nvidia2_crop_91_35.shape)
plt.imshow(image_nvidia2_crop_91_35)


# In[35]:


IMAGE_WIDTH, IMAGE_HEIGHT = 320, 160
image_resized = cv2.resize(image, (IMAGE_WIDTH, IMAGE_HEIGHT), cv2.INTER_AREA)
# plt.imshow(image)
plt.imshow(image_resized)
print(image_resized.shape)
print("Expected (160, 320, 3) nvidia2")


# In[36]:


image_nvidia2_crop_77_35 = crop(image_resized, 77, -35)
# cropping test
print("nvidia2 77 -35 crop (55, 320, 3)")
print(image_nvidia2_crop_77_35.shape)
plt.imshow(image_nvidia2_crop_77_35)


# In[37]:


image_resized = cv2.resize(image_nvidia2_crop_77_35, (IMAGE_WIDTH, IMAGE_HEIGHT), cv2.INTER_ARE
A)
# plt.imshow(image)
plt.imshow(image_resized)
print(image_resized.shape)
print("Expected (160, 320, 3) nvidia2")


# In[21]:


# cropping test nvidia2 - higher crop to include all road markings


import matplotlib.pyplot as plt
IMAGE_WIDTH, IMAGE_HEIGHT = 320, 160
image_resized = cv2.resize(image, (IMAGE_WIDTH, IMAGE_HEIGHT), cv2.INTER_AREA)
# plt.imshow(image)
plt.imshow(image_resized)
print(image_resized.shape)
print("Expected (160, 320, 3) nvidia1")


# In[32]:
```

```
image_nvidia2_crop_81_35 = crop(image_resized,81, -35)
# cropping test
print("nvidia2 81 -35 crop (35, 160, 3)")
print(image_nvidia2_crop_81_35.shape)
plt.imshow(image_nvidia2_crop_81_35)


# In[33]:


IMAGE_WIDTH, IMAGE_HEIGHT = 320, 160
image_resized = cv2.resize(image_nvidia2_crop_81_35, (IMAGE_WIDTH, IMAGE_HEIGHT), cv2.INTER_ARE
A)
# plt.imshow(image)
plt.imshow(image_resized)
print(image_resized.shape)
print("Expected (160, 320, 3) nvidia2")


# In[34]:


IMAGE_WIDTH, IMAGE_HEIGHT = 320, 160
image_resized = cv2.resize(image_nvidia2_crop_81_35, (IMAGE_WIDTH, IMAGE_HEIGHT), cv2.INTER_ARE
A)
image_nvidia2_crop_77_35 = crop(image_resized,77, -35)
# cropping test
print("nvidia2 77 -35 crop (35, 160, 3)")
print(image_nvidia2_crop_77_35.shape)
plt.imshow(image_nvidia2_crop_77_35)


# In[64]:


# images for dissertation - Augmentation section
# Finally move from RGB to YUV colour space
image = load_image(fp)
image = cv2.resize(image, (320, 160), cv2.INTER_AREA)
steering_angle = 0.07
image, steering_angle = augment(image, steering_angle)
image = crop(image)
image = resize(image)
image = rgb2yuv(image)
# print(steering_angle)
plt.imshow(image)
print(image.shape) # original size of this image: 120 x 160


# In[103]:


# images for dissertation - Augmentation section
```

```
image = load_image(fp)
image1 = cv2.resize(image, (320, 160), cv2.INTER_AREA) # original NVIDIA capture size
steering_angle = 0.07
image2, steering_angle = augment(image1, steering_angle) # augmented
image3 = crop(image2) # cropped
image4 = resize(image3) # resized to network design
image5 = rgb2yuv(image4) # RGB to YUV transform
# print(steering_angle)
#plt.imshow(image)
#print(image.shape) # original size of this image: 120 x 160

# plot for dissertation - cannot adjust padding between rows so further image processing required
import matplotlib.pyplot as plt
import numpy as np
import matplotlib
matplotlib.rcParams['font.size'] = 12.0

fig, axs = plt.subplots(2, 3)

fig.set_figheight(15)
fig.set_figwidth(15)

# Raw
axs[0, 0].title.set_text('Raw ' + str(image1.shape))
axs[0, 0].imshow(image1)

# Augmented
axs[0, 1].title.set_text('Augmented ' + str(image2.shape))
axs[0, 1].imshow(image2)

# cropped
axs[0, 2].title.set_text('Cropped ' + str(image3.shape))
axs[0, 2].imshow(image3)

# resized
axs[1, 0].title.set_text('Resized ' + str(image4.shape))
axs[1, 0].imshow(image4)

# RGBtoYUV
axs[1, 1].title.set_text('RGB to YUV ' + str(image5.shape))
axs[1, 1].imshow(image5)

# Dummy to better format
axs[1, 2].title.set_text('RGB to YUV ' + str(image5.shape))
axs[1, 2].imshow(image5)


plt.show()


# In[91]:


import numpy as np
```

```
import matplotlib.pyplot as plt
from matplotlib import gridspec

nrow = 2
ncol = 3

fig = plt.figure(figsize=(10, 10))

gs = gridspec.GridSpec(nrow, ncol, width_ratios=[1, 1, 1],
         wspace=0.0, hspace=0.0, top=0.95, bottom=0.05, left=0.17, right=0.845)

for i in range(2):
    for j in range(3):
        im = np.random.rand(28,28)
        ax= plt.subplot(gs[i,j])
        ax.imshow(image)
        ax.set_xticklabels([])
        ax.set_yticklabels([])

#plt.tight_layout() # do not use this!!
plt.show()



##################################################
# 4. conf.py
# Note: code based on
# https://github.com/tawnkramer/sdsandbox/blob/master/src/conf.py
# Available for audit in audit_files/tawn from sharepoint link
##################################################


import math

training_patience = 6
training_default_epochs = 100
training_default_aug_mult = 1
training_default_aug_percent = 0.0
learning_rate = 0.00001 # 0.00001

# default model name
# model_name = 'nvidia1'
#nvidia 1 - use this size for both
#nvidia 1 and 2, just change nvidia size before
# presenting to neural network
#image_width = 160
#image_height = 120
#nvidia 2

# size augmentation process is expecting, i.e. what came from camera
# AlexNet 224x224, Udacity 320x160, Unity 160x120, etc

# IMAGE DIMS INDEXES
# expected original image size
```

```
IMG_WIDTH_IDX = 0
IMG_HEIGHT_IDX = 1
IMG_DEPTH_IDX = 2
# size to be presented to network
IMG_WIDTH_NET_IDX = 3
IMG_HEIGHT_NET_IDX = 4
# to crop road from image
IMG_TOP_CROP_IDX = 5
IMG_BOTTOM_CROP_IDX = 6

# What these lists mean:
# Expected width, height and depth of acquired image,
# width and height of image expected by network
# top crop and bottom crop to remove car and sky from image
# ALEXNET
ALEXNET = 'alexnet'
alexnet_img_dims = [224,224,3,224,224,60,-25]
# NVIDIA1
NVIDIA1 = 'nvidia1' # a.k.a. TawnNe,
nvidia1_img_dims = [160,120,3,160,120,60,-25]
# NVIDIA2
NVIDIA2 = 'nvidia2' # a.k.a. NaokiNet
nvidia2_img_dims = [320,160,3,200,66,81,-35]
# NVIDIA_BASELINE
NVIDIA_BASELINE = 'nvidia_baseline' # a.k.a. NaokiNet
nvidia_baseline_img_dims = [160,120,3,200,66,60,-25]

# Alexnet
image_width_alexnet = 224
image_height_alexnet = 224
# nvidia
image_width = 160
image_height = 120
#nvidia2 (Udacity NaokiNet)
#image_width = 160
#image_height = 120

# size network is expecting
image_width_net = 160
image_height_net = 120
# same for all
image_depth = 3

row = image_height_net
col = image_width_net
ch = image_depth

# training for steering and throttle:
num_outputs = 2
# steering alone:
# num_outputs = 1

throttle_out_scale = 1.0
# alexnet
```

```
batch_size = 64

# Using class members to avoid passing same parameters through various functions
# The original NVIDIA paper mentions augmentation but no cropping i.e. road only
# augmentation
aug = False
# pre-process image: crop, resize and rgb2yuv
preproc = False
# image normalization constant, Unity model maximum steer
norm_const = 25

# rain type and slant
rt = ''
st = 0

# video recording
VIDEO_WIDTH, VIDEO_HEIGHT = 800, 600
IMAGE_STILL_WIDTH, IMAGE_STILL_HEIGHT = 800, 600
record = False

def setdims(modelname):
    """
    Set image dimensions for training and predicting
    Inputs
        modelname: string, network name
    Outputs
        none
    Example
    setdims('alexnet') # set width and height to 224
    """
    if(modelname=='alexnet'):
        conf.row = image_width_alexnet
        self.col = image_height_alexnet


#######################################################
# 5. data_predict.py
#######################################################


# Predict steering angles for a dataset for which a ground truth exists
# dsikar@gmail.com

from __future__ import print_function

import argparse
import fnmatch
import json
import os
import pickle
import random
from datetime import datetime
from time import strftime
import numpy as np
```

```
from PIL import Image
from tensorflow import keras
import tensorflow as tf
import conf
import models
from helper_functions import hf_mkdir
from augmentation import augment, preprocess
import cv2
from train import load_json, get_files
from augmentation import preprocess
from utils.steerlib import gos, plotSteeringAngles
from pathlib import Path

from tensorflow.python.keras.models import load_model



# 1. get a list of files to predict (sequential)
# 2. read the steering angle (json file)
# 3. generate a prediction
# 4. Store results in list
# 5. Generate a "goodness of steer" value (average steering error)
# 6. Generate graph

def predict_drive(datapath, modelpath, nc):
    """
    Generate predictions from a model for a dataset
    Inputs
        datapath: string, path to data
        modelpath: string, path to trained model
        nc: steering angle normalization constant
    """
    print("loading model", modelpath)
    model = load_model(modelpath)

    # In this mode, looks like we have to compile it
    # NB this is a bit tricky, do need to use optimizer and loss function used to train model?
    model.compile("sgd", "mse")

    files = get_files(datapath, True)
    outputs = []
    for fullpath in files:
        frame_number = os.path.basename(fullpath).split("_")[0]
        json_filename = os.path.join(os.path.dirname(fullpath), "record_" + frame_number + ".json")
        data = load_json(json_filename)
        # ground truth
        steering = float(data["user/angle"]) # normalized - divided by nc by simulator
        # prediction
        image = cv2.imread(fullpath)
        # The image will be 1. resized to expected pre-processing size and 2.resized to expected
        # size to be presented to network. This is network architecture and dataset dependant and
        # currently managed in conf.py
        image = preprocess(image)
        image = image.reshape((1,) + image.shape)
```

```
        mod_pred = model.predict(image)
        # append prediction and ground truth to list
        outputs.append([mod_pred[0][1], steering])
    # get goodness of steer
    sarr = np.asarray(outputs)
    p = sarr[:, 0]
    g = sarr[:, 1]
    gs = gos(p,g,nc)
    print(gs)
    # def plotSteeringAngles(p, g=None, n=1, save=False, track= "Track Name", mname="model name", title='title'):
    gss = "{:.2f}".format(gs)
    modelpath = modelpath.split('/')
    datapath = datapath.split('/')
    plotSteeringAngles(p, g, nc, True, datapath[-2], modelpath[-1], 'Gs ' + gss)


# dataset ../dataset/unity/jungle1/
# model ../trained_models/nvidia2/20201124032017_nvidia2.h5
    # calculate gos (average steering error)
    # plot graph unormalized angles. + average steering error
    # save graph.
    # done

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description='prediction server')
    parser.add_argument('--datapath', type=str, default='/home/simbox/git/msc-data/unity/genTrackOneLap_3/*.jpg',
help='model filename')
    parser.add_argument('--modelpath', type=str, default='/home/simbox/git/sdsandbox/trained_models/sanity/202011
20171015_sanity.h5', help='Model')
    parser.add_argument('--nc', type=int, default=1, help='Steering Angle Normalization Constant')
    # time allowing, set image sizes based on model name. For now, these have to be managed in conf.py
    #parser.add_argument('--model', type=str, default='nvidia1', help='model name')

    # set dimensions

    args = parser.parse_args()

    predict_drive(args.datapath, args.modelpath, args.nc)
    # max value for slant is 20
    # Example
    # python3 predict_client.py --model=../trained_models/sanity/20201120171015_sanity.h5 --rain=light --slant=0
```

```
#####################################################
# 6. GetSteeringAnglesFromtcpflow.ipynb.py
#####################################################


#!/usr/bin/env python
```

```
# coding: utf-8

# In[22]:


import json
import numpy as np
import matplotlib.pyplot as plt
import statistics
import seaborn as sns
import os
import fnmatch

def GetSteeringFromtcpflow(filename):
    """
    Get a tcpflow log and extract steering values obtained from network communication between.
    Note, we only plot the predicted steering angle jsondict['steering']
    and the value of jsondict['steering_angle'] is ignored. Assumed to be the steering angle
    calculated by PID given the current course.
    sim and prediction engine (predict_client.py)
    Inputs
        filename: string, name of tcpflow log
    Returns
        sa: list of arrays, steering angle predicton and actual value tuple.
    Example


    """
    # open file
    sa = []
    # initialize prediction
    pred = ''
    f = open(filename, "r")
    file = f.read()
    try:
        #readline = f.read()
        lines = file.splitlines()
        for line in lines:
            # print(line)
            start = line.find('{')
            if(start == -1):
                continue
            jsonstr = line[start:]
            # print(jsonstr)
            jsondict = json.loads(jsonstr)
            if "steering" in jsondict:
                # predicted
                pred = jsondict['steering']
                # jsondict['steering_angle']
                # sa.append([float(pred), act])
                sa.append([float(pred), float(pred)]) # append twice to keep code from breaking
            #if "steering_angle" in jsondict:
                # actual
             #   act = jsondict['steering_angle']
```

```
            # save pair, only keep last pred in case two were send as it does happen i.e.:
            # 127.000.000.001.59460-127.000.000.001.09091: {"msg_type": "control", "steering": "-0.071960375", "t
hrottle": "0.08249988406896591", "brake": "0.0"}
            # 127.000.000.001.59460-127.000.000.001.09091: {"msg_type": "control", "steering": "-0.079734944", "t
hrottle": "0.08631626516580582", "brake": "0.0"}
            # 127.000.000.001.09091-127.000.000.001.59460: {"msg_type":"telemetry","steering_angle":-0.0719603
7,(...)
          #   if(len(pred) > 0):
           #      sa.append([float(pred), act])
           #      pred = '' # need to save this image
             # deal with image later, sort out plot first
             #imgString = jsondict["image"]
             #image = Image.open(BytesIO(base64.b64decode(imgString)))
             #img_arr = np.asarray(image, dtype=np.float32)
    except Exception as e:
        print("Exception raise: " + str(e))
    # file should be automatically closed but will close for good measure
    f.close()
    return sa

def plotBinsFromArray(svals, nc=25, pname=None, logname = "tcpflow log name"):
    svalscp = [element * nc for element in svals]
    values = len(svalscp)
    mymean = ("%.2f" % statistics.mean(svalscp))
    mystd = ("%.2f" % statistics.stdev(svalscp))
    plt.title=(pname)
    # NB Plotted as normalized histogram
    sns.distplot(svalscp, bins=nc*2, kde=False, norm_hist=True,
    axlabel=  "tcpflow: " + logname + ", model: " + pname + ' steer. degs. norm. hist. ' + str(values) + ' values, mean =
 ' + mymean + ' std = ' + mystd)
    #if(save):
    #    sns.save("output.png")
    plt.savefig(pname + '.png')
# Steering angle predictions by model 20201107210627_nvidia1.h5

def plotSteeringAngles(p, g, n, save=False, track= "Track Name", mname="model.h5"):
    """
    Plot predicted steering angles
    """
    plt.rcParams["figure.figsize"] = (18,3)

    plt.plot(p*25)
    # plt.plot(sarr[:,1]*25, label="simulator")

    plt.ylabel('Steering angle')
    plt.xlabel('Frame number')
    # Set a title of the current axes.
    mytitle = 'tcpflow log predicted steering angles: track ' + str(track) + ' model ' + str(mname)
    plt.title(mytitle)
    # show a legend on the plot
    #plt.legend()
    # Display a figure.
    # horizontal grid only
    plt.grid(axis='y')
```

```
    # set limit
    plt.xlim([-5,len(p)+5])
    plt.gca().invert_yaxis()
    plt.show()

def plotMultipleSteeringAngles(p, n, save=False, track= "Track Name", mname="model.h5", w=18, h=3):
    """
    Plot multiple predicted steering angles
    Inputs
        p: list of tuples, steering angles and labels
        n: integer,
    """
    plt.rcParams["figure.figsize"] = (18,3)


    plt.plot(p*25)
    # plt.plot(sarr[:,1]*25, label="simulator")

    plt.ylabel('Steering angle')
    plt.xlabel('Frame number')
    # Set a title of the current axes.
    mytitle = 'tcpflow log predicted steering angles: track ' + str(track) + ' model ' + str(mname)
    plt.title(mytitle)
    # show a legend on the plot
    #plt.legend()
    # Display a figure.
    # horizontal grid only
    plt.grid(axis='y')
    # set limit
    plt.xlim([-5,len(p)+5])
    plt.gca().invert_yaxis()
    plt.show()

def GetJSONSteeringAngles(filemask):
    """
    Get steering angles stored as 'user/angle' attributes in .json files
    Inputs:
        filemask: string, path and mask
    Outputs
        svals: list, steering values
    """
    filemask = os.path.expanduser(filemask)
    path, mask = os.path.split(filemask)

    matches = []
    for root, dirnames, filenames in os.walk(path):
        for filename in fnmatch.filter(filenames, mask):
            matches.append(os.path.join(root, filename))
    # sort by create date
    matches = sorted(matches, key=os.path.getmtime)
    # steering values
    svals = []
    for fullpath in matches:
            frame_number = os.path.basename(fullpath).split("_")[0]
```

```
        json_filename = os.path.join(os.path.dirname(fullpath), "record_" + frame_number + ".json")
        jobj = load_json(json_filename)
        svals.append(jobj['user/angle'])
    return svals

def load_json(filepath):
    """
    Load a json file
    Inputs
        filepath: string, path to file
    Outputs
        data: dictionary, json key, value pairs
    Example
    path = "~/git/msc-data/unity/roboRacingLeague/log/logs_Sat_Nov_14_12_36_16_2020/record_11640.json"
    js = load_json(path)
    """
    with open(filepath, "rt") as fp:
        data = json.load(fp)
    return data


# In[2]:


sa = GetSteeringFromtcpflow('../trained_models/nvidia2/tcpflow/20201207124146_nvidia2_tcpflow.log')
sarr = np.asarray(sa)
p = sarr[:,0]
g = sarr[:,0]
plotSteeringAngles(p, g, 25, False, "Generated Track", "20201207124146_nvidia2.h5")
plotBinsFromArray(p, 25, "20201207124146_nvidia2.h5", "20201207124146_nvidia2_tcpflow.log")


# In[28]:


# plot ground truth steering angles for
filemask = '../dataset/unity/genTrack/genTrackOneLap/logs_Wed_Nov_25_23_39_22_2020/*.jpg'
g = GetJSONSteeringAngles(filemask)
# print(type(g)) # list
g = np.asarray(g)
# print(type(g)) # <class 'numpy.ndarray'>
plt.rcParams["figure.figsize"] = (18,3)
nc = 25 # norm. constant, maximum steering angle

plt.plot(g*nc)
# plt.plot(sarr[:,1]*25, label="simulator")

plt.ylabel('Steering angle')
plt.xlabel('Frame number')
# Set a title of the current axes.
mytitle = 'Ground truth steering for logs_Wed_Nov_25_23_39_22_2020 - One lap recorded from Generated Track i
n "Auto Drive w Rec" mode'
plt.title(mytitle)
plt.grid(axis='y')
```

```
# set limit
plt.xlim([-5,len(g)+5])
plt.gca().invert_yaxis()
plt.show()


# In[2]:


# Steering angle predictions by model 20201107210627_nvidia1.h5
def plotSteeringAngles(p, g, n, save=False, track= "Track Name", mname="model.h5"):
    """
    Plot predicted steering angles
    """
    import matplotlib.pyplot as plt

    plt.rcParams["figure.figsize"] = (18,3)

    plt.plot(p*25)
    # plt.plot(sarr[:,1]*25, label="simulator")

    plt.ylabel('Steering angle')
    plt.ylabel('Frame number')
    # Set a title of the current axes.
    plt.title('tcpflow log predicted steering angles: track ' + track + ' model ' + mname)
    # show a legend on the plot
    #plt.legend()
    # Display a figure.
    # horizontal grid only
    plt.grid(axis='y')
    # set limit
    plt.xlim([-5,len(p)+5])
    plt.gca().invert_yaxis()
    plt.show()

sa = GetSteeringFromtcpflow('../dataset/unity/genRoad/tcpflow/20201120184912_sanity.log')
sarr = np.asarray(sa)
p = sarr[:,0]
g = sarr[:,1]

plotSteeringAngles(p, g, 25, False, "Generated Road, ", "20201120184912_sanity.h5")
#sa = GetSteeringFromtcpflow('../dataset/unity/genRoad/tcpflow/20201120184912_sanity.log')
#plotBinsFromArray(p, 25, "20201120184912_sanity.h5", "tcpflow/20201120184912_sanity.log")
#type(g)

# TypeError: 'str' object is not callable - cleared by restarting kernel (clearing variables?)


# In[3]:


def gos(p, g, n):
    """
    Calculate the goodness-of-steer between a prediction and a ground truth array.
```

```
    Inputs
        p: array of floats, steering angle prediction
        g: array of floats, steering angle ground truth.
        n: float, normalization constant
    Output
        gos: float, average of absolute difference between ground truth and prediction arrays
    """
    # todo add type assertion
    assert len(p) == len(g), "Arrays must be of equal length"
    return sum(abs(p - g)) / len(p) * n

#p = sarr[:,0]
#g = sarr[:,1]

#sterr = gos(p,g, 25)

#print("Goodness of steer: {:.2f}".format(sterr))


# In[4]:


import statistics
import matplotlib.pyplot as plt
import seaborn as sns
def plotBinsFromArray(svals, nc=25, pname=None, logname = "tcpflow log name"):
    svalscp = [element * nc for element in svals]
    values = len(svalscp)
    mean = ("%.2f" % statistics.mean(svalscp))
    std = ("%.2f" % statistics.stdev(svalscp))
    plt.title=(pname)
    # NB Plotted as normalized histogram
    sns.distplot(svalscp, bins=nc*2, kde=False, norm_hist=True,
    axlabel=  "tcpflow: " + logname + ", model: " + pname + ' steer. degs. norm. hist. ' + str(values) + ' values, mean =
 ' + mean + ' std = ' + std)
    #if(save):
    #    sns.save("output.png")
    plt.savefig(pname + '.png')

#sa = GetSteeringFromtcpflow('../dataset/unity/genRoad/tcpflow/20201120184912_sanity.log')
#sarr = np.asarray(sa)
#p = sarr[:,0]
#p = sarr[:,0]
#plotBinsFromArray(p, 25, "20201120184912_sanity.h5", "tcpflow/20201120184912_sanity.log")


# In[5]:


#### ../dataset/unity/genRoad/tcpflow/20201123162643_sanity.log
sa = GetSteeringFromtcpflow('../dataset/unity/genRoad/tcpflow/20201120184912_sanity.log')
sarr = np.asarray(sa)
p = sarr[:,0]
p = sarr[:,0]
```

```
plotSteeringAngles(p, g, 25, False, "Generated Road", "20201120184912_sanity.h5")
plotBinsFromArray(p, 25, "20201120184912_sanity.h5", "tcpflow/20201123162643_sanity.log")


# In[6]:


#### ../dataset/unity/genRoad/tcpflow/20201123162643_sanity.log
sa = GetSteeringFromtcpflow('../dataset/unity/genRoad/tcpflow/20201123162643_sanity.log')
sarr = np.asarray(sa)
p = sarr[:,0]
p = sarr[:,0]
plotSteeringAngles(p, g, 25, False, "Generated Road", "20201123162643_sanity.h5")
plotBinsFromArray(p, 25, "20201123162643_sanity.h5", "tcpflow/20201123162643_sanity.log")


# In[ ]:


#### ../dataset/unity/genRoad/tcpflow/20201123162643_sanity_pp.log
sa = GetSteeringFromtcpflow('../dataset/unity/genRoad/tcpflow/20201123162643_sanity_pp.log')
sarr = np.asarray(sa)
p = sarr[:,0]
p = sarr[:,0]
plotSteeringAngles(p, g, 25, False, "Generated Road", "20201123162643_sanity_pp.h5")
plotBinsFromArray(p, 25, "20201123162643_sanity.h5", "tcpflow/20201123162643_sanity_pp.log")


# In[7]:


sa = GetSteeringFromtcpflow('../trained_models/nvidia2/tcpflow/20201207111940_nvidia2_tcpflow.log')
sarr = np.asarray(sa)
p = sarr[:,0]
p = sarr[:,0]
plotSteeringAngles(p, g, 25, False, "Generated Track", "20201207111940_nvidia2.h5")
plotBinsFromArray(p, 25, "20201207111940_nvidia2.h5", "20201207111940_nvidia2_tcpflow.log")


# In[ ]:




#####################################################
# 7. GetSteering.py
#####################################################


import argparse
import fnmatch
import json
```

```
import os
from io import BytesIO
from PIL import Image
import base64
import numpy as np
import matplotlib.pyplot as plt
from augmentation import preprocess

def GetSteering(filename):
    """
    Get a tcpflow log and extract steering values received from and sent to sim
    Inputs
        filename: string, name of tcpflow log
    """
    # open file
    sa = []
    # initialize prediction
    pred = ''
    f = open(filename, "r")
    file = f.read()
    try:
        #readline = f.read()
        lines = file.splitlines()
        for line in lines:
            #print(line)
            start = line.find('{')
            if(start == -1):
                continue
            jsonstr = line[start:]
            #print(jsonstr)
            jsondict = json.loads(jsonstr)
            if "steering" in jsondict:
                # predicted
                pred = jsondict['steering']
            if "steering_angle" in jsondict:
                # actual
                act = jsondict['steering_angle']
                # save pair, only keep last pred in case two were send as it does happen i.e.:
                # 127.000.000.001.59460-127.000.000.001.09091: {"msg_type": "control", "steering": "-0.071960375", "t
hrottle": "0.08249988406896591", "brake": "0.0"}
                # 127.000.000.001.59460-127.000.000.001.09091: {"msg_type": "control", "steering": "-0.079734944", "t
hrottle": "0.08631626516580582", "brake": "0.0"}
                # 127.000.000.001.09091-127.000.000.001.59460: {"msg_type":"telemetry","steering_angle":-0.0719603
7,(...)
                if(len(pred) > 0):
                    sa.append([float(pred), act])
                    pred = '' # need to save this image
                # deal with image later, sort out plot first
                    imgString = jsondict["image"]
                    image = Image.open(BytesIO(base64.b64decode(imgString)))
                    img_arr = np.asarray(image, dtype=np.float32)
                    img_arr_proc = preprocess(img_arr)
                    stitch = stitchImages(img_arr, img_arr_proc, 160, 120)
                    plt.imshow(stitch)
```

```
    except Exception as e:
        print("Exception raise: " + str(e))
    # file should be automatically closed but will close for good measure
    f.close()


def stitchImages(a, b, w, h):
    """
    Stitch two images together side by side
    Inputs
        a, b: floating point image arrays
        w, h: integer width and height dimensions
    Output
        c: floating point stitched image array
    """
    # https://stackoverflow.com/questions/30227466/combine-several-images-horizontally-with-python
    total_width = w * 2
    max_height = h

    a = Image.fromarray(a.astype('uint8'), 'RGB')
    b = Image.fromarray(b.astype('uint8'), 'RGB')

    new_im = Image.new('RGB', (total_width, max_height))
    new_im.paste(a, (0,0))
    new_im.paste(b, (w,0))

    return new_im # new_im



if __name__ == "__main__":
    parser = argparse.ArgumentParser(description='train script')
    parser.add_argument('--filename', type=str, help='tcpflow log')
    args = parser.parse_args()
    GetSteering(args.filename)

#####################################################
# 8. jsonclean.py
#####################################################


"""
Go though Unity3D generated files, delete .jpg if a corresponding .json file does not exist
$ python3 jsonclean.py --inputs=../../dataset/unity/log/*.jpg --delete=[False|True]
"""

import os
import argparse
import fnmatch
import json


def load_json(filename):
    with open(filename, "rt") as fp:
```

```
        data = json.load(fp)
    return data

def cleanjson(filemask, delete):
    # filemask = '~/git/sdsandbox/dataset/unity/log/*.jpg'
    filemask = os.path.expanduser(filemask)
    path, mask = os.path.split(filemask)

    matches = []
    for root, dirnames, filenames in os.walk(path):
        for filename in fnmatch.filter(filenames, mask):
            matches.append(os.path.join(root, filename))

    # deleted file count
    dc = 0
    for fullpath in matches:
            frame_number = os.path.basename(fullpath).split("_")[0]
            json_filename = os.path.join(os.path.dirname(fullpath), "record_" + frame_number + ".json")
            try:
                load_json(json_filename)
            except:
                print('No matching .json file for: ', fullpath)
                # No matching .json file for:  ../../dataset/unity/log/logs_Mon_Jul_13_09_03_21_2020/35095_cam-image
_array_.jpg
                if(delete):
                    print("File deleted.")
                    os.remove(fullpath)
                    dc += 1
                continue

    print("Files deleted:", dc)
def parse_bool(b):
    return b == "True"

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description='JSON missing file handler/cleaner')
    parser.add_argument('--inputs', default='../dataset/unity/jungle1/log/*.jpg', help='input mask to gather images')
    parser.add_argument('--delete', type=parse_bool, default=False, help='image deletion flag')
    args = parser.parse_args()

    cleanjson(args.inputs, args.delete)

#######################################################
# 9. makebins.py
#######################################################


"""
Go though Unity3D generated files, get all steering angles and plot a histogram
$ python3 makebins.py --inputs=../../dataset/unity/log/*.jpg
"""

import os
import argparse
```

```
import fnmatch
import json
import seaborn as sns
import os


def load_json(filename):
    with open(filename, "rt") as fp:
        data = json.load(fp)
    return data

def cleanjson(filemask):

    filemask = os.path.expanduser(filemask)
    path, mask = os.path.split(filemask)

    matches = []
    for root, dirnames, filenames in os.walk(path):
        for filename in fnmatch.filter(filenames, mask):
            matches.append(os.path.join(root, filename))

    # steering values
    svals = []
    for fullpath in matches:
            frame_number = os.path.basename(fullpath).split("_")[0]
            json_filename = os.path.join(os.path.dirname(fullpath), "record_" + frame_number + ".json")
            jobj = load_json(json_filename)
            svals.append(jobj['user/angle'])

    print("Files deleted:", dc)
def parse_bool(b):
    return b == "True"

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description='JSON missing file handler/cleaner')
    parser.add_argument('--inputs', default='../dataset/unity/jungle1/log/*.jpg', help='input mask to gather images')
    args = parser.parse_args()

    cleanjson(args.inputs)

#####################################################
# 10. MakeVideoFromtcpflow.ipynb.py
#####################################################


#!/usr/bin/env python
# coding: utf-8

# In[123]:


#import argparse
#import fnmatch
import json
```

```
#import os
import base64
import datetime
from io import BytesIO
from PIL import Image
import sys
import numpy as np
import matplotlib.pyplot as plt

# convert image array to PIL image, to see if we can concatenate them
# from PIL import Image
from matplotlib import cm


def GetSteeringFromtcpflow(filename):
    """
    Get a tcpflow log and extract steering values obtained from network communication between
    sim and prediction engine (predict_client.py)
    Inputs
        filename: string, name of tcpflow log
    Returns
        sa: list of arrays, steering angle predicton and actual value tuple.
    """
    # open file
    sa = []
    # initialize prediction
    pred = ''
    f = open(filename, "r")
    file = f.read()
    try:
        #readline = f.read()
        lines = file.splitlines()
        for line in lines:
            # print(line)
            start = line.find('{')
            if(start == -1):
                continue
            jsonstr = line[start:]
            # print(jsonstr)
            jsondict = json.loads(jsonstr)
            if "steering" in jsondict:
                # predicted
                pred = jsondict['steering']
            if "steering_angle" in jsondict:
                # actual
                act = jsondict['steering_angle']
                # save pair, only keep last pred in case two were send as it does happen i.e.:
                # 127.000.000.001.59460-127.000.000.001.09091: {"msg_type": "control", "steering": "-0.071960375", "t
hrottle": "0.08249988406896591", "brake": "0.0"}
                # 127.000.000.001.59460-127.000.000.001.09091: {"msg_type": "control", "steering": "-0.079734944", "t
hrottle": "0.08631626516580582", "brake": "0.0"}
                # 127.000.000.001.09091-127.000.000.001.59460: {"msg_type":"telemetry","steering_angle":-0.0719603
7,(...)
                if(len(pred) > 0):
```

```
                sa.append([float(pred), act])
                pred = '' # need to save this image

                imgString = jsondict["image"]
                image = Image.open(BytesIO(base64.b64decode(imgString)))

                img_arr = np.asarray(image, dtype=np.float32)

                img_arr_proc = preprocess(img_arr)
                img_arr = crop(img_arr)
                img_arr = resize(img_arr)
                # something happens here which "fixes" the image, i.e.
                img_arr = rgb2yuv(img_arr)
                #im = Image.fromarray(np.uint8(cm.gist_earth(img_arr)))
                #stitch = stitchImages(img_arr, img_arr_proc, 160, 120)
                #print(img_arr_proc.shape)
                #print(type(image))
                # plt.imshow(img_arr_proc)
                print(type(img_arr))
                myconc = np.concatenate((img_arr_proc, img_arr), axis = 1)
                # plt.imshow(img_arr)
                plt.imshow(myconc)
                # print(image.size[0])
                sys.exit(0)
    except Exception as e:
        print("Exception raise: " + str(e))
    # file should be automatically closed but will close for good measure
    f.close()
    return sa

sa = GetSteeringFromtcpflow('/tmp/tcpflow.log')


# In[52]:


def stitchImages(a, b, w, h):
    """
    Stitch two images together side by side
    Inputs
        a, b: floating point image arrays
        w, h: integer width and height dimensions
    Output
        c: floating point stitched image array
    """
    # https://stackoverflow.com/questions/30227466/combine-several-images-horizontally-with-python
    total_width = w * 2
    max_height = h
    # convert to PIL image to paste
    a = Image.fromarray(a.astype('uint8'), 'RGB')
    b = Image.fromarray(b.astype('uint8'), 'YCbCr')

    new_im = Image.new('RGB', (total_width, max_height))
    new_im.paste(a, (0,0))
```

```
    new_im.paste(b, (w,0))
    # convert back to float array
    return new_im # np.asarray(new_im, dtype=np.float32) # new_im



# In[124]:


myimg = np.asarray([[[1,1,1], [1,1,1], [1,1,1]],
                    [[1,1,1], [1,1,1], [1,1,1]],
                    [[1,1,1], [1,1,1], [1,1,1]]])


myimg2 = np.asarray([[[2,2,2], [2,2,2], [2,2,2]],
                    [[2,2,2], [2,2,2], [2,2,2]],
                    [[2,2,2], [2,2,2], [2,2,2]]])

myimg3 = np.asarray([[[1,1,1], [1,1,1], [1,1,1]],
                    [[1,1,1], [1,1,1], [1,1,1]],
                    [[1,1,1], [1,1,1], [1,1,1]],
                    [[2,2,2], [2,2,2], [2,2,2]],
                    [[2,2,2], [2,2,2], [2,2,2]],
                    [[2,2,2], [2,2,2], [2,2,2]]])

myimg4 = np.asarray([[[1,1,1], [1,1,1], [1,1,1],[2,2,2], [2,2,2], [2,2,2]],
                    [[1,1,1], [1,1,1], [1,1,1],[2,2,2], [2,2,2], [2,2,2]],
                    [[1,1,1], [1,1,1], [1,1,1],[2,2,2], [2,2,2], [2,2,2]]])

print(myimg.shape)
print(myimg2.shape)
print(myimg3.shape)
print(myimg4.shape)
# np.append(myimg, myimg2, axis=0)

import numpy as np
myconc = np.concatenate((myimg, myimg2), axis = 1)
myconc.shape


# In[127]:


# quick test, can we concatenate if we read from disk?
myimg = load_image('../dataset/unity/jungle1/log/logs_Sat_Nov__7_11_11_06_2020/10000_cam-image_array_.jpg'
)

myprocimg = preprocess(myimg)
myconc = np.concatenate((myimg, myprocimg), axis = 1)
plt.imshow(myconc)


# In[4]:
```

```
import cv2, os
import numpy as np
import matplotlib.image as mpimg

IMAGE_HEIGHT, IMAGE_WIDTH, IMAGE_CHANNELS = 160, 120, 3
# IMAGE_HEIGHT, IMAGE_WIDTH, IMAGE_CHANNELS = 66, 200, 3
# IMAGE_HEIGHT, IMAGE_WIDTH, IMAGE_CHANNELS = 120, 160, 3
INPUT_SHAPE = (IMAGE_HEIGHT, IMAGE_WIDTH, IMAGE_CHANNELS)
# Dimensions expected by network
IMAGE_WIDTH_NET, IMAGE_HEIGHT_NET = 160, 120


def load_image(image_path):
    """
    Load RGB images from a file
    """
    return mpimg.imread(image_path)


def crop(image):
    """
    Crop the image (removing the sky at the top and the car front at the bottom)
    """
    return image[60:-25, :, :] # remove the sky and the car front


def resize(image):
    """
    Resize the image to the input shape used by the network model
    """
    return cv2.resize(image, (IMAGE_WIDTH_NET, IMAGE_HEIGHT_NET), cv2.INTER_AREA)


def rgb2yuv(image):
    """
    Convert the image from RGB to YUV (This is what the NVIDIA model does)
    """
    return cv2.cvtColor(image, cv2.COLOR_RGB2YUV)


def preprocess(image):
    """
    Combine all preprocess functions into one
    """
    image = crop(image)
    image = resize(image)
    image = rgb2yuv(image)
    return image


def choose_image(data_dir, center, left, right, steering_angle):
    """
    Randomly choose an image from the center, left or right, and adjust
```

```
    the steering angle.
    """
    choice = np.random.choice(3)
    if choice == 0:
        return load_image(data_dir, left), steering_angle + 0.2
    elif choice == 1:
        return load_image(data_dir, right), steering_angle - 0.2
    return load_image(data_dir, center), steering_angle


def random_flip(image, steering_angle):
    """
    Randomly flipt the image left <-> right, and adjust the steering angle.
    """
    if np.random.rand() < 0.5:
        image = cv2.flip(image, 1)
        steering_angle = -steering_angle
    return image, steering_angle


def random_translate(image, steering_angle, range_x, range_y):
    """
    Randomly shift the image virtially and horizontally (translation).
    """
    trans_x = range_x * (np.random.rand() - 0.5)
    trans_y = range_y * (np.random.rand() - 0.5)
    steering_angle += trans_x * 0.002
    trans_m = np.float32([[1, 0, trans_x], [0, 1, trans_y]])
    height, width = image.shape[:2]
    image = cv2.warpAffine(image, trans_m, (width, height))
    return image, steering_angle


def random_shadow(image):
    """
    Generates and adds random shadow
    """
    # (x1, y1) and (x2, y2) forms a line
    # xm, ym gives all the locations of the image
    x1, y1 = IMAGE_WIDTH * np.random.rand(), 0
    x2, y2 = IMAGE_WIDTH * np.random.rand(), IMAGE_HEIGHT
    # could this be a bug?
    xm, ym = np.mgrid[0:IMAGE_HEIGHT, 0:IMAGE_WIDTH]
    # xm, ym = np.mgrid[0:IMAGE_WIDTH, 0:IMAGE_HEIGHT]

    # mathematically speaking, we want to set 1 below the line and zero otherwise
    # Our coordinate is up side down.  So, the above the line:
    # (ym-y1)/(xm-x1) > (y2-y1)/(x2-x1)
    # as x2 == x1 causes zero-division problem, we'll write it in the below form:
    # (ym-y1)*(x2-x1) - (y2-y1)*(xm-x1) > 0
    mask = np.zeros_like(image[:, :, 1])
    mask[(ym - y1) * (x2 - x1) - (y2 - y1) * (xm - x1) > 0] = 1

    # choose which side should have shadow and adjust saturation
```

```
    cond = mask == np.random.randint(2)
    s_ratio = np.random.uniform(low=0.2, high=0.5)

    # adjust Saturation in HLS(Hue, Light, Saturation)
    hls = cv2.cvtColor(image, cv2.COLOR_RGB2HLS)
    hls[:, :, 1][cond] = hls[:, :, 1][cond] * s_ratio
    return cv2.cvtColor(hls, cv2.COLOR_HLS2RGB)


def random_brightness(image):
    """
    Randomly adjust brightness of the image.
    """
    # HSV (Hue, Saturation, Value) is also called HSB ('B' for Brightness).
    hsv = cv2.cvtColor(image, cv2.COLOR_RGB2HSV)
    ratio = 1.0 + 0.4 * (np.random.rand() - 0.5)
    hsv[:,:,2] =  hsv[:,:,2] * ratio
    return cv2.cvtColor(hsv, cv2.COLOR_HSV2RGB)


def augment(image, steering_angle, range_x=100, range_y=10):
    """
    Generate an augumented image and adjust steering angle.
    (The steering angle is associated with the center image)
    """
    # resize - we start with assumed image capture size
    image = cv2.resize(image, (320,160), cv2.INTER_AREA)
    # image = cv2.resize(image, (IMAGE_WIDTH, IMAGE_HEIGHT), cv2.INTER_AREA)
    image, steering_angle = random_flip(image, steering_angle)
    image, steering_angle = random_translate(image, steering_angle, range_x, range_y)
    image = random_shadow(image)
    image = random_brightness(image)
    return image, steering_angle


def batch_generator(data_dir, image_paths, steering_angles, batch_size, is_training):
    """
    Generate training image give image paths and associated steering angles
    """
    images = np.empty([batch_size, IMAGE_HEIGHT, IMAGE_WIDTH, IMAGE_CHANNELS])
    steers = np.empty(batch_size)
    while True:
        i = 0
        for index in np.random.permutation(image_paths.shape[0]):
            center, left, right = image_paths[index]
            steering_angle = steering_angles[index]
            # argumentation
            if is_training and np.random.rand() < 0.6:
                image, steering_angle = augument(data_dir, center, left, right, steering_angle)
            else:
                image = load_image(data_dir, center)
            # add the image and steering angle to the batch
            images[i] = preprocess(image)
            steers[i] = steering_angle
```

```
            i += 1
            if i == batch_size:
                break
        yield images, steers


# In[5]:


# Steering angle predictions by model 20201107210627_nvidia1.h5
def plotSteering(p,g,n):
    """
    Plot
    Inputs
        p: array of floats, steering angle prediction
        g: array of floats, steering angle ground truth.
        n: float, normalization constant
    Output
    """
    import matplotlib.pyplot as plt

    plt.rcParams["figure.figsize"] = (18,3)

    plt.plot(p*n, label="model")
    plt.plot(g*n, label="simulator")

    plt.ylabel('Steering angle')
    # Set a title of the current axes.
    plt.title('Predicted and actual steering angles: SDSandbox simulator and model 20201107210627_nvidia1.h5')
    # show a legend on the plot
    plt.legend()
    # Display a figure.
    # horizontal grid only
    plt.grid(axis='y')
    # set limit
    plt.xlim([-5,len(sarr)+5])
    # invert axis so if seen sideways plot corresponds to direction of steering wheel
    plt.gca().invert_yaxis()
    plt.show()

p = sarr[:,0]
g = sarr[:,1]
n = 25 # 25 frames per second
plotSteering(p,g,n)


# $ g_s(p,g) = \frac{\sum_i^N \lvert p(i)-g(i) \rvert }{N} \times n_c $

# In[2]:


def gos(p, g, n):
    """
    Calculate the goodness-of-steer between a prediction and a ground truth array.
```

```
    Inputs
        p: array of floats, steering angle prediction
        g: array of floats, steering angle ground truth.
        n: float, normalization constant
    Output
        gos: float, average of absolute difference between ground truth and prediction arrays
    """
    # todo add type assertion
    assert len(p) == len(g), "Arrays must be of equal length"
    return sum(abs(p - g)) / len(p) * n

p = sarr[:,0]
g = sarr[:,1]

sterr = gos(p,g, 25)

print("Goodness of steer: {:.2f}".format(sterr))


# In[2]:


import os
path = '~/git/sdsandbox/dataset/ford/2017-08-04-V2-Log1-Center'
for root, dirnames, filenames in os.walk(path):
    print(os.path.join(root, filename))


# In[ ]:




#########################################################
# 11. MakeVideo.py
#########################################################


import argparse
import fnmatch
import json
import os
from io import BytesIO
from PIL import Image
import base64
import numpy as np
import matplotlib.pyplot as plt
from augmentation import preprocess
import cv2
import conf
# import debug.RecordVideo as RecordVideo
```

```
def MakeVideo(filename, model, preproc=False):
    """
    Make video from tcpflow logged images.
    video.avi is written to disk
    Inputs
        filename: string, name of tcpflow log
        model: name of model to stamp onto video
        preproc: boolean, show preprocessed image next to original
    Output
        none
    """
    # video name
    video_name = model + '.avi'
    VIDEO_WIDTH, VIDEO_HEIGHT = 800, 600
    IMAGE_WIDTH, IMAGE_HEIGHT = 800, 600
    if(preproc == True): # wide angle
        VIDEO_WIDTH = IMAGE_WIDTH*2
    video = cv2.VideoWriter(video_name, 0, 11, (VIDEO_WIDTH, VIDEO_HEIGHT)) # assumed 11fps
    # font
    font = cv2.FONT_HERSHEY_SIMPLEX

    # normalization constant
    # open file
    sa = []
    # initialize prediction
    pred = ''
    f = open(filename, "r")
    file = f.read()
    try:
        #readline = f.read()
        lines = file.splitlines()
        for line in lines:
            #print(line)
            start = line.find('{')
            if(start == -1):
                continue
            jsonstr = line[start:]
            #print(jsonstr)
            jsondict = json.loads(jsonstr)
            if "steering" in jsondict:
                # predicted
                pred = jsondict['steering']
            if "steering_angle" in jsondict:
                # actual
                act = jsondict['steering_angle']
                # save pair, only keep last pred in case two were send as it does happen i.e.:
                # 127.000.000.001.59460-127.000.000.001.09091: {"msg_type": "control", "steering": "-0.071960375", "t
hrottle": "0.08249988406896591", "brake": "0.0"}
                # 127.000.000.001.59460-127.000.000.001.09091: {"msg_type": "control", "steering": "-0.079734944", "t
hrottle": "0.08631626516580582", "brake": "0.0"}
                # 127.000.000.001.09091-127.000.000.001.59460: {"msg_type":"telemetry","steering_angle":-0.0719603
7,(...)
                if(len(pred) > 0):
                    # save steering angles
```

```
                sa.append([float(pred), act])
                pred = '' # need to save this image
                # process image
                imgString = jsondict["image"]
                # decode string
                image = Image.open(BytesIO(base64.b64decode(imgString)))
                # try to convert to jpg
                #image = np.array(image) # sky colour turns orange (TODO investigate)
                # save
                image.save('frame.jpg')
                # reopen with user-friendlier cv2
                image = cv2.imread('frame.jpg') # 120x160x3
                image_copy = image
                # resize so we can write some info onto image
                image = cv2.resize(image, (IMAGE_WIDTH, IMAGE_HEIGHT), cv2.INTER_AREA)
                # add Info to frame
                cv2.putText(image, model, (50, 50), font, 1, (255, 255, 255), 2, cv2.LINE_AA)
                # Predicted steering angle
                pst = sa[len(sa)-1][0]
                pst *= conf.norm_const
                simst = "Predicted steering angle: {:.2f}".format(pst)
                cv2.putText(image, simst, (50, 115), font, 1, (255, 255, 255), 2, cv2.LINE_AA)
                # create a preprocessed copy to compare what simulator generates to what network "sees"
                if (preproc == True):  # wide angle
                    image2 = preprocess(image_copy)
                    image2 = cv2.resize(image2, (IMAGE_WIDTH, IMAGE_HEIGHT), cv2.INTER_AREA)
                    cv2.putText(image2, 'Network Image', (50, 50), font, 1, (255, 255, 255), 2, cv2.LINE_AA)
                # concatenate
                if (preproc == True):  # wide angle
                    cimgs = np.concatenate((image, image2), axis=1)
                    image = cimgs
                # model name
                # model
                video.write(image);
                pred = ''
    except Exception as e:
        print("Exception raise: " + str(e))
    # file should be automatically closed but will close for good measure
    f.close()
    cv2.destroyAllWindows()
    video.release()


    return "DummyName.mp4"
if __name__ == "__main__":
    parser = argparse.ArgumentParser(description='Make Video script')
    parser.add_argument('--filename', type=str, help='tcpflow log')
    parser.add_argument('--model', type=str, help='model name for video label')
    args = parser.parse_args()
    MakeVideo(args.filename, args.model, True)
    # example
    # python MakeVideo.py --filename=/tmp/tcpflow.log --model=20201120184912_sanity.h5
```

```
#####################################################
# 12. mech_turk.py
#####################################################


import os
import fnmatch
from shutil import copyfile

def MechTurkBatch(filemask, op):
    files = 4445 # we want 50 uniformly distributed files
    mod = 88 # 445 / 50
    count = 1
    filemask = os.path.expanduser(filemask)
    path, mask = os.path.split(filemask)

    #make path
    os.makedirs(path + op, exist_ok=True)
    for root, dirnames, filenames in os.walk(path):
        for filename in fnmatch.filter(filenames, mask):
            # matches.append(os.path.join(root, filename))

            if(count % mod == 0):
                src = os.path.join(root, filename)
                dst = root + '/out1/' + filename
                #print("cp " + source + "./mech_turk-v2-1")
                copyfile(src, dst)
            count += 1




if __name__ == "__main__":
    path = '../../dataset/ford/2017-08-04-V2-Log1-Center/*.png'
    op = 'out1'
    MechTurkBatch(path, op)

#####################################################
# 13. models.py
# Note: code based on
# https://github.com/tawnkramer/sdsandbox/blob/master/src/models.py
# Available for audit in audit_files/tawn from sharepoint link
# and
# https://github.com/naokishibuya/car-behavioral-cloning/blob/master/model.py
# Available for audit in audit_files/naoki from sharepoint link

#####################################################


'''
Models
Define the different NN models we will use
Author: Tawn Kramer
'''
```

```
from __future__ import print_function
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Input
from tensorflow.keras.layers import Dense, Lambda, ELU
from tensorflow.keras.layers import Activation, Dropout, Flatten, Dense, BatchNormalization
from tensorflow.keras.layers import Cropping2D
from tensorflow.keras.optimizers import Adadelta, Adam, SGD
from tensorflow.keras import initializers, regularizers
# Alexnet


import conf

def show_model_summary(model):
    model.summary()
    for layer in model.layers:
        print(layer.output_shape)

def nvidia_baseline(num_outputs):
    '''
    this model is approximately equal to:
    https://images.nvidia.com/content/tegra/automotive/images/2016/solutions/pdf/end-to-end-dl-using-px.pdf
    Although nothing is said about dropout or activation, which is assumed to be RELU

    Hi Daniel,

    We used the following settings (we haven't documented them in any publication):

    loss function: MSE
    optimizer: adadelta
    learning rate: 1e-4 (but not really used in adadelta)
    dropout: 0.25

    Best regards,
    Urs
    '''
    # note row and col values are now from _NET
    # Adjust sizes accordingly in conf.py
    row, col, ch = conf.row, conf.col, conf.ch

    drop = 0.1 # spreading droupout
    # batch_init = initializers.glorot_uniform #  Original AlexNet initializers.RandomNormal(mean=0., stddev=0.01);
    img_in = Input(shape=(row, col, ch), name='img_in')
    x = img_in
    # RGB values assumed to be normalized and not centered i.e. x/127.5 - 1.
    x = Lambda(lambda x: x / 255.0)(x)
    x = Conv2D(24, (5, 5), strides=(2, 2), activation='relu', name="conv2d_1")(x)
    x = Dropout(drop)(x)
    x = Conv2D(36, (5, 5), strides=(2, 2), activation='relu', name="conv2d_2")(x) #2nd
    x = Dropout(drop)(x)
    x = Conv2D(48, (5, 5), strides=(2, 2), activation='relu', name="conv2d_3")(x)
    x = Dropout(drop)(x)
    x = Conv2D(64, (3, 3), strides=(1, 1), activation='relu', name="conv2d_4")(x) # default strides=(1,1) # 4th
    x = Dropout(drop)(x)
```

```
    x = Conv2D(64, (3, 3), strides=(1, 1), activation='relu', name="conv2d_5")(x) #5th
    x = Dropout(drop)(x)
    x = Flatten(name='flattened')(x)
    # x = Dense(1164, activation='relu', name="dense_1", kernel_initializer=batch_init, bias_initializer='ones')(x)
    #x = Dropout(drop)(x)
    x = Dense(100, activation='relu', name="dense_2")(x)
    #x = Dropout(drop)(x)
    x = Dense(50, activation='relu', name="dense_3")(x) # Added in Naoki's model
    #x = Dropout(drop)(x)
    # x = Dense(10, activation='relu', name="dense_4", kernel_initializer=batch_init, bias_initializer='zeros')(x)
    #x = Dropout(drop)(x)
    outputs = []
    # outputs.append(Dense(num_outputs, activation='linear', name='steering_throttle')(x))
    outputs.append(Dense(num_outputs, activation='linear', name='steering')(x))

    model = Model(inputs=[img_in], outputs=outputs)
    # opt = Adadelta(learning_rate=0.001, rho=0.95, epsilon=1e-07, name="Adadelta")
    opt = Adam(lr=conf.learning_rate)
    model.compile(optimizer=opt, loss="mse", metrics=['acc'])

    # add weight decay
    # https://stackoverflow.com/questions/41260042/global-weight-decay-in-keras
    #alpha = 0.0005  # weight decay coefficient
    #for layer in model.layers:
    #    if isinstance(layer, Conv2D) or isinstance(layer, Dense):
    #        layer.add_loss(lambda: regularizers.l2(alpha)(layer.kernel))
    #    if hasattr(layer, 'bias_regularizer') and layer.use_bias:
    #        layer.add_loss(lambda: regularizers.l2(alpha)(layer.bias))
    return model

def nvidia_model1(num_outputs):
    '''
    This model expects image input size 160hx120w
    this model is inspired by the NVIDIA paper
    https://images.nvidia.com/content/tegra/automotive/images/2016/solutions/pdf/end-to-end-dl-using-px.pdf
    Activation is RELU
    '''
    # row, col, ch = conf.row, conf.col, conf.ch
    # better albeit less readable
    row, col, ch = conf.nvidia1_img_dims[conf.IMG_HEIGHT_NET_IDX], conf.nvidia1_img_dims[conf.IMG_WID
TH_NET_IDX], conf.nvidia1_img_dims[conf.IMG_DEPTH_IDX]
    drop = 0.1

    img_in = Input(shape=(row, col, ch), name='img_in')
    x = img_in
    # x = Cropping2D(cropping=((10,0), (0,0)))(x) #trim 10 pixels off top
    # x = Lambda(lambda x: x/127.5 - 1.)(x) # normalize and re-center
    x = Lambda(lambda x: x/255.0)(x)
    x = Conv2D(24, (5,5), strides=(2,2), activation='relu', name="conv2d_1")(x)
    x = Dropout(drop)(x)
    x = Conv2D(32, (5,5), strides=(2,2), activation='relu', name="conv2d_2")(x)
    x = Dropout(drop)(x)
    x = Conv2D(64, (5,5), strides=(2,2), activation='relu', name="conv2d_3")(x)
    x = Dropout(drop)(x)
```

```
    x = Conv2D(64, (3,3), strides=(1,1), activation='relu', name="conv2d_4")(x)
    x = Dropout(drop)(x)
    x = Conv2D(64, (3,3), strides=(1,1), activation='relu', name="conv2d_5")(x)
    x = Dropout(drop)(x)

    x = Flatten(name='flattened')(x)
    x = Dense(1064, activation='relu')(x)
    x = Dense(100, activation='relu')(x)
    #x = Dropout(drop)(x)
    x = Dense(50, activation='relu')(x)
    #x = Dropout(drop)(x)
    x = Dense(10, activation='relu')(x)

    outputs = []
    outputs.append(Dense(num_outputs, activation='linear', name='steering_throttle')(x))


    model = Model(inputs=[img_in], outputs=outputs)
    opt = Adam(lr=0.0001)
    model.compile(optimizer=opt, loss="mse", metrics=['acc'])
    # might want to try metrics=['acc', 'loss']  https://stackoverflow.com/questions/51047676/how-to-get-accuracy-of-
model-using-keras
    return model


def nvidia_model2(num_outputs):
    '''
    A.K.A. NaokiNet - https://github.com/naokishibuya/car-behavioral-cloning/blob/master/model.py
    This model expects images of size 66,200,3
    '''
    # row, col, ch = conf.row, conf.col, conf.ch
    row, col, ch = conf.nvidia2_img_dims[conf.IMG_HEIGHT_NET_IDX], conf.nvidia2_img_dims[conf.IMG_WID
TH_NET_IDX], \
                conf.nvidia2_img_dims[conf.IMG_DEPTH_IDX]

    drop = 0.1

    img_in = Input(shape=(row, col, ch), name='img_in')
    x = img_in
    # x = Cropping2D(cropping=((10,0), (0,0)))(x) #trim 10 pixels off top
    # x = Lambda(lambda x: x/127.5 - 1.0)(x) # normalize and re-center
    x = Lambda(lambda x: x / 255.0)(x)
    x = Conv2D(24, (5, 5), strides=(2, 2), activation='elu', name="conv2d_1")(x)
    x = Dropout(drop)(x)
    x = Conv2D(32, (5, 5), strides=(2, 2), activation='elu', name="conv2d_2")(x)
    x = Dropout(drop)(x)
    x = Conv2D(48, (5, 5), strides=(2, 2), activation='elu', name="conv2d_3")(x)
    x = Dropout(drop)(x)
    x = Conv2D(64, (3, 3), activation='elu', name="conv2d_4")(x) # default strides=(1,1)
    x = Dropout(drop)(x)
    x = Conv2D(64, (3, 3), activation='elu', name="conv2d_5")(x)
    x = Dropout(drop)(x)

    x = Flatten(name='flattened')(x)
```

```
    x = Dense(100, activation='elu')(x)
    # x = Dropout(drop)(x)
    x = Dense(50, activation='elu')(x)
    # x = Dropout(drop)(x)
    x = Dense(10, activation='elu')(x) # Added in Naoki's model

    outputs = []
    # outputs.append(Dense(num_outputs, activation='linear', name='steering_throttle')(x))
    outputs.append(Dense(num_outputs, name='steering_throttle')(x))

    model = Model(inputs=[img_in], outputs=outputs)
    opt = Adam(lr=0.0001)
    model.compile(optimizer=opt, loss="mse", metrics=['acc'])
    return model

def nvidia_model3(num_outputs):
    '''
    This model expects images of size 66,200,3
    '''
    row, col, ch = conf.row, conf.col, conf.ch

    drop = 0.1

    img_in = Input(shape=(row, col, ch), name='img_in')
    x = img_in
    # x = Cropping2D(cropping=((10,0), (0,0)))(x) #trim 10 pixels off top
    x = Lambda(lambda x: x/127.5 - 1.0)(x) # normalize and re-center
    # x = Lambda(lambda x: x / 255.0)(x)
    x = Conv2D(24, (5, 5), strides=(2, 2), activation='elu', name="conv2d_1")(x)
    # x = Dropout(drop)(x)
    x = Conv2D(36, (5, 5), strides=(2, 2), activation='elu', name="conv2d_2")(x)
    #x = Dropout(drop)(x)
    x = Conv2D(48, (5, 5), strides=(2, 2), activation='elu', name="conv2d_3")(x)
    #x = Dropout(drop)(x)
    x = Conv2D(64, (3, 3), activation='elu', name="conv2d_4")(x) # default strides=(1,1)
    #x = Dropout(drop)(x)
    x = Conv2D(64, (3, 3), activation='elu', name="conv2d_5")(x)
    x = Dropout(drop)(x)

    x = Flatten(name='flattened')(x)

    x = Dense(100, activation='elu')(x)
    # x = Dropout(drop)(x)
    x = Dense(50, activation='elu')(x)
    # x = Dropout(drop)(x)
    x = Dense(10, activation='elu')(x) # Added in Naoki's model

    outputs = []
    # outputs.append(Dense(num_outputs, activation='linear', name='steering_throttle')(x))
    outputs.append(Dense(num_outputs, name='steering_throttle')(x))

    model = Model(inputs=[img_in], outputs=outputs)
    opt = Adam(lr=0.0001)
```

```
    model.compile(optimizer=opt, loss="mse", metrics=['acc'])
    return model

def get_alexnet(num_outputs):
    '''
    this model is also inspired by the NVIDIA paper
    https://images.nvidia.com/content/tegra/automotive/images/2016/solutions/pdf/end-to-end-dl-using-px.pdf
    but taken from
    https://github.com/naokishibuya/car-behavioral-cloning/blob/master/model.py
    model = Sequential()
    model.add(Lambda(lambda x: x/127.5-1.0, input_shape=INPUT_SHAPE))
    model.add(Conv2D(24, 5, 5, activation='elu', strides=2))
    model.add(Conv2D(36, 5, 5, activation='elu', strides=2))
    model.add(Conv2D(48, 5, 5, activation='elu', strides=2))
    model.add(Conv2D(64, 3, 3, activation='elu'))
    model.add(Conv2D(64, 3, 3, activation='elu'))
    model.add(Dropout(args.keep_prob))
    model.add(Flatten())
    model.add(Dense(100, activation='elu'))
    model.add(Dense(50, activation='elu'))
    model.add(Dense(10, activation='elu'))
    model.add(Dense(1))
    model.summary()
    NB Tawn Kramer's model uses dropout = 0.1 on five layers, Naoki uses
    0.5 on a single layer
    '''
    #row, col, ch = conf.image_width_alexnet, conf.image_height_alexnet, conf.ch
    row, col, ch = conf.alexnet_img_dims[conf.IMG_HEIGHT_NET_IDX], conf.nvidia2_img_dims[conf.IMG_WID
TH_NET_IDX], \
    conf.nvidia2_img_dims[conf.IMG_DEPTH_IDX]
    drop = 0.5
    # read https://stackoverflow.com/questions/58636087/tensorflow-valueerror-failed-to-convert-a-numpy-array-to-a
-tensor-unsupporte
    # to work out shapes
    img_in = Input(shape=(row, col, ch), name='img_in')
    x = img_in
    # x = Cropping2D(cropping=((10,0), (0,0)))(x) #trim 10 pixels off top
    x = Lambda(lambda x: x/127.5 - 1.0)(x) # normalize and re-center
    # x = Lambda(lambda x: x / 255.0)(x)
    x = Conv2D(48, (8, 8), strides=(4, 4), padding='valid', activation='relu', name="conv2d_1")(x)
    # x = Dropout(drop)(x)
    x = MaxPooling2D(48, (1, 1), padding="same", name="maxpool2d_1")(x)

    x = Conv2D(128, (3, 3), strides=(2, 2), padding='valid', activation='relu', name="conv2d_2")(x)
    #x = Dropout(drop)(x)
    x = MaxPooling2D(128, (1, 1), padding="same", name="maxpool2d_2")(x)

    x = Conv2D(192, (3, 3), strides=(2, 2), padding='valid', activation='relu', name="conv2d_3")(x)
    #x = Dropout(drop)(x)
    x = Conv2D(192, (3, 3), strides=(1, 1), padding='same', activation='relu', name="conv2d_4")(x) # default strides=
(1,1)
    #x = Dropout(drop)(x)

    x = Conv2D(128, (3, 3), strides=(1, 1), padding='same', activation='relu', name="conv2d_5")(x)
```

```
    x = MaxPooling2D(128, (1, 1), padding="same", name="maxpool2d_3")(x)

    #x = Conv2D(64, (3, 3), activation='relu', name="conv2d_6")(x)
    # error Negative dimension size caused by subtracting 128 from 10 for '{{node max_pooling2d/MaxPool}} = Ma
xPool[T=DT_FLOAT,
    # data_format="NHWC", ksize=[1, 128, 128, 1], padding="VALID", strides=[1, 3, 3, 1]](conv2d_4/Identity)'
    # with input shapes: [?,10,10,192].
    # x = MaxPooling2D(128, (3, 3), padding="SAME")(x)
    # By commenting out line above, error is:
    #  Input to reshape is a tensor with 1843200 values, but the requested shape requires a multiple of 101568
    #   [[node model/flattened/Reshape (defined at /git/sdsandbox/src/train.py:272) ]] [Op:__inference_train_function
_1081]

    x = Dropout(drop)(x)

    """
        x = Conv2D(24, (5, 5), strides=(2, 2), activation='elu', name="conv2d_1")(x)
    # x = Dropout(drop)(x)
    x = Conv2D(36, (5, 5), strides=(2, 2), activation='elu', name="conv2d_2")(x)
    #x = Dropout(drop)(x)
    x = Conv2D(64, (5, 5), strides=(2, 2), activation='elu', name="conv2d_3")(x)
    #x = Dropout(drop)(x)
    x = Conv2D(64, (3, 3), activation='elu', name="conv2d_4")(x) # default strides=(1,1)
    #x = Dropout(drop)(x)
    x = MaxPooling2D(64, (3, 3), name="maxpool2d_5")(x)
    x = Dropout(drop)(x)

    x = Flatten(name='flattened')(x)

    x = Dense(2048, activation='elu')(x)
    # x = Dropout(drop)(x)
    x = Dense(2048, activation='elu')(x)
    # x = Dropout(drop)(x)
    # x = Dense(10, activation='elu')(x) # Added in Naoki's model
    """


    x = Flatten(name='flattened')(x) # error when followed by

    x = Dense(2048, name='Dense_1', activation='relu')(x)# 2048, 2048 ~  Input to reshape is a tensor with 442368 va
lues, but the requested shape requires a multiple of 21632
  # [[node model/flattened/Reshape (defined at /git/sdsandbox/src/train.py:272) ]] [Op:__inference_train_function_1
192]

    # x = Dropout(drop)(x)
    x = Dense(50, name='Dense_2', activation='elu')(x)
    # x = Dropout(drop)(x)
    x = Dense(10, activation='elu')(x) # Added in Naoki's model

    outputs = []
    # outputs.append(Dense(num_outputs, activation='linear', name='steering_throttle')(x))
    outputs.append(Dense(num_outputs, name='steering_throttle')(x))

    model = Model(inputs=[img_in], outputs=outputs)
```

```
    opt = Adam(lr=0.0001)
    model.compile(optimizer=opt, loss="mse", metrics=['acc'])
    return model
"""
def alexnet_model(img_shape=(224, 224, 3), n_classes=10, l2_reg=0.,
 weights=None):

 # Initialize model
 alexnet = Sequential()

 # Layer 1
 alexnet.add(Conv2D(96, (11, 11), input_shape=img_shape,
  padding='same', kernel_regularizer=l2(l2_reg)))
 alexnet.add(BatchNormalization())
 alexnet.add(Activation('relu'))
 alexnet.add(MaxPooling2D(pool_size=(2, 2)))

 # Layer 2
 alexnet.add(Conv2D(256, (5, 5), padding='same'))
 alexnet.add(BatchNormalization())
 alexnet.add(Activation('relu'))
 alexnet.add(MaxPooling2D(pool_size=(2, 2)))

 # Layer 3
 alexnet.add(ZeroPadding2D((1, 1)))
 alexnet.add(Conv2D(512, (3, 3), padding='same'))
 alexnet.add(BatchNormalization())
 alexnet.add(Activation('relu'))
 alexnet.add(MaxPooling2D(pool_size=(2, 2)))

 # Layer 4
 alexnet.add(ZeroPadding2D((1, 1)))
 alexnet.add(Conv2D(1024, (3, 3), padding='same'))
 alexnet.add(BatchNormalization())
 alexnet.add(Activation('relu'))

 # Layer 5
 alexnet.add(ZeroPadding2D((1, 1)))
 alexnet.add(Conv2D(1024, (3, 3), padding='same'))
 alexnet.add(BatchNormalization())
 alexnet.add(Activation('relu'))
 alexnet.add(MaxPooling2D(pool_size=(2, 2)))

 # Layer 6
 alexnet.add(Flatten())
 alexnet.add(Dense(3072))
 alexnet.add(BatchNormalization())
 alexnet.add(Activation('relu'))
 alexnet.add(Dropout(0.5))

 # Layer 7
 alexnet.add(Dense(4096))
 alexnet.add(BatchNormalization())
 alexnet.add(Activation('relu'))
```

```
 alexnet.add(Dropout(0.5))

 # Layer 8
 alexnet.add(Dense(n_classes))
 alexnet.add(BatchNormalization())
 alexnet.add(Activation('softmax'))

 if weights is not None:
  alexnet.load_weights(weights)

 return alexnet
"""

#######################################################
# 14. predict_client.py
# Note: code based on
# https://github.com/tawnkramer/sdsandbox/blob/master/src/predict_client.py
# Available for audit in audit_files/tawn from sharepoint link
#######################################################


'''
Predict Server
Create a server to accept image inputs and run them against a trained neural network.
This then sends the steering output back to the client.
Author: Tawn Kramer
'''
from __future__ import print_function
import os
import sys
import argparse
import time
import json
import base64
import datetime
from io import BytesIO
import signal
import tensorflow as tf
from tensorflow.python import keras
from tensorflow.python.keras.models import load_model
from PIL import Image
import numpy as np
from gym_donkeycar.core.fps import FPSTimer
from gym_donkeycar.core.message import IMesgHandler
from gym_donkeycar.core.sim_client import SimClient
# same preprocess as for training
from augmentation import augment, preprocess
import conf
from helper_functions import parse_bool
import utils.RecordVideo as RecordVideo
import Augmentation


if tf.__version__ == '1.13.1':
```

```
    from tensorflow import ConfigProto, Session

    # Override keras session to work around a bug in TF 1.13.1
    # Remove after we upgrade to TF 1.14 / TF 2.x.
    config = ConfigProto()
    config.gpu_options.allow_growth = True
    session = Session(config=config)
    keras.backend.set_session(session)

# need to import file TODO
import Automold as am
import Helpers as hp
import numpy as np
# helper function for prediction

def add_rain(image_arr, rt=None, st=0):
    """
    Add rain to image
    Inputs:
        image_arr: numpy array containing image
        rt: string, rain type "heavy" or "torrential"
        st: range to draw a random slant from
    Output
        image_arr: numpy array containing image with rain
    """
    # print("Adding rain...")
    if(st != 0):
        # draw a random number for slant
        st = np.random.randint(-1 * st, st)

    if(rt!='light'): # heavy or torrential
        image_arr = am.add_rain_single(image_arr, rain_type=rt, slant=st)
    else:
         # no slant
        image_arr = am.add_rain_single(image_arr)

    return image_arr

class DonkeySimMsgHandler(IMesgHandler):

    STEERING = 0
    THROTTLE = 1

    def __init__(self, model, constant_throttle, image_cb=None, rand_seed=0):
        self.model = model
        self.constant_throttle = constant_throttle
        self.client = None
        self.timer = FPSTimer()
        self.img_arr = None
        self.image_cb = image_cb
        self.steering_angle = 0.
        self.throttle = 0.
        self.rand_seed = rand_seed
        self.fns = {'telemetry' : self.on_telemetry,\
```

```
            'car_loaded' : self.on_car_created,\
            'on_disconnect' : self.on_disconnect,
            'aborted' : self.on_aborted}
    # images to record
    self.img_orig = None
    self.img_add_rain = None
    self.img_processed = None
    self.frame_count = 0

    # model name


def on_connect(self, client):
    self.client = client
    self.timer.reset()

def on_aborted(self, msg):
    self.stop()

def on_disconnect(self):
    pass

def on_recv_message(self, message):
    self.timer.on_frame()
    if not 'msg_type' in message:
        print('expected msg_type field')
        print("message:", message)
        return

    msg_type = message['msg_type']
    if msg_type in self.fns:
        self.fns[msg_type](message)
    else:
        print('unknown message type', msg_type)

def on_car_created(self, data):
    if self.rand_seed != 0:
        self.send_regen_road(0, self.rand_seed, 1.0)

def on_telemetry(self, data):
    imgString = data["image"]
    image = Image.open(BytesIO(base64.b64decode(imgString)))
    img_arr = np.asarray(image, dtype=np.float32)
    self.frame_count += 1
    self.img_orig = img_arr

    # set to same image size expected from acquisition process
    img_arr = ag.resize_expected(img_arr)

    # check for rain
    if(conf.rt != ''):
        img_arr = add_rain(img_arr, conf.rt, conf.st)
        self.img_add_rain = img_arr
```

```
    # same preprocessing as for training
    img_arr = ag.preprocess(img_arr)
    self.img_processed = img_arr

    #if(conf.record == True):
    #    text = (['Network Image', 'No Rain'])
    #    rv.add_image(img_arr, text)

    # if we are testing the network with rain
    self.img_arr = img_arr.reshape((1,) + img_arr.shape)

    if self.image_cb is not None:
        self.image_cb(img_arr, self.steering_angle )

def update(self):
    if self.img_arr is not None:
        self.predict(self.img_arr)
        self.img_arr = None

def predict(self, image_array):
    outputs = self.model.predict(image_array)
    # check if we are recording
    if (conf.record == True):

        # Add first image, with name of network and frame number
        # TODO, get network name from argument
        text = ([rv.modelname, 'Intensity Multiplie: 1', 'Acquired image', 'Frame:  ' + str(self.frame_count)])
        rv.add_image(self.img_orig, text)

        # Add second image, preprocessed with rain or without
        # text = (['Network image', 'No rain'])
        # rv.add_image(self.img_processed, text)
        # if rain added
        # check for rain
        if (conf.rt != ''):
            rtype = 'Type: ' + conf.rt
            s = 'Slant: -+' + str(conf.st)
            text = (['Added rain', rtype, s])
            rv.add_image(self.img_add_rain, text)

            # add third image with prediction
        steering = outputs[0][0]
        steering *= conf.norm_const
        st_str = "{:.2f}".format(steering)
        st_str = "Predicted steering angle: " + st_str
        # st_str = "Predicted steering angle: 20"
        #rtype = 'Type: ' + conf.rt
        #s = 'Slant: -+' + str(conf.st)
        text = (["Network image", st_str])
        rv.add_image(image_array[0], text)
        rv.add_frame()
    self.parse_outputs(outputs)
```

```
    def parse_outputs(self, outputs):
        res = []

        # Expects the model with final Dense(2) with steering and throttle
        for i in range(outputs.shape[1]):
            res.append(outputs[0][i])

        self.on_parsed_outputs(res)

    def on_parsed_outputs(self, outputs):
        self.outputs = outputs
        self.steering_angle = 0.0
        self.throttle = 0.2

        if len(outputs) > 0:
            self.steering_angle = outputs[self.STEERING]

        if self.constant_throttle != 0.0:
            self.throttle = self.constant_throttle
        elif len(outputs) > 1:
            self.throttle = outputs[self.THROTTLE] * conf.throttle_out_scale

        self.send_control(self.steering_angle, self.throttle)

    def send_control(self, steer, throttle):
        # print("send st:", steer, "th:", throttle)
        msg = { 'msg_type' : 'control', 'steering': steer.__str__(), 'throttle':throttle.__str__(), 'brake': '0.0' }
        self.client.queue_message(msg)

    def send_regen_road(self, road_style=0, rand_seed=0, turn_increment=0.0):
        '''
        Regenerate the road, where available. For now only in level 0.
        In level 0 there are currently 5 road styles. This changes the texture on the road
        and also the road width.
        The rand_seed can be used to get some determinism in road generation.
        The turn_increment defaults to 1.0 internally. Provide a non zero positive float
        to affect the curviness of the road. Smaller numbers will provide more shallow curves.
        '''
        msg = { 'msg_type' : 'regen_road',
            'road_style': road_style.__str__(),
            'rand_seed': rand_seed.__str__(),
            'turn_increment': turn_increment.__str__() }

        self.client.queue_message(msg)

    def stop(self):
        self.client.stop()

    def __del__(self):
        self.stop()



def clients_connected(arr):
```

```
    for client in arr:
        if not client.is_connected():
            return False
    return True


def go(filename, address, constant_throttle=0, num_cars=1, image_cb=None, rand_seed=None):

    print("loading model", filename)
    model = load_model(filename)

    # In this mode, looks like we have to compile it
    model.compile("sgd", "mse")

    clients = []

    for _ in range(0, num_cars):
        # setup the clients
        handler = DonkeySimMsgHandler(model, constant_throttle, image_cb=image_cb, rand_seed=rand_seed)
        client = SimClient(address, handler)
        clients.append(client)

    while clients_connected(clients):
        try:
            time.sleep(0.02)
            for client in clients:
                client.msg_handler.update()
        except KeyboardInterrupt:
            # unless some hits Ctrl+C and then we get this interrupt
            print('stopping')
            break

def stop_exec(signum, frame):
    # restore the original signal handler as otherwise evil things will happen
    # in raw_input when CTRL+C is pressed, and our signal handler is not re-entrant
    signal.signal(signal.SIGINT, original_sigint)

    try:
        # changed raw_input to input
        if input("\nFinish recording video? (y/n)> ").lower().startswith('y'):
            print("*** CTRL+C to stop ***")
            rv.save_video()
            sys.exit(1)

    except KeyboardInterrupt:
        print("Ok ok, quitting")
        sys.exit(1)

    # restore the exit gracefully handler here
    signal.signal(signal.SIGINT, stop_exec)

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description='prediction server')
    parser.add_argument('--model', type=str, help='model filename')
```

```
    parser.add_argument('--modelname', type=str, default='nvidia1', help='model filename')
    parser.add_argument('--host', type=str, default='127.0.0.1', help='server sim host')
    parser.add_argument('--port', type=int, default=9091, help='bind to port')
    parser.add_argument('--num_cars', type=int, default=1, help='how many cars to spawn')
    parser.add_argument('--constant_throttle', type=float, default=0.0, help='apply constant throttle')
    parser.add_argument('--rand_seed', type=int, default=0, help='set road generation random seed')
    parser.add_argument('--rain', type=str, default='', help='type of rain [light|heavy|torrential')
    parser.add_argument('--slant', type=int, default=0, help='Rain slant deviation')
    parser.add_argument('--record', type=parse_bool, default="False", help='Record video of raw and processed imag
es')
    # parser.add_argument('--img_cnt', type=int, default=3, help='Number of side by side images to record')


    args = parser.parse_args()
    address = (args.host, args.port)

    conf.rt = args.rain
    conf.st = args.slant
    conf.record = args.record

    ag = Augmentation.Augmentation(args.modelname)

    if conf.record == True:
        print("*** When finished, press CTRL+C and y to finish recording, then CTRL+C to quit ***")
        original_sigint = signal.getsignal(signal.SIGINT)
        signal.signal(signal.SIGINT, stop_exec)
        rv = RecordVideo.RecordVideo(args.model, conf.rt)

    go(args.model, address, args.constant_throttle, num_cars=args.num_cars, rand_seed=args.rand_seed)
    # max value for slant is 20
    # Example
    # python3 predict_client.py --model=../trained_models/sanity/20201120171015_sanity.h5 --rain=light --slant=0



#######################################################
# 15. RecordVideo.py
#######################################################


# https://docs.python.org/3/tutorial/classes.html
import cv2
import conf
import numpy as np


class RecordVideo():
    """
    Record video class, used to record videos from tcpflow log or still images
    """

    def __init__(self, model, rt):
        """
        Record video while running predictions
```

```
        Inputs
            model: string, model name
            rt: type of rain
        """
        model = model.split('/')
        self.modelname = model[-1]
        videoname = self.modelname + '.avi'
        # 3 images side by side in the rain
        img_cnt = 3
        if(rt == ''): # 2 images in the dry
            img_cnt = 2
        self.img_cnt = img_cnt
        self.VIDEO_WIDTH, self.VIDEO_HEIGHT = conf.VIDEO_WIDTH, conf.VIDEO_HEIGHT  # 800, 600
        self.IMAGE_WIDTH, self.IMAGE_HEIGHT = conf.IMAGE_STILL_WIDTH, conf.IMAGE_STILL_HEIGH
T
        self.VIDEO_WIDTH = self.IMAGE_WIDTH * img_cnt
        self.video = cv2.VideoWriter(videoname, 0, 11,
                                     (self.VIDEO_WIDTH, self.VIDEO_HEIGHT))  # assumed 11fps approximately
        self.font = cv2.FONT_HERSHEY_SIMPLEX
        # video line spacing
        self.images = []

    def add_image(self, image, text):
        # self.img_arr_1 =
        # cv2.putText(image, model, (50, 50), font, 1, (255, 255, 255), 2, cv2.LINE_AA)
        # Predicted steering angle
        # pst = sa[len(sa) - 1][0]
        # pst *= conf.norm_const
        # simst = "Predicted steering angle: {:.2f}".format(pst)
        # cv2.putText(image, simst, (50, 115), font, 1, (255, 255, 255), 2, cv2.LINE_AA)
        self.images.append([image, text])

    def add_frame(self):
        y_offset = 50;
        x_offset = 50
        step = 65;
        img_cnt = len(self.images)
        # prepare images
        for i in range(0, img_cnt):
            # resize image
            image = self.images[i][0]
            image = cv2.resize(image, (self.IMAGE_WIDTH, self.IMAGE_HEIGHT), cv2.INTER_AREA)
            # add text
            lines = self.images[i][1]
            for line in lines:
                cv2.putText(image, line, (x_offset, y_offset), self.font, 1, (255, 255, 255), 2, cv2.LINE_AA)
                y_offset += step
                # print(line)
            # set start to top for printing lines in next frame
            y_offset = x_offset
            # store processed image
            self.images[i][0] = image
        # concatenate
        output_image = self.images[0][0]
```

```
        for i in range(1, img_cnt):
            output_image = np.concatenate((output_image, self.images[i][0]), axis=1)
        # append
        try:
            self.video.write(np.uint8(output_image))  # catch error Assertion failed) image.depth() == CV_8U
        except Exception as e:
            print("Exception raise: " + str(e))
        # blank images
        self.images = []

    def save_video(self):
        # save video as videoname
        cv2.destroyAllWindows()
        self.video.release()


#######################################################
# 16. result_plots.py
#######################################################


import numpy as np
from utils.steerlib import GetSteeringFromtcpflow, plotMultipleSteeringAngles

def plot1():
    """
    Intensity multiplier 1, 20201207091932_nvidia1.h5
    """

    # intensity multiplier = 1
    p = []
    # No rain mult 1 (shaw we have a "no rain" for every multiplier ?)
    sa = GetSteeringFromtcpflow('../../trained_models/nvidia1/tcpflow/20201207091932_nvidia1_no_rain_tcpflow.lo
g')
    sarr = np.asarray(sa)
    pa = sarr[:,0]
    p.append([pa, 'no rain'])
    # light mult 1
    sa = GetSteeringFromtcpflow('../../trained_models/nvidia1/tcpflow/20201207091932_nvidia1_light_rain_mult_1_
tcpflow.log')
    sarr = np.asarray(sa)
    pa = sarr[:,0]
    p.append([pa, 'light rain'])
    # heavy mult 1
    sa = GetSteeringFromtcpflow('../../trained_models/nvidia1/tcpflow/20201207091932_nvidia1_heavy_10_mult_1_
tcpflow.log')
    sarr = np.asarray(sa)
    pa = sarr[:,0]
    p.append([pa, 'heavy rain slant +-10'])
    # torrential mult 1
    sa = GetSteeringFromtcpflow('../../trained_models/nvidia1/tcpflow/20201207091932_nvidia1_torrential_20_mult
_1_tcpflow.log')
    sarr = np.asarray(sa)
    pa = sarr[:,0]
```

```
    p.append([pa, 'torrential rain slant +-20'])

    plotMultipleSteeringAngles(p, 25, True, "Generated Track intensity multiplier 1", "20201207091932_nvidia1.h5"
, 'tcpflow log predicted')

def plot2():
    """
    Intensity multiplier 4, 20201207091932_nvidia1.h5
    """
    # intensity multiplier = 4
    p = []
    # No rain mult 1 (shaw we have a "no rain" for every multiplier ?)
    sa = GetSteeringFromtcpflow('../../trained_models/nvidia1/tcpflow/20201207091932_nvidia1_no_rain_tcpflow.lo
g')
    sarr = np.asarray(sa)
    pa = sarr[:, 0]
    p.append([pa, 'no rain'])
    # light mult 1
    sa = GetSteeringFromtcpflow('../../trained_models/nvidia1/tcpflow/20201207091932_nvidia1_light_rain_mult_4_
tcpflow.log')
    sarr = np.asarray(sa)
    pa = sarr[:, 0]
    p.append([pa, 'light rain'])
    # heavy mult 1
    sa = GetSteeringFromtcpflow('../../trained_models/nvidia1/tcpflow/20201207091932_nvidia1_heavy_10_mult_4_
tcpflow.log')
    sarr = np.asarray(sa)
    pa = sarr[:, 0]
    p.append([pa, 'heavy rain slant +-10'])
    # torrential mult 1
    sa = GetSteeringFromtcpflow('../../trained_models/nvidia1/tcpflow/20201207091932_nvidia1_torrential_20_mult
_4_tcpflow.log')
    sarr = np.asarray(sa)
    pa = sarr[:, 0]
    p.append([pa, 'torrential rain slant +-20'])

    plotMultipleSteeringAngles(p, 25, True, "Generated Track intensity multiplier 4", "20201207091932_nvidia1.h5"
,
                               'tcpflow log predicted')

def plot3():
    """
    Intensity multiplier 8, 20201207091932_nvidia1.h5
    """

    # intensity multiplier = 8
    p = []
    # No rain mult 1 (shaw we have a "no rain" for every multiplier ?)
    sa = GetSteeringFromtcpflow('../../trained_models/nvidia1/tcpflow/20201207091932_nvidia1_no_rain_tcpflow.lo
g')
    sarr = np.asarray(sa)
    pa = sarr[:, 0]
    p.append([pa, 'no rain'])
    # light mult 1
```

```
    sa = GetSteeringFromtcpflow(
        '../../trained_models/nvidia1/tcpflow/20201207091932_nvidia1_light_rain_mult_8_tcpflow.log')
    sarr = np.asarray(sa)
    pa = sarr[:, 0]
    p.append([pa, 'light rain'])
    # heavy mult 1
    sa = GetSteeringFromtcpflow(
        '../../trained_models/nvidia1/tcpflow/20201207091932_nvidia1_heavy_10_mult_8_tcpflow.log')
    sarr = np.asarray(sa)
    pa = sarr[:, 0]
    p.append([pa, 'heavy rain slant +-10'])
    # torrential mult 1
    sa = GetSteeringFromtcpflow(
        '../../trained_models/nvidia1/tcpflow/20201207091932_nvidia1_torrential_20_mult_8_tcpflow.log')
    sarr = np.asarray(sa)
    pa = sarr[:, 0]
    p.append([pa, 'torrential rain slant +-20'])

    plotMultipleSteeringAngles(p, 25, True, "Generated Track intensity multiplier 8", "20201207091932_nvidia1.h5"
,
                               'tcpflow log predicted')

def plot4():
    """
    Intensity multiplier 1, 20201207192948_nvidia2.h5
    """

    # intensity multiplier = 1
    p = []
    # No rain mult 1 (shaw we have a "no rain" for every multiplier ?)
    sa = GetSteeringFromtcpflow('../../trained_models/nvidia2/tcpflow/20201207192948_nvidia2_no_rain_tcpflow.lo
g')
    sarr = np.asarray(sa)
    pa = sarr[:,0]
    p.append([pa, 'no rain'])
    # light mult 1
    sa = GetSteeringFromtcpflow('../../trained_models/nvidia2/tcpflow/20201207192948_nvidia2_light_rain_mult_1_
tcpflow.log')
    sarr = np.asarray(sa)
    pa = sarr[:,0]
    p.append([pa, 'light rain'])
    # heavy mult 1
    sa = GetSteeringFromtcpflow('../../trained_models/nvidia2/tcpflow/20201207192948_nvidia2_heavy_10_mult_1_
tcpflow.log')
    sarr = np.asarray(sa)
    pa = sarr[:,0]
    p.append([pa, 'heavy rain slant +-10'])
    # torrential mult 1
    sa = GetSteeringFromtcpflow('../../trained_models/nvidia2/tcpflow/20201207192948_nvidia2_torrential_20_mult
_1_tcpflow.log')
    sarr = np.asarray(sa)
    pa = sarr[:,0]
    p.append([pa, 'torrential rain slant +-20'])
```

```
    plotMultipleSteeringAngles(p, 25, True, "Generated Track intensity multiplier 1", "20201207192948_nvidia2.h5"
, 'tcpflow log predicted')

def plot5():
    """
    Intensity multiplier 4, 20201207192948_nvidia2.h5
    """
    # intensity multiplier = 4
    p = []
    # No rain mult 1 (shaw we have a "no rain" for every multiplier ?)
    sa = GetSteeringFromtcpflow('../../trained_models/nvidia2/tcpflow/20201207192948_nvidia2_no_rain_tcpflow.lo
g')
    sarr = np.asarray(sa)
    pa = sarr[:, 0]
    p.append([pa, 'no rain'])
    # light mult 1
    sa = GetSteeringFromtcpflow('../../trained_models/nvidia2/tcpflow/20201207192948_nvidia2_light_rain_mult_4_
tcpflow.log')
    sarr = np.asarray(sa)
    pa = sarr[:, 0]
    p.append([pa, 'light rain'])
    # heavy mult 1
    sa = GetSteeringFromtcpflow('../../trained_models/nvidia2/tcpflow/20201207192948_nvidia2_heavy_10_mult_4_
tcpflow.log')
    sarr = np.asarray(sa)
    pa = sarr[:, 0]
    p.append([pa, 'heavy rain slant +-10'])
    # torrential mult 1
    sa = GetSteeringFromtcpflow('../../trained_models/nvidia2/tcpflow/20201207192948_nvidia2_torrential_20_mult
_4_tcpflow.log')
    sarr = np.asarray(sa)
    pa = sarr[:, 0]
    p.append([pa, 'torrential rain slant +-20'])

    plotMultipleSteeringAngles(p, 25, True, "Generated Track intensity multiplier 4", "20201207192948_nvidia2.h5"
,
                               'tcpflow log predicted')

def plot6():
    """
    Intensity multiplier 8, 20201207192948_nvidia2.h5
    """

    # intensity multiplier = 8
    p = []
    # No rain mult 1 (shaw we have a "no rain" for every multiplier ?)
    sa = GetSteeringFromtcpflow('../../trained_models/nvidia2/tcpflow/20201207192948_nvidia2_no_rain_tcpflow.lo
g')
    sarr = np.asarray(sa)
    pa = sarr[:, 0]
    p.append([pa, 'no rain'])
    # light mult 1
    sa = GetSteeringFromtcpflow(
        '../../trained_models/nvidia2/tcpflow/20201207192948_nvidia2_light_rain_mult_8_tcpflow.log')
```

```
    sarr = np.asarray(sa)
    pa = sarr[:, 0]
    p.append([pa, 'light rain'])
    # heavy mult 1
    sa = GetSteeringFromtcpflow(
        '../../trained_models/nvidia2/tcpflow/20201207192948_nvidia2_heavy_10_mult_8_tcpflow.log')
    sarr = np.asarray(sa)
    pa = sarr[:, 0]
    p.append([pa, 'heavy rain slant +-10'])
    # torrential mult 1
    sa = GetSteeringFromtcpflow(
        '../../trained_models/nvidia2/tcpflow/20201207192948_nvidia2_torrential_20_mult_8_tcpflow.log')
    sarr = np.asarray(sa)
    pa = sarr[:, 0]
    p.append([pa, 'torrential rain slant +-20'])

    plotMultipleSteeringAngles(p, 25, True, "Generated Track intensity multiplier 8", "20201207192948_nvidia2.h5"
,
                               'tcpflow log predicted')
if __name__ == "__main__":
    # 20201207091932_nvidia1.h5
    # mult_1
    # plot1()
    # mult_4
    # plot2()
    # mult_8
    # plot3()
    # 20201207192948_nvidia2.h5
    # mult_1
    # plot4()
    # mult_4
    # plot5()
    # mult_8
    # plot6()

#######################################################
# 17. SteeringAngleBins.ipynb.py
#######################################################


#!/usr/bin/env python
# coding: utf-8

# In[1]:


import fnmatch
import json
import seaborn as sns
import os
import numpy as np
import matplotlib.pyplot as plt
import statistics
```

```
def load_json(filepath):
    """
    Load a json file
    Inputs
        filepath: string, path to file
    Outputs:
        data: dictionary, json key, value pairs
    """
    with open(filepath, "rt") as fp:
        data = json.load(fp)
    return data

def GetJSONSteeringAngles(filemask):
    """
    Get steering angles stored as 'user/angle' attributes in .json files
    Inputs:
        filemask: string, path and mask
    Outputs
        svals: list, steering values
    """
    filemask = os.path.expanduser(filemask)
    path, mask = os.path.split(filemask)

    matches = []
    for root, dirnames, filenames in os.walk(path):
        for filename in fnmatch.filter(filenames, mask):
            matches.append(os.path.join(root, filename))

    # matches = sorted(matches, key=os.path.getmtime)

    # steering values
    svals = []
    for fullpath in matches:
            frame_number = os.path.basename(fullpath).split("_")[0]
            json_filename = os.path.join(os.path.dirname(fullpath), "record_" + frame_number + ".json")
            jobj = load_json(json_filename)
            svals.append(jobj['user/angle'])
    return svals


# In[5]:


#als = GetJSONSteeringAngles('~/git/msc-data/unity/genRoad/*.jpg')
#es = len(svals)
# Unity steering angle
#nityMaxSteering = 25#
#valscp = [element * UnityMaxSteering for element in svals]
# NB Plotted as normalized histogram
#ns.distplot(svalscp, bins=50, kde=False, norm_hist=True,
#           axlabel='Steering Angles (degrees) norm. hist. ' + str(values) + " data points \n \
#           mean = 0.00 std = 0.00")

# sns_plot.savefig("output.png")
```

```
def jsonSteeringBins(filemask, pname="output", save=True, nc=25):
    """
    Plot a steering values' histogram
    Inputs
        filemask: string, where to search for images, and corresponding .json files
        pname: string, output plot name
        save: boolean, save plot to disk
        nc: int, normalization constant, used in the simulator to put angles in range
        -1, 1. Default is 25.
    Outputs
        svals: list containing non-normalized steering angles
    """
    svals = GetJSONSteeringAngles(filemask)
    svalscp = [element * nc for element in svals]
    values = len(svalscp)
    mean = ("%.2f" % statistics.mean(svals))
    std = ("%.2f" % statistics.stdev(svals))
    plt.title=(pname)
    # NB Plotted as normalized histogram
    sns.distplot(svalscp, bins=nc*2, kde=False, norm_hist=True,
    axlabel= pname + ' steer. degs. norm. hist. ' + str(values) + ' values, mean = ' + mean + ' std = ' + std)
    #if(save):
    #    sns.save("output.png")
    plt.savefig(pname + '.png')

    # return for downstream processing if required
    return svals

def listSteeringBins(svals, pname="output", save=True, nc=25):
    """
    Plot a steering values' histogram
    Inputs
        svals: list, array of normalized steering values
        pname: string, output plot name
        save: boolean, save plot to disk
        nc: int, normalization constant, used in the simulator to put angles in range
        -1, 1. Default is 25.
    Outputs
        none
    """
    svalscp = [element * nc for element in svals]
    values = len(svalscp)
    mean = ("%.2f" % statistics.mean(svals))
    std = ("%.2f" % statistics.stdev(svals))
    plt.title=(pname)
    # NB Plotted as normalized histogram
    sns.distplot(svalscp, bins=nc*2, kde=False, norm_hist=True,
    axlabel= pname + ' steer. degs. norm. hist. ' + str(values) + ' values, mean = ' + mean + ' std = ' + std)
    #if(save):
    #    sns.save("output.png")
    plt.savefig(pname + '.png')

    # return for downstream processing if required
    return svals
```

```
svals = jsonSteeringBins('~/git/msc-data/unity/genRoad/*.jpg', 'genRoad')
```

In[ ]:

In[52]:

```
def printTimeMS(nf, fr):
    """
    Print time in minutes and seconds.
    Inputs
        nf: integer, number of frames
        fr: float, frame rate
    Outputs
        None
    """
    total_secs = nf / fr
    minutes = int(nf / fr / 60)
    seconds = round(nf / fr / 60 % minutes * 60)
    print("{}m{}s".format(minutes, seconds))

printTimeMS(45410, 24)
printTimeMS(280727, 24)
```

In[54]:

In[87]:

```
# simpler
import datetime
str(datetime.timedelta(seconds=280727/24))
```

In[4]:

```
svals = GetJSONSteeringAngles('~/git/msc-data/unity/genRoad/*.jpg')
values = len(svals)
# Unity steering angle
UnityMaxSteering = 25
svalscp = [element * UnityMaxSteering for element in svals]
```

```
# NB Plotted as normalized histogram
sns.distplot(svalscp, bins=50, kde=False, norm_hist=True,
             axlabel='Steering Angles (degrees) norm. hist. ' + str(values) + " data points")


# In[99]:


def printJSONSteeringAngles(filemask):
    """
    Get steering angles stored as 'user/angle' attributes in .json files
    Inputs:
        filemask: string, path and mask
    Outputs
        svals: list, steering values
    """
    filemask = os.path.expanduser(filemask)
    path, mask = os.path.split(filemask)

    matches = []
    for root, dirnames, filenames in os.walk(path):
        for filename in fnmatch.filter(filenames, mask):
            matches.append(os.path.join(root, filename))

    # steering values
    svals = []
    for fullpath in matches:
            frame_number = os.path.basename(fullpath).split("_")[0]
            json_filename = os.path.join(os.path.dirname(fullpath), "record_" + frame_number + ".json")
            jobj = load_json(json_filename)
            svals.append(jobj['user/angle'])
    return svals
    # return svals
svals = printJSONSteeringAngles('~/git/sdsandbox/sdsim/log/*.jpg')
cnt = 0
sec = 0
for vals in svals:
    cnt += 1
    if(cnt % 24 == 0):
        sec += 1
        print("Frame {} ========================".format(sec))
    print(vals)


# In[116]:


UnityMaxSteering = 25
svalscp = [element * UnityMaxSteering for element in svals]
my_iterator = filter(lambda svalscp: svalscp <= 20 and svalscp >= -20, svalscp)

svals2020 = list(my_iterator)

rm = len(svals) - len(svals2020)
```

```
pctrm =  rm * 100  / len(svals)
print("Removed entries outside -20 + 20 range: ", len(svals) - len(svals2020))
print("%.2f" % pctrm)


# In[121]:


# mean and std
import statistics
print("%.2f" % statistics.mean(svals2020))
print("%.2f" % statistics.stdev(svals2020))


# In[6]:


######################
# SkewClenaup.png
######################


# genRoad outlier cleanup
# NB Name is last one plotted
# plot everything
# used in SkewCleanup.png
import os
# plot all - nice effect but breaks kernel
# Now let's do one plot per folder to find out where these outliers are
path = '~/git/msc-data/unity/genRoad' # both work
path = '../../dataset/unity/genRoad/'
dirs = os.walk(path)
for mydir in dirs:
    if (len(mydir[1]) == 0):
        fn = mydir[0].split('/')
        path = mydir[0] + '/*.jpg'
        svals = jsonSteeringBins(path, fn[-1])


# deleted empties logs_Thu_Jul__9_16_12_28_2020
# both ~/git ... and ../.. work ok
# \../../dataset/unity/genRoad/logs_Thu_Jul__9_12_25_38_2020/*.jpg', 'test')


# In[20]:


import os
# plot all - nice effect but breaks kernel
# Now let's do one plot per folder to find out where these outliers are
path = '~/git/msc-data/unity/genRoad' # both work
path = '../../dataset/unity/genRoad/'
dirs = os.walk(path)
for mydir in dirs:
```

```
    if (len(mydir[1]) == 0):
        fn = mydir[0].split('/')
        path = mydir[0] + '/*.jpg'
        print("svals = jsonSteeringBins('{}', '{}')".format(path, fn[-1]))
        # svals = jsonSteeringBins(path, fn[-1])


# In[34]:


# find skewer
#svals = jsonSteeringBins('../../dataset/unity/genRoad/logs_Fri_Jul_10_09_29_13_2020/*.jpg', 'logs_Fri_Jul_10_09
_29_13_2020')
#svals = jsonSteeringBins('../../dataset/unity/genRoad/logs_Thu_Jul__9_15_47_22_2020/*.jpg', 'logs_Thu_Jul__9_1
5_47_22_2020')
#svals = jsonSteeringBins('../../dataset/unity/genRoad/logs_Thu_Jul__9_16_06_19_2020/*.jpg', 'logs_Thu_Jul__9_1
6_06_19_2020')
#svals = jsonSteeringBins('../../dataset/unity/genRoad/logs_Thu_Jul__9_15_53_22_2020/*.jpg', 'logs_Thu_Jul__9_1
5_53_22_2020')
#svals = jsonSteeringBins('../../dataset/unity/genRoad/logs_Fri_Jul_10_09_32_12_2020/*.jpg', 'logs_Fri_Jul_10_09
_32_12_2020')
#svals = jsonSteeringBins('../../dataset/unity/genRoad/logs_Thu_Jul__9_12_57_15_2020/*.jpg', 'logs_Thu_Jul__9_1
2_57_15_2020')
#svals = jsonSteeringBins('../../dataset/unity/genRoad/logs_Thu_Jul__9_13_03_17_2020/*.jpg', 'logs_Thu_Jul__9_1
3_03_17_2020')
# svals = jsonSteeringBins('../../dataset/unity/genRoad/logs_Thu_Jul__9_12_33_37_2020/*.jpg', 'logs_Thu_Jul__9_
12_33_37_2020')
#svals = jsonSteeringBins('../../dataset/unity/genRoad/logs_Thu_Jul__9_14_59_57_2020/*.jpg', 'logs_Thu_Jul__9_1
4_59_57_2020')
#svals = jsonSteeringBins('../../dataset/unity/genRoad/logs_Fri_Jul_10_09_22_57_2020/*.jpg', 'logs_Fri_Jul_10_09
_22_57_2020')
#svals = jsonSteeringBins('../../dataset/unity/genRoad/logs_Thu_Jul__9_16_08_45_2020/*.jpg', 'logs_Thu_Jul__9_1
6_08_45_2020')
svals = jsonSteeringBins('../../dataset/unity/genRoad/logs_Thu_Jul__9_16_00_15_2020/*.jpg', 'logs_Thu_Jul__9_16
_00_15_2020')
#svals = jsonSteeringBins('../../dataset/unity/genRoad/logs_Fri_Jul_10_09_16_18_2020/*.jpg', 'logs_Fri_Jul_10_09
_16_18_2020')
# svals = jsonSteeringBins('../../dataset/unity/genRoad/logs_Thu_Jul__9_12_25_38_2020/*.jpg', 'logs_Thu_Jul__9_
12_25_38_2020')


# In[35]:


# moved skewing logs
# mv genRoad/logs_Thu_Jul__9_16_00_15_2020 quarantine/
svals = jsonSteeringBins('../../dataset/unity/genRoad/*.jpg', 'logs_Fri_Jul_10_09_29_13_2020')


# In[ ]:
```

```
#####################################################
# 18. steerlib.ipynb.py
#####################################################


#!/usr/bin/env python
# coding: utf-8

# In[10]:


# steerlib - Helper library to create videos and plots

# modules
import fnmatch
import json
import seaborn as sns
import os
import numpy as np
import argparse
import matplotlib.pyplot as plt
import statistics
import seaborn as sns
import pickle


# In[15]:


def load_json(filepath):
    """
    Load a json file
    Inputs
        filepath: string, path to file
    Outputs
        data: dictionary, json key, value pairs
    Example
    path = "~/git/msc-data/unity/roboRacingLeague/log/logs_Sat_Nov_14_12_36_16_2020/record_11640.json"
    js = load_json(path)
    """
    with open(filepath, "rt") as fp:
        data = json.load(fp)
    return data


# In[16]:


def GetJSONSteeringAngles(filemask):
    """
    Get steering angles stored as 'user/angle' attributes in .json files
    Inputs:
```

```
    filemask: string, path and mask
  Outputs
    svals: list, steering values
  """
  filemask = os.path.expanduser(filemask)
  path, mask = os.path.split(filemask)

  matches = []
  for root, dirnames, filenames in os.walk(path):
    for filename in fnmatch.filter(filenames, mask):
      matches.append(os.path.join(root, filename))

  # steering values
  svals = []
  for fullpath in matches:
      frame_number = os.path.basename(fullpath).split("_")[0]
      json_filename = os.path.join(os.path.dirname(fullpath), "record_" + frame_number + ".json")
      jobj = load_json(json_filename)
      svals.append(jobj['user/angle'])
  return svals


# In[64]:


def jsonSteeringBins(filemask, pname="output", save=True, nc=25, rmout=0):
  """
  Plot a steering values' histogram
  Inputs
    filemask: string, where to search for images, and corresponding .json files
    pname: string, output plot name
    save: boolean, save plot to disk
    nc: int, normalization constant, used in the simulator to put angles in range
    -1, 1. Default is 25.
    rmout: integer, outlier range to remove
  Outputs
    svals: list containing non-normalized steering angles
  Example:
  # svals = jsonSteeringBins('~/git/msc-data/unity/genRoad/*.jpg', 'genRoad', save=True, nc=25, rmout=20)
  """
  svals = GetJSONSteeringAngles(filemask)
  values = len(svals)
  svalscp = [element * nc for element in svals]
  if(rmout>0):
    my_iterator = filter(lambda svalscp: svalscp <= rmout and svalscp >= (-1 * rmout), svalscp)
    svalsrmout = list(my_iterator)
    svalscp = svalsrmout
    values = len(svalsrmout)
    print("Removed {} records".format(len(svals) - len(svalsrmout)))
    svals = svalsrmout
  mean = ("%.2f" % statistics.mean(svalscp))
  std = ("%.2f" % statistics.stdev(svalscp))
  plt.title=(pname)
  # NB Plotted as normalized histogram
```

```
    sns.distplot(svalscp, bins=nc*2, kde=False, norm_hist=True,
    axlabel= pname + ' st. degs. norm. hist. ' + str(values) + ' values, mean = ' + mean + ' std = ' + std)
    #if(save):
    #    sns.save("output.png")
    if(save):
        plt.savefig(pname + '.png')
    plt.show()
    # return for downstream processing if required
    return svals


# In[63]:


def removeOutliers(svals, rmout, nc):
    """
    Remove outliers from a list
    Inputs
        svals: double, steering values
        rmout: integer, -+ range tp remove
        nc: steering normalization constant - same as used in simulator (max steering)
    Output
        svals: list, list with values excluded
    """
    svalscp = [element * nc for element in svals]
    my_iterator = filter(lambda svalscp: svalscp <= rmout and svalscp >= (-1 * rmout), svalscp)
    svalsrmout = list(my_iterator)
    svalscp = svalsrmout
    values = len(svalsrmout)
    print("Removed {} records".format(len(svalscp) - len(svalsrmout)))
    svals = svalsrmout
    return svals


# In[65]:


def listSteeringBins(svals, pname="output", save=True, nc=25, rmout=0):
    """
    Plot a steering values' histogram
    Inputs
        svals: list, array of normalized steering values
        pname: string, output plot name
        save: boolean, save plot to disk
        nc: int, normalization constant, used in the simulator to put angles in range
        -1, 1. Default is 25.
        rmout: integer, outlier range to remove
    Outputs
        none
    """
    svalscp = [element * nc for element in svals]
    values = len(svals)

    # remove outliers
```

```
    if(rmout>0):
        #my_iterator = filter(lambda svalscp: svalscp <= rmout and svalscp >= (-1 * rmout), svalscp)
        #svalsrmout = list(my_iterator)
        #svalscp = svalsrmout
        #values = len(svalsrmout)
        #print("Removed {} records".format(len(svals) - len(svalsrmout)))
        #svals = svalsrmout
        svals = removeOutliers(svalscp, rmout, nc)
        values = len(svals)
    mean = ("%.2f" % statistics.mean(svalscp))
    std = ("%.2f" % statistics.stdev(svalscp))
    plt.title=(pname)
    # NB Plotted as normalized histogram
    sns.distplot(svalscp, bins=nc*2, kde=False, norm_hist=True,
    axlabel= pname + ' steer. degs. norm. hist. ' + str(values) + ' values, mean = ' + mean + ' std = ' + std)
    #if(save):
    #    sns.save("output.png")
    if(save):
        plt.savefig(pname + '.png')
    plt.show()


# In[66]:


filemask = '~/git/msc-data/unity/genRoad/*.jpg'
svals = GetJSONSteeringAngles(filemask)
listSteeringBins(svals, pname="outputExOut20", save=True, nc=25, rmout=20)


# In[58]:


filemask = '~/git/msc-data/unity/genRoad/*.jpg'
svals = GetJSONSteeringAngles(filemask)
listSteeringBins(svals, pname="output", save=True, nc=25, rmout=0)


# In[68]:


# generate training dataset latex tables for report
for folder in ['genRoad','log_sample', 'roboRacingLeague','smallLoop', 'smallLoopingCourse','warehouse']:
    print(folder)

"""
SDSandbox   unity/smallLoopingCourse/log/* 34443 from small\_looping\_course
SDSandbox   unity/warehouse/*   41126 From Warehouse course
SDSandbox   unity/smallLoop/*   45422   From small\_looping\_course
SDSandbox   unity/roboRacingLeague/* 12778 From "Robot Racing League" course
SDSandbox   unity/log\_sample   25791   From small\_looping\_course
SDSandbox   unity/genRoad 280727 From "Generated Road" course
"""
```

In[ ]:

\begin{table}[]
\begin{center}
\begin{tabular}{|l|l|l|l|}
\hline
\multicolumn{4}{|c|}{Deliverables - Datasets} \\ \hline

get_ipython().run_line_magic('ID', 'Task Deliverable Description')
get_ipython().run_line_magic('1', 'Download D1 Udacity real world dataset')
get_ipython().run_line_magic('2', 'Generate   D2 Unity3D simulator data')
get_ipython().run_line_magic('3', 'Combine D3 Udacity real and simulator data')
get_ipython().run_line_magic('4', 'Mechanical Turk dry/rainy Ford dataset')

ID & Task & Deliverable & Description \\ \hline\hline
1 & Download & D1 & Udacity real world dataset \\ \hline
2 & Generate & D2 & Udacity simulator data \\ \hline
3 & Combine & D3 & Udacity real and simulator data \\ \hline
4 & Gather & D4 & Mechanical Turk dry/rainy Ford dataset \\ \hline

\end{tabular}
\end{center}
\caption{Datasets used to train models}
\label{Deliverables-Datasets}
\end{table}

###
19. steerlib.py
###

Helper library to create videos and plots

import fnmatch
import json
import os
import numpy as np
import matplotlib.pyplot as plt
import statistics
import seaborn as sns
import pickle
from PIL import Image
# prediction
import tensorflow as tf
from tensorflow.python import keras
from tensorflow.python.keras.models import load_model
# rain
from predict_client import add_rain
# Augmentation library
import Augmentation

```
def load_json(filepath):
    """
    Load a json file
    Inputs
        filepath: string, path to file
    Outputs
        data: dictionary, json key, value pairs
    Example
    path = "~/git/msc-data/unity/roboRacingLeague/log/logs_Sat_Nov_14_12_36_16_2020/record_11640.json"
    js = load_json(path)
    """
    with open(filepath, "rt") as fp:
        data = json.load(fp)
    return data

def GetSteeringFromtcpflow(filename):
    """
    Get a tcpflow log and extract steering values obtained from network communication between.
    Note, we only plot the predicted steering angle jsondict['steering']
    and the value of jsondict['steering_angle'] is ignored. Assumed to be the steering angle
    calculated by PID given the current course.
    sim and prediction engine (predict_client.py)
    Inputs
        filename: string, name of tcpflow log
    Returns
        sa: list of arrays, steering angle predicton and actual value tuple.
    Example


    """
    # open file
    sa = []
    # initialize prediction
    pred = ''
    f = open(filename, "r")
    file = f.read()
    try:
        # readline = f.read()
        lines = file.splitlines()
        for line in lines:
            # print(line)
            start = line.find('{')
            if (start == -1):
                continue
            jsonstr = line[start:]
            # print(jsonstr)
            jsondict = json.loads(jsonstr)
            if "steering" in jsondict:
                # predicted
                pred = jsondict['steering']
                # jsondict['steering_angle']
                # sa.append([float(pred), act])
                sa.append([float(pred), float(pred)])  # append twice to keep code from breaking
```

```
            # if "steering_angle" in jsondict:
            # actual
            #   act = jsondict['steering_angle']
            # save pair, only keep last pred in case two were send as it does happen i.e.:
            # 127.000.000.001.59460-127.000.000.001.09091: {"msg_type": "control", "steering": "-0.071960375", "thr
ottle": "0.08249988406896591", "brake": "0.0"}
            # 127.000.000.001.59460-127.000.000.001.09091: {"msg_type": "control", "steering": "-0.079734944", "thr
ottle": "0.08631626516580582", "brake": "0.0"}
            # 127.000.000.001.09091-127.000.000.001.59460: {"msg_type":"telemetry","steering_angle":-0.07196037,(.
..)
            #   if(len(pred) > 0):
            #      sa.append([float(pred), act])
            #      pred = '' # need to save this image
            # deal with image later, sort out plot first
            # imgString = jsondict["image"]
            # image = Image.open(BytesIO(base64.b64decode(imgString)))
            # img_arr = np.asarray(image, dtype=np.float32)
    except Exception as e:
        print("Exception raise: " + str(e))
    # file should be automatically closed but will close for good measure
    f.close()
    return sa

def GetJSONSteeringAngles(filemask):
    """
    Get steering angles stored as 'user/angle' attributes in .json files
    Inputs:
        filemask: string, path and mask
    Outputs
        svals: list, steering values
    """
    filemask = os.path.expanduser(filemask)
    path, mask = os.path.split(filemask)

    matches = []
    for root, dirnames, filenames in os.walk(path):
        for filename in fnmatch.filter(filenames, mask):
            matches.append(os.path.join(root, filename))
    # sort by create date
    matches = sorted(matches, key=os.path.getmtime)
    # steering values
    svals = []
    for fullpath in matches:
            frame_number = os.path.basename(fullpath).split("_")[0]
            json_filename = os.path.join(os.path.dirname(fullpath), "record_" + frame_number + ".json")
            jobj = load_json(json_filename)
            svals.append(jobj['user/angle'])
    return svals

def jsonSteeringBins(filemask, pname="output", save=True, nc=25):
    """
    Plot a steering values' histogram
    Inputs
        filemask: string, where to search for images, and corresponding .json files
```

```
        pname: string, output plot name
        save: boolean, save plot to disk
        nc: int, normalization constant, used in the simulator to put angles in range
        -1, 1. Default is 25.
    Outputs
        svals: list containing non-normalized steering angles
    Example:
    # svals = jsonSteeringBins('~/git/msc-data/unity/genRoad/*.jpg', 'genRoad')
    """
    svals = GetJSONSteeringAngles(filemask)
    values = len(svals)
    svalscp = [element * nc for element in svals]
    mean = ("%.2f" % statistics.mean(svals))
    std = ("%.2f" % statistics.stdev(svals))
    plt.title=(pname)
    # NB Plotted as normalized histogram
    sns.distplot(svalscp, bins=nc*2, kde=False, norm_hist=True,
    axlabel= pname + ' steer. degs. norm. hist. ' + str(values) + ' values, mean = ' + mean + ' std = ' + std)
    #if(save):
    #    sns.save("output.png")
    plt.savefig(pname + '.png')

    # return for downstream processing if required
    return svals

def plot_hist(path):
    """
    Create loss/accuracy plot for training and save to disk
    Inputs
        path: file to history pickle file
    Outputs
        none
    Example:
    path = "/home/simbox/git/sdsandbox/trained_models/nvidia1/20201107144927_nvidia1.history"
    plot_hist(path)
    or
    $ python steerlib.py
    $ python
    $ >>> import steerlib
    $ >>> path = "/home/simbox/git/sdsandbox/trained_models/nvidia1/20201107144927_nvidia1.history"
    $ >>> plot_hist(path)
    """
    history = pickle.load(open(path, "rb"))
    # type(history)
    # <class 'dict'>
    # history.keys()
    # dict_keys(['loss', 'acc', 'val_loss', 'val_acc'])
    do_plot = True
    try:
        if do_plot:
            fig = plt.figure()  # when loading dictionary keys, we omit .history (see train.py)
            plot_name = path.split('/')[-1]
            sp = plot_name \
                + ' - ' + '(l,vl,a,va)' + '{0:.3f}'.format(history['loss'][-1]) \
```

```
            + ',' + '{0:.3f}'.format(history['val_loss'][-1]) \
            + ',' + '{0:.3f}'.format(history['acc'][-1]) \
            + ',' + '{0:.3f}'.format(history['val_acc'][-1])

        fig.suptitle(sp, fontsize=9)
        ax = fig.add_subplot(111)
        ax.plot(history['loss'], 'r-', label='Training Loss', )
        ax.plot(history['val_loss'], 'm-', label='Validation Loss')
        ax2 = ax.twinx()
        ax2.plot(history['acc'], '-', label='Training Accuracy')
        ax2.plot(history['val_acc'], '-', label='Validation Accuracy')
        ax.legend(loc=2) # https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.legend.html
        ax.grid()
        ax.set_xlabel("Epoch")
        ax.set_ylabel(r"Loss")
        ax2.set_ylabel(r"Accurary")
        ax.set_ylim(0, 0.2)
        ax2.set_ylim(0.5, 1)
        ax2.legend(loc=1)
        aimg = plot_name.split('.')[0]
        aimg = str(aimg)  + '_accuracy.png'
        plt.savefig(aimg)
    except Exception as e:
        print("Failed to save accuracy/loss graph: " + str(e))

def gos(p, g, n):
    """
    Calculate the goodness-of-steer between a prediction and a ground truth array.
    Inputs
        p: array of floats, steering angle prediction
        g: array of floats, steering angle ground truth.
        n: float, normalization constant
    Output
        gos: float, average of absolute difference between ground truth and prediction arrays
    """
    # todo add type assertion
    assert len(p) == len(g), "Arrays must be of equal length"
    return sum(abs(p - g)) / len(p) * n
    # print("Goodness of steer: {:.2f}".format(steer))

def plotSteeringAngles(p, g=None, n=1, save=False, track= "Track Name", mname="model name", title='title'):
    """
    Plot predicted and (TODO) optionally ground truth steering angles
    Inputs
        p, g: prediction and ground truth float arrays
        n: float, steering normalization constant
        save: boolean, save plot flag
        track, mname, title: string, track (data), trained model and title strings for plot
    Outputs
        plt: pyplot, plot
    Example
    # set argument variables (see data_predict.py)
    plotSteeringAngles(p, g, nc, True, datapath[-2], modelpath[-1], 'Gs ' + gss)
    """
```

```
    plt.rcParams["figure.figsize"] = (18,3)

    plt.plot(p*n, label="predicted")
    try:
        if (g is not None):
            plt.plot(g*n, label="ground truth")
    except Exception as e:
        print("problems plotting: " + str(e))

    plt.ylabel('Steering angle')
    plt.xlabel('Frame number')
    # Set a title of the current axes.
    # plt.title('tcpflow log predicted steering angles: track ' + track + ' model ' + mname)
    plt.title(title + ' Steering angles: track ' + track + ', model ' + mname)
    # show a legend on the plot
    plt.legend()
    # Display a figure.
    # horizontal grid only
    plt.grid(axis='y')
    # set limit
    plt.xlim([-5,len(p)+5])
    plt.gca().invert_yaxis()
    # plt.show()
    if(save==True):
        plt.savefig('sa_' + track + '_' + mname + '.png')
    # if need be
    return plt

def plotMultipleSteeringAngles(p, n=25, save=False, track="Track Name", mname="model name", title='title', w=1
8, h=3):
    """
    Plot multiple predicted and (TODO) optionally ground truth steering angles
    Inputs
        p: list of tuples, prediction and labels
        n: float, steering normalization constant
        save: boolean, save plot flag
        track, mname, title: string, track (data), trained model and title strings for plot
        w: integer, plot width
        h: integer, plot height
    Outputs
        plt: pyplot, plot
    Example
    # get some steering angles
    sa = GetSteeringFromtcpflow('../trained_models/nvidia1/tcpflow/20201207091932_nvidia1_no_rain_tcpflow.log'
)
    sarr = np.asarray(sa)
    pa = sarr[:,0]
    p.append([pa, 'no rain'])
    plotSteeringAngles(p, g, nc, True, datapath[-2], modelpath[-1], 'Gs ' + gss)
    """
    import matplotlib.pyplot as plt # local copy
    plt.rcParams["figure.figsize"] = (w,h)
```

```
    for i in range (0, len(p)):
        plt.plot(p[i][0]*n, label=p[i][1])
    #try:
      #  if (g is not None):
     #       plt.plot(g*n, label="ground truth")
    #except Exception as e:
    #    print("problems plotting: " + str(e))

    plt.ylabel('Steering angle')
    plt.xlabel('Frame number')
    # Set a title of the current axes.
    # plt.title('tcpflow log predicted steering angles: track ' + track + ' model ' + mname)
    plt.title(title + ' Steering angles: track ' + track + ', model ' + mname)
    # show a legend on the plot
    plt.legend()
    # Display a figure.
    # horizontal grid only
    plt.grid(axis='y')
    # set limit
    plt.xlim([-5,len(p[0][0])+5])
    plt.gca().invert_yaxis()
    # plt.show()
    if(save==True):
        plt.savefig('sa_' + track + '_' + mname + '.png')
    # if need be
    return plt

def getSteeringFromtcpflow(filename):
    """
    Get a tcpflow log and extract steering values obtained from network communication between sim and predict_clie
nt.py.
    Note, we only plot the predicted steering angle jsondict['steering']
    and the value of jsondict['steering_angle'] is ignored. Assumed to be the steering angle
    calculated by PID given the current course.
    sim and prediction engine (predict_client.py)
    Inputs
        filename: string, name of tcpflow log
    Returns
        sa: list of arrays, steering angle predicton and actual value tuple.
    Example


    """
    # open file
    sa = []
    # initialize prediction
    pred = ''
    f = open(filename, "r")
    file = f.read()
    try:
        # readline = f.read()
        lines = file.splitlines()
        for line in lines:
            # print(line)
```

```
            start = line.find('{')
            if (start == -1):
                continue
            jsonstr = line[start:]
            # print(jsonstr)
            jsondict = json.loads(jsonstr)
            if "steering" in jsondict:
                # predicted
                pred = jsondict['steering']
                # jsondict['steering_angle']
                # sa.append([float(pred), act])
                sa.append([float(pred), float(pred)])  # append twice to keep code from breaking
            # if "steering_angle" in jsondict:
            # actual
            #   act = jsondict['steering_angle']
            # save pair, only keep last pred in case two were send as it does happen i.e.:
            # 127.000.000.001.59460-127.000.000.001.09091: {"msg_type": "control", "steering": "-0.071960375", "thr
ottle": "0.08249988406896591", "brake": "0.0"}
            # 127.000.000.001.59460-127.000.000.001.09091: {"msg_type": "control", "steering": "-0.079734944", "thr
ottle": "0.08631626516580582", "brake": "0.0"}
            # 127.000.000.001.09091-127.000.000.001.59460: {"msg_type":"telemetry","steering_angle":-0.07196037,(.
..)
            #   if(len(pred) > 0):
            #      sa.append([float(pred), act])
            #      pred = '' # need to save this image
            # deal with image later, sort out plot first
            # imgString = jsondict["image"]
            # image = Image.open(BytesIO(base64.b64decode(imgString)))
            # img_arr = np.asarray(image, dtype=np.float32)
    except Exception as e:
        print("Exception raise: " + str(e))
    # file should be automatically closed but will close for good measure
    f.close()
    return sa

"""

# get image

# get steering

# add / don't add rain

outputs = self.model.predict(image_array)

Append groundtruth, predicted to list.
"""
def PrintLatexRowModelGOS(filemask, modelpath, modelname, rt='', st=0):
    """
    Generate a "goodness of fit value" for a model on a given track
    Inputs
        filemask: string, path and mask
        modelpath: string, path to keras model
        modelname: string, canonical model name e.g. nvidia1, nvidia2, nvidia_baseline
```

```
    rt: string, rain type e.g. drizzle/light, heavy torrential
    st: integer, -+20 degree rain slant
Outputs
    svals: list, ground truth steering values and predictions
"""
# load augmentation library for correct model geometry
ag = Augmentation.Augmentation(modelname)

# load model
print("loading model", modelpath)
model = load_model(modelpath)

# In this mode, looks like we have to compile it
model.compile("sgd", "mse")
filemask = os.path.expanduser(filemask)
path, mask = os.path.split(filemask)

matches = []
for root, dirnames, filenames in os.walk(path):
    for filename in fnmatch.filter(filenames, mask):
        matches.append(os.path.join(root, filename))

# steering values
svals = []
for fullpath in matches:
    frame_number = os.path.basename(fullpath).split("_")[0]
    json_filename = os.path.join(os.path.dirname(fullpath), "record_" + frame_number + ".json")
    jobj = load_json(json_filename)
    # steering ground truth
    steer_gt = jobj['user/angle']
    # open the image
    img_arr = Image.open(fullpath)
    # Convert PIL Image to numpy array
    img_arr = np.array(img_arr, dtype=np.float32)
    # add rain if need be
    if rt != '':
        img_arr = add_rain(img_arr, rt, st)
    # apply same preprocessing
    # same preprocessing as for training
    img_arr = ag.preprocess(img_arr)

    # put in correct format
    img_arr = img_arr.reshape((1,) + img_arr.shape)
    # generate prediction
    outputs = model.predict(img_arr)
    # store predictions
    steer_pred = outputs[0][0]
    # store ground truth and prediction
    svals.append([steer_pred, steer_gt])
# get goodness of fit
sarr = np.asarray(svals)
p = sarr[:, 0]
g = sarr[:, 1]
nc = 25 # unity maximum steering angle / normalization constant - should hold in conf.py and managed with Aug
```

```
mentation
    mygos = gos(p, g, nc)
    # format to human readable/friendlier 2 decimal places
    gos_str = "{:.2f}".format(round(mygos, 2))
    # strip path from modelpath
    modelfile = modelpath.split('/')
    modelfile = modelfile[-1]
    # print latex formated data
    # header
    hd_str = 'Filename & Model & Rain Type & Slant & gos \\\\ \hline'
    # log file
    print('Log: ', path, '\\\\ \hline')
    print(hd_str)
    # results
    res_str = f'{modelfile} & {modelname} & {rt} & {st} & {gos_str} \\\\ \hline'
    print(res_str)

def GetPredictedSteeringAngles(filemask, model, modelname, rt='', st=0):
    """
    Generate a "goodness of fit value" for a model on a given track
    Inputs
        filemask: string, path and mask
        modelpath: string, path to keras model
        modelname: string, canonical model name e.g. nvidia1, nvidia2, nvidia_baseline
        rt: string, rain type e.g. drizzle/light, heavy torrential
        st: integer, -+20 degree rain slant
    Outputs
        svals: list, ground truth steering values and predictions
    """
    # load augmentation library for correct model geometry
    ag = Augmentation.Augmentation(modelname)

    # load model
    # print("loading model", modelpath)
    # assume model is loaded and compiled
    # model = load_model(modelpath)

    # In this mode, looks like we have to compile it
    # model.compile("sgd", "mse")
    filemask = os.path.expanduser(filemask)
    path, mask = os.path.split(filemask)

    matches = []
    for root, dirnames, filenames in os.walk(path):
        for filename in fnmatch.filter(filenames, mask):
            matches.append(os.path.join(root, filename))

    # sort by create date
    matches = sorted(matches, key=os.path.getmtime)
    # steering values
    svals = []
    for fullpath in matches:
        frame_number = os.path.basename(fullpath).split("_")[0]
        json_filename = os.path.join(os.path.dirname(fullpath), "record_" + frame_number + ".json")
```

```
        jobj = load_json(json_filename)
        # steering ground truth
        steer_gt = jobj['user/angle']
        # open the image
        img_arr = Image.open(fullpath)
        # Convert PIL Image to numpy array
        img_arr = np.array(img_arr, dtype=np.float32)
        # add rain if need be
        if rt != '':
            img_arr = add_rain(img_arr, rt, st)
        # apply same preprocessing
        # same preprocessing as for training
        img_arr = ag.preprocess(img_arr)

        # put in correct format
        img_arr = img_arr.reshape((1,) + img_arr.shape)
        # generate prediction
        outputs = model.predict(img_arr)
        # store predictions
        steer_pred = outputs[0][0]
        # store ground truth and prediction
        svals.append([steer_pred, steer_gt])
    return svals

def printGOSRows():
    """
    Print GOS rows for results report.
    We are comparing mainly the two best models for nvidia1 and nvidia2, would have been nice to also test
    the driveable nvidia_baseline, and sanity models
    """
    # models
    # nvidia2 - ../../trained_models/nvidia2/20201207192948_nvidia2.h5
    # nvidia1 - ../../trained_models/nvidia1/20201207091932_nvidia1.h5
    # 20201120124421\_nvidia\_baseline.h5
    # log logs_Wed_Nov_25_23_39_22_2020
    ################################################
    # 1. nvidia2 Generated track
    ################################################
    # log = '../../dataset/unity/genTrack/genTrackOneLap/logs_Wed_Nov_25_23_39_22_2020/*.jpg'
    #modelpath = '../../trained_models/nvidia2/20201207192948_nvidia2.h5'
    #modelname = 'nvidia2'
    #rt = 'torrential'
    #st = 20
    #PrintLatexRowModelGOS(log, modelpath, modelname, rt, st)
    # Log:  ../../dataset/unity/genTrack/genTrackOneLap/logs_Wed_Nov_25_23_39_22_2020 \\ \hline
    # Filename & Model & Rain Type & Slant & gos \\ \hline
    # 20201207192948_nvidia2.h5 & nvidia2 &  & 0 & 1.68 \\ \hline
    # Log:  ../../dataset/unity/genTrack/genTrackOneLap/logs_Wed_Nov_25_23_39_22_2020 \\ \hline
    # Filename & Model & Rain Type & Slant & gos \\ \hline
    # 20201207192948_nvidia2.h5 & nvidia2 & light & 0 & 2.12 \\ \hline
    # Log:  ../../dataset/unity/genTrack/genTrackOneLap/logs_Wed_Nov_25_23_39_22_2020 \\ \hline
    # Filename & Model & Rain Type & Slant & gos \\ \hline
    # 20201207192948_nvidia2.h5 & nvidia2 & heavy & 10 & 2.17 \\ \hline
    # Log:  ../../dataset/unity/genTrack/genTrackOneLap/logs_Wed_Nov_25_23_39_22_2020 \\ \hline
```

```
# Filename & Model & Rain Type & Slant & gos \\ \hline
# 20201207192948_nvidia2.h5 & nvidia2 & torrential & 20 & 2.30 \\ \hline
#########################################################
# 2. nvidia1 Generated track
#########################################################
# modelpath = '../../trained_models/nvidia1/20201207091932_nvidia1.h5'
#modelname = 'nvidia1'
#rt = 'torrential'
#st = 20
#PrintLatexRowModelGOS(log, modelpath, modelname, rt, st)
# Log:  ../../dataset/unity/genTrack/genTrackOneLap/logs_Wed_Nov_25_23_39_22_2020 \\ \hline
# Filename & Model & Rain Type & Slant & gos \\ \hline
# 20201207091932_nvidia1.h5 & nvidia1 &  & 0 & 1.82 \\ \hline
# Log:  ../../dataset/unity/genTrack/genTrackOneLap/logs_Wed_Nov_25_23_39_22_2020 \\ \hline
# Filename & Model & Rain Type & Slant & gos \\ \hline
# 20201207091932_nvidia1.h5 & nvidia1 & light & 0 & 2.11 \\ \hline
# Log:  ../../dataset/unity/genTrack/genTrackOneLap/logs_Wed_Nov_25_23_39_22_2020 \\ \hline
# Filename & Model & Rain Type & Slant & gos \\ \hline
# 20201207091932_nvidia1.h5 & nvidia1 & heavy & 10 & 2.13 \\ \hline
# Log:  ../../dataset/unity/genTrack/genTrackOneLap/logs_Wed_Nov_25_23_39_22_2020 \\ \hline
# Filename & Model & Rain Type & Slant & gos \\ \hline
# 20201207091932_nvidia1.h5 & nvidia1 & torrential & 20 & 2.28 \\ \hline
#########################################################
# 3. nvidia_baseline Generated track
#########################################################
#modelpath = '../../trained_models/nvidia_baseline/20201207201157_nvidia_baseline.h5'
#modelname = 'nvidia2_baseline'
#rt = 'torrential'
#st = 20
#PrintLatexRowModelGOS(log, modelpath, modelname, rt, st)
# Log:  ../../dataset/unity/genTrack/genTrackOneLap/logs_Wed_Nov_25_23_39_22_2020 \\ \hline
# Filename & Model & Rain Type & Slant & gos \\ \hline
# 20201207201157_nvidia_baseline.h5 & nvidia2_baseline &  & 0 & 2.32 \\ \hline
# Log:  ../../dataset/unity/genTrack/genTrackOneLap/logs_Wed_Nov_25_23_39_22_2020 \\ \hline
# Filename & Model & Rain Type & Slant & gos \\ \hline
# 20201207201157_nvidia_baseline.h5 & nvidia2_baseline & light & 0 & 3.12 \\ \hline
# Log:  ../../dataset/unity/genTrack/genTrackOneLap/logs_Wed_Nov_25_23_39_22_2020 \\ \hline
# Filename & Model & Rain Type & Slant & gos \\ \hline
# 20201207201157_nvidia_baseline.h5 & nvidia2_baseline & heavy & 10 & 3.17 \\ \hline
# Log:  ../../dataset/unity/genTrack/genTrackOneLap/logs_Wed_Nov_25_23_39_22_2020 \\ \hline
# Filename & Model & Rain Type & Slant & gos \\ \hline
# 20201207201157_nvidia_baseline.h5 & nvidia2_baseline & torrential & 20 & 3.39 \\ \hline
#########################################################
# 4. sanity Generated track 20201120171015\_ sanity.h5
#########################################################
#modelpath = '../../trained_models/sanity/20201120171015_sanity.h5'
#modelname = 'nvidia1'
#rt = 'torrential'
#st = 20
#PrintLatexRowModelGOS(log, modelpath, modelname, rt, st)
# Log:  ../../dataset/unity/genTrack/genTrackOneLap/logs_Wed_Nov_25_23_39_22_2020 \\ \hline
# Filename & Model & Rain Type & Slant & gos \\ \hline
# 20201120171015_sanity.h5 & nvidia1 &  & 0 & 5.03 \\ \hline
# Log:  ../../dataset/unity/genTrack/genTrackOneLap/logs_Wed_Nov_25_23_39_22_2020 \\ \hline
```

```
    # Filename & Model & Rain Type & Slant & gos \\ \hline
    # 20201120171015_sanity.h5 & nvidia1 & light & 0 & 3.11 \\ \hline
    # Log:  ../../dataset/unity/genTrack/genTrackOneLap/logs_Wed_Nov_25_23_39_22_2020 \\ \hline
    # Filename & Model & Rain Type & Slant & gos \\ \hline
    # 20201120171015_sanity.h5 & nvidia1 & heavy & 10 & 3.07 \\ \hline
    # Log:  ../../dataset/unity/genTrack/genTrackOneLap/logs_Wed_Nov_25_23_39_22_2020 \\ \hline
    # Filename & Model & Rain Type & Slant & gos \\ \hline
    # 20201120171015_sanity.h5 & nvidia1 & torrential & 20 & 3.00 \\ \hline

    ################################################
    # 5. nvidia2 Generated Road
    ################################################
    log = '../../dataset/unity/genRoad/logs_Fri_Jul_10_09_16_18_2020/*.jpg'
    #modelpath = '../../trained_models/nvidia2/20201207192948_nvidia2.h5'
    #modelname = 'nvidia2'
    #rt = 'torrential'
    #st = 20
    #PrintLatexRowModelGOS(log, modelpath, modelname, rt, st)
    # Log:  ../../dataset/unity/genRoad/logs_Fri_Jul_10_09_16_18_2020 \\ \hline
    # Filename & Model & Rain Type & Slant & gos \\ \hline
    # 20201207192948_nvidia2.h5 & nvidia2 &  & 0 & 2.99 \\ \hline # drove 16 minutes on this road https://youtu.be
/z9nILq9dQfI
    # Log:  ../../dataset/unity/genRoad/logs_Fri_Jul_10_09_16_18_2020 \\ \hline
    # Filename & Model & Rain Type & Slant & gos \\ \hline
    # 20201207192948_nvidia2.h5 & nvidia2 & light & 0 & 3.20 \\ \hline
    # Log:  ../../dataset/unity/genRoad/logs_Fri_Jul_10_09_16_18_2020 \\ \hline
    # Filename & Model & Rain Type & Slant & gos \\ \hline
    # 20201207192948_nvidia2.h5 & nvidia2 & heavy & 10 & 3.22 \\ \hline
    # Log:  ../../dataset/unity/genRoad/logs_Fri_Jul_10_09_16_18_2020 \\ \hline
    # Filename & Model & Rain Type & Slant & gos \\ \hline
    # 20201207192948_nvidia2.h5 & nvidia2 & torrential & 20 & 3.27 \\ \hline
    ################################################
    # 6. nvidia1 Generated Road
    ################################################
    #modelpath = '../../trained_models/nvidia1/20201207091932_nvidia1.h5'
    #modelname = 'nvidia1'
    # rt = 'torrential'
    # st = 20
    # PrintLatexRowModelGOS(log, modelpath, modelname, rt, st)
    # Log:  ../../dataset/unity/genRoad/logs_Fri_Jul_10_09_16_18_2020 \\ \hline
    # Filename & Model & Rain Type & Slant & gos \\ \hline
    # 20201207091932_nvidia1.h5 & nvidia1 &  & 0 & 3.87 \\ \hline
    # Log:  ../../dataset/unity/genRoad/logs_Fri_Jul_10_09_16_18_2020 \\ \hline
    # Filename & Model & Rain Type & Slant & gos \\ \hline
    # 20201207091932_nvidia1.h5 & nvidia1 & light & 0 & 3.75 \\ \hline
    # Log:  ../../dataset/unity/genRoad/logs_Fri_Jul_10_09_16_18_2020 \\ \hline
    # Filename & Model & Rain Type & Slant & gos \\ \hline
    # 20201207091932_nvidia1.h5 & nvidia1 & heavy & 10 & 3.70 \\ \hline
    # Log:  ../../dataset/unity/genRoad/logs_Fri_Jul_10_09_16_18_2020 \\ \hline
    # Filename & Model & Rain Type & Slant & gos \\ \hline
    # 20201207091932_nvidia1.h5 & nvidia1 & torrential & 20 & 3.57 \\ \hline

    ################################################
    # 7. nvidia_baseline Generated Road
```

```
    ########################################################
    #modelpath = '../../trained_models/nvidia_baseline/20201207201157_nvidia_baseline.h5'
    #modelname = 'nvidia2_baseline'
    #rt = 'torrential'
    #st = 20
    #PrintLatexRowModelGOS(log, modelpath, modelname, rt, st)
    # Log:  ../../dataset/unity/genRoad/logs_Fri_Jul_10_09_16_18_2020 \\ \hline
    # Filename & Model & Rain Type & Slant & gos \\ \hline
    # 20201207201157_nvidia_baseline.h5 & nvidia2_baseline &  & 0 & 5.51 \\ \hline
    # Log:  ../../dataset/unity/genRoad/logs_Fri_Jul_10_09_16_18_2020 \\ \hline
    # Filename & Model & Rain Type & Slant & gos \\ \hline
    # 20201207201157_nvidia_baseline.h5 & nvidia2_baseline & light & 0 & 4.97 \\ \hline
    # Log:  ../../dataset/unity/genRoad/logs_Fri_Jul_10_09_16_18_2020 \\ \hline
    # Filename & Model & Rain Type & Slant & gos \\ \hline
    # 20201207201157_nvidia_baseline.h5 & nvidia2_baseline & heavy & 10 & 4.98 \\ \hline
    # Log:  ../../dataset/unity/genRoad/logs_Fri_Jul_10_09_16_18_2020 \\ \hline
    # Filename & Model & Rain Type & Slant & gos \\ \hline
    # 20201207201157_nvidia_baseline.h5 & nvidia2_baseline & torrential & 20 & 5.05 \\ \hline
    ########################################################
    # 8. sanity Generated track 20201120171015\_ sanity.h5
    ########################################################
    modelpath = '../../trained_models/sanity/20201120171015_sanity.h5'
    modelname = 'nvidia1'
    rt = 'torrential'
    st = 20
    PrintLatexRowModelGOS(log, modelpath, modelname, rt, st)
    # Log:  ../../dataset/unity/genRoad/logs_Fri_Jul_10_09_16_18_2020 \\ \hline
    # Filename & Model & Rain Type & Slant & gos \\ \hline
    # 20201120171015_sanity.h5 & nvidia1 &  & 0 & 3.85 \\ \hline
    # Log:  ../../dataset/unity/genRoad/logs_Fri_Jul_10_09_16_18_2020 \\ \hline
    # Filename & Model & Rain Type & Slant & gos \\ \hline
    # 20201120171015_sanity.h5 & nvidia1 & light & 0 & 3.06 \\ \hline
    # Log:  ../../dataset/unity/genRoad/logs_Fri_Jul_10_09_16_18_2020 \\ \hline
    # Filename & Model & Rain Type & Slant & gos \\ \hline
    # 20201120171015_sanity.h5 & nvidia1 & heavy & 10 & 3.05 \\ \hline
    # Log:  ../../dataset/unity/genRoad/logs_Fri_Jul_10_09_16_18_2020 \\ \hline
    # Filename & Model & Rain Type & Slant & gos \\ \hline
    # 20201120171015_sanity.h5 & nvidia1 & torrential & 20 & 3.02 \\ \hline

def printMultiPlots(model1_nvidia2, model2_nvidia1, model3_nvidia_baseline, model4_sanity):
    """
    Print multiple plots to reference first 16 results in Goodness of steer tables
    Inputs
     model1_nvidia2, model2_nvidia1, model3_nvidia_baseline, model4_sanity: the required keras models
    """
    # init plot list
    plot_list = []
    # define log
    log = '../../dataset/unity/genTrack/genTrackOneLap/logs_Wed_Nov_25_23_39_22_2020/*.jpg'
    # Get ground truth values
    gt = GetJSONSteeringAngles(log)
    gt = np.asarray(gt)
    # get predictions
    ############################################
```

```
    # 1. nvidia2 Generated track
    ###############################################
    # 1.1 dry
    """
    print('Predicting nvidia2 dry...')
    sa = GetPredictedSteeringAngles(log, model1_nvidia2, 'nvidia2', rt='', st=0)
    sarr = np.asarray(sa)
    p = sarr[:,0]
    g = sarr[:, 1]
    plot_list.append([g, 'ground truth'])
    plot_list.append([p, 'prediction - no rain'])
    # 1.2 light rain
    print('Predicting nvidia2 light rain...')
    sa = GetPredictedSteeringAngles(log, model1_nvidia2, 'nvidia2', rt='light', st=0)
    sarr = np.asarray(sa);
    p = sarr[:,0]
    plot_list.append([p, 'prediction - light rain'])
    # 1.3 heavy rain, slant = +-10
    print('Predicting nvidia2 heavy rain...')
    sa = GetPredictedSteeringAngles(log, model1_nvidia2, 'nvidia2', rt='heavy', st=10)
    p = sarr[:,0]
    plot_list.append([p, 'prediction - heavy rain +-10'])
    # 1.4 torrential rain, slant = +-20
    print('Predicting nvidia2 torrential rain...')
    sa = GetPredictedSteeringAngles(log, model1_nvidia2, 'nvidia2', rt='torrential', st=20)
    sarr = np.asarray(sa);
    p = sarr[:,0]
    plot_list.append([p, 'prediction - torrential rain +-20'])
    print('Plotting...')
    plotMultipleSteeringAngles(plot_list, 25, True, "Generated_Track", "20201207192948_nvidia2.h5", title='SDSan
dbox log genTrackOneLap/logs_Wed_Nov_25_23_39_22_2020', w=18, h=4)
    """
    ###############################################
    # 2. nvidia1 Generated track
    ###############################################
    # 2.1 dry
    print('Predicting nvidia1 dry...')
    sa = GetPredictedSteeringAngles(log, model2_nvidia1, 'nvidia1', rt='', st=0)
    sarr = np.asarray(sa);
    p = sarr[:,0]
    g = sarr[:, 1]
    plot_list.append([g, 'ground truth'])
    plot_list.append([p, 'prediction - no rain'])
    # 2.2 light rain
    print('Predicting nvidia1 light rain...')
    sa = GetPredictedSteeringAngles(log, model2_nvidia1, 'nvidia1', rt='light', st=0)
    sarr = np.asarray(sa);
    p = sarr[:,0]
    plot_list.append([p, 'prediction - light rain'])
    # 2.3 heavy rain, slant = +-10
    print('Predicting nvidia1 heavy rain...')
    sa = GetPredictedSteeringAngles(log, model2_nvidia1, 'nvidia1', rt='heavy', st=10)
    sarr = np.asarray(sa);
    p = sarr[:,0]
```

```
    plot_list.append([p, 'prediction - heavy rain +-10'])
    # 2.4 torrential rain, slant = +-20
    print('Predicting nvidia1 torrential rain...')
    sa = GetPredictedSteeringAngles(log, model2_nvidia1, 'nvidia1', rt='torrential', st=20)
    sarr = np.asarray(sa);
    p = sarr[:,0]
    plot_list.append([p, 'prediction - torrential rain +-20'])
    print('Plotting...')
    plotMultipleSteeringAngles(plot_list, 25, True, "Generated_Track", "20201207091932_nvidia1.h5", title='SDSan
dbox log genTrackOneLap/logs_Wed_Nov_25_23_39_22_2020', w=18, h=4)

    ##############################################
    # 3. nvidia_baseline Generated track
    ##############################################
    # 3.1 dry
    """
    print('Predicting model3_nvidia_baseline dry...')
    sa = GetPredictedSteeringAngles(log, model3_nvidia_baseline, 'nvidia1', rt='', st=0)
    sarr = np.asarray(sa);
    p = sarr[:,0]
    g = sarr[:, 1]
    plot_list.append([g, 'ground truth'])
    plot_list.append([p, 'prediction - no rain'])
    # 3.2 light rain
    print('Predicting model3_nvidia_baseline light rain...')
    sa = GetPredictedSteeringAngles(log, model3_nvidia_baseline, 'nvidia1', rt='light', st=0)
    sarr = np.asarray(sa);
    p = sarr[:,0]
    plot_list.append([p, 'prediction - light rain'])
    # 3.3 heavy rain, slant = +-10
    print('Predicting model3_nvidia_baseline heavy rain...')
    sa = GetPredictedSteeringAngles(log, model3_nvidia_baseline, 'nvidia1', rt='heavy', st=10)
    sarr = np.asarray(sa);
    p = sarr[:,0]
    plot_list.append([p, 'prediction - heavy rain +-10'])
    # 3.4 torrential rain, slant = +-20
    print('Predicting model3_nvidia_baseline torrential rain...')
    sa = GetPredictedSteeringAngles(log, model3_nvidia_baseline, 'nvidia1', rt='torrential', st=20)
    sarr = np.asarray(sa);
    p = sarr[:,0]
    plot_list.append([p, 'prediction - torrential rain +-20'])
    print('Plotting...')
    plotMultipleSteeringAngles(plot_list, 25, True, "Generated_Track", "20201207201157_nvidia_baseline.h5", title
='SDSandbox log genTrackOneLap/logs_Wed_Nov_25_23_39_22_2020', w=18, h=3)
    """

    """
    ##############################################
    # 4. sanity Generated track
    ##############################################
    # 4.1 dry
    print('Predicting model4_sanity dry...')
    sa = GetPredictedSteeringAngles(log, model4_sanity, 'nvidia1', rt='', st=0)
    sarr = np.asarray(sa);
```

```
    p = sarr[:,0]
    g = sarr[:, 1]
    plot_list.append([g, 'ground truth'])
    plot_list.append([p, 'prediction - no rain'])
    # 4.2 light rain
    print('Predicting model4_sanity light rain...')
    sa = GetPredictedSteeringAngles(log, model4_sanity, 'nvidia1', rt='light', st=0)
    sarr = np.asarray(sa);
    p = sarr[:,0]
    plot_list.append([p, 'prediction - light rain'])
    # 4.3 heavy rain, slant = +-10
    print('Predicting model4_sanity heavy rain...')
    sa = GetPredictedSteeringAngles(log, model4_sanity, 'nvidia1', rt='heavy', st=10)
    sarr = np.asarray(sa);
    p = sarr[:,0]
    plot_list.append([p, 'prediction - heavy rain +-10'])
    # 4.4 torrential rain, slant = +-20
    print('Predicting model4_sanity torrential rain...')
    sa = GetPredictedSteeringAngles(log, model4_sanity, 'nvidia1', rt='torrential', st=20)
    sarr = np.asarray(sa);
    p = sarr[:,0]
    plot_list.append([p, 'prediction - torrential rain +-20'])
    print('Plotting...')
    plotMultipleSteeringAngles(plot_list, 25, True, "Generated_Track", "20201120171015_sanity.h5", title='SDSand
box log genTrackOneLap/logs_Wed_Nov_25_23_39_22_2020', w=18, h=3)
    """
if __name__ == "__main__":
    # plot_hist("/home/simbox/git/sdsandbox/trained_models/nvidia1/20201107144927_nvidia1.history")
#if __name__ == "__main__":
#    parser = argparse.ArgumentParser(description='Plot Steering Utils')
#    parser.add_argument('--inputs', type=str, help='file path')

    # args = parser.parse_args()
    #svals = jsonSteeringBins('~/git/msc-data/unity/genRoad/*.jpg', 'genRoad')

    # PrintLatexRowModelGOS('../../dataset/unity/genTrack/genTrackOneLap/logs_Wed_Nov_25_23_39_22_2020/
*.jpg', '../../trained_models/nvidia2/20201207192948_nvidia2.h5', 'nvidia2')
    # printGOSRows()
    # load models
    model1_nvidia2 = ''
    model2_nvidia1 = ''
    model3_nvidia_baseline = ''
    model4_sanity = ''
    # modelpath = '../../trained_models/sanity/20201120171015_sanity.h5'
    # modelpath = '../../trained_models/nvidia_baseline/20201207201157_nvidia_baseline.h5'
    #modelpath = '../../trained_models/nvidia1/20201207091932_nvidia1.h5'
    # modelpath = '../../trained_models/nvidia2/20201207192948_nvidia2.h5'
    # assume model is loaded and compiled
    # nvidia2
    modelpath = '../../trained_models/nvidia2/20201207192948_nvidia2.h5'
    model1_nvidia2 = load_model(modelpath)
    model1_nvidia2.compile("sgd", "mse")
    # nvidia1
    modelpath = '../../trained_models/nvidia1/20201207091932_nvidia1.h5'
```

```
    model2_nvidia1 = load_model(modelpath)
    model2_nvidia1.compile("sgd", "mse")
    # nvidia_baseline
    modelpath = '../../trained_models/nvidia_baseline/20201207201157_nvidia_baseline.h5'
    model3_nvidia_baseline = load_model(modelpath)
    model3_nvidia_baseline.compile("sgd", "mse")
    # sanity
    modelpath = '../../trained_models/sanity/20201120171015_sanity.h5'
    model4_sanity = load_model(modelpath)
    model4_sanity.compile("sgd", "mse")

    printMultiPlots(model1_nvidia2, model2_nvidia1, model3_nvidia_baseline, model4_sanity)

    #path = 'record_11640.json'
    #js = load_json(path)
    #print(js)
    # plotSteeringAngles(p, None, 25, True, "Generated Track", "20201120171015_sanity.h5", 'tcpflow log predicted'
)
    # plots1()


#######################################################
# 20. train.py
# Note: code based on
# https://github.com/tawnkramer/sdsandbox/blob/master/src/train.py
# Available for audit in audit_files/tawn from sharepoint link
#######################################################


'''
Train
Train your nerual network
Author: Tawn Kramer
To fix missing .jpeg files, see utils/
'''
from __future__ import print_function

import argparse
import fnmatch
import json
import os
import pickle
import random
from datetime import datetime
from time import strftime
import numpy as np
from PIL import Image
from tensorflow import keras
# import tensorflow as tf
import conf
import models
from helper_functions import hf_mkdir, parse_bool
from augmentation import augment, preprocess
# import cv2
```

```
import Augmentation

'''
matplotlib can be a pain to setup. So handle the case where it is absent. When present,
use it to generate a plot of training results.
'''
try:
    import matplotlib
    # Force matplotlib to not use any Xwindows backend.
    matplotlib.use('Agg')

    import matplotlib.pyplot as plt
    do_plot = True
except:
    do_plot = False


def shuffle(samples):
    '''
    randomly mix a list and return a new list
    '''
    ret_arr = []
    len_samples = len(samples)
    while len_samples > 0:
        iSample = random.randrange(0, len_samples)
        ret_arr.append(samples[iSample])
        del samples[iSample]
        len_samples -= 1
    return ret_arr

def load_json(filename):
    with open(filename, "rt") as fp:
        data = json.load(fp)
    return data

def generator(samples, is_training, batch_size=64):
    '''
    Rather than keep all data in memory, we will make a function that keeps
    it's state and returns just the latest batch required via the yield command.

    As we load images, we can optionally augment them in some manner that doesn't
    change their underlying meaning or features. This is a combination of
    brightness, contrast, sharpness, and color PIL image filters applied with random
    settings. Optionally a shadow image may be overlayed with some random rotation and
    opacity.
    We flip each image horizontally and supply it as a another sample with the steering
    negated.
    '''
    num_samples = len(samples)

    while 1: # Loop forever so the generator never terminates
        samples = shuffle(samples)
        #divide batch_size in half, because we double each output by flipping image.
        for offset in range(0, num_samples, batch_size):
```

```
batch_samples = samples[offset:offset+batch_size]

images = []
controls = []
for fullpath in batch_samples: # not sure this is doing anything, as images are not being flipped
    try:
        frame_number = os.path.basename(fullpath).split("_")[0]
        json_filename = os.path.join(os.path.dirname(fullpath), "record_" + frame_number + ".json")
        data = load_json(json_filename)
        steering = float(data["user/angle"])
        throttle = float(data["user/throttle"])

        try:
            image = Image.open(fullpath)
        except:
            print('failed to open', fullpath)
            continue

        #PIL Image as a numpy array
        image = np.array(image, dtype=np.float32)
        # image_cp = image
        # resize for nvidia
        # nvidia 2
        # image = cv2.resize(image, (200, 66), cv2.INTER_AREA)
        # augmentation only for training
        if(conf.aug):
            if is_training and np.random.rand() < 0.6:
                image, steering = ag.augment(image, steering)

        # This provides this actual size network is expecting, so must run
        if (conf.preproc):
            image = ag.preprocess(image) # preprocess(image)
        # assert (preprocess(image)==ag.preprocess(image))

        # for nvidia2 model
        # 224 224 Alexnet
        # image = cv2.resize(image, (224, 224), cv2.INTER_AREA)
        # for NVIDIA should be 200x66
        images.append(image)

        if conf.num_outputs == 2:
            controls.append([steering, throttle])
        elif conf.num_outputs == 1:
            controls.append([steering])
        else:
            print("expected 1 or 2 outputs")

    except Exception as e:
        print(e)
        print("we threw an exception on:", fullpath)
        yield [], []


# final np array to submit to training
```

```
            X_train = np.array(images)
            y_train = np.array(controls)
            yield X_train, y_train


def get_files(filemask, s=False):
    '''
    Use a filemask and search a path recursively for matches
    Inputs
        filemask: string passed as command line option, must not be enclosed in quotes
        s: boolean, sort by create date flag
    '''

    filemask = os.path.expanduser(filemask)
    path, mask = os.path.split(filemask)

    matches = []
    for root, dirnames, filenames in os.walk(path):
        for filename in fnmatch.filter(filenames, mask):
            matches.append(os.path.join(root, filename))
    if(s == True):
        matches = sorted(matches, key=os.path.getmtime)
    return matches


def train_test_split(lines, test_perc):
    '''
    split a list into two parts, percentage of test used to seperate
    '''
    train = []
    test = []

    for line in lines:
        if random.uniform(0.0, 1.0) < test_perc:
            test.append(line)
        else:
            train.append(line)

    return train, test

def make_generators(inputs, limit=None, batch_size=conf.batch_size):
    '''
    load the job spec from the csv and create some generator for training
    '''

    #get the image/steering pairs from the csv files
    lines = get_files(inputs)
    print("found %d files" % len(lines))

    if limit is not None:
        lines = lines[:limit]
        print("limiting to %d files" % len(lines))

    train_samples, validation_samples = train_test_split(lines, test_perc=0.2)
```

```
    print("num train/val", len(train_samples), len(validation_samples))

    # compile and train the model using the generator function
    train_generator = generator(train_samples, True, batch_size=batch_size)
    validation_generator = generator(validation_samples, False, batch_size=batch_size)

    n_train = len(train_samples)
    n_val = len(validation_samples)

    return train_generator, validation_generator, n_train, n_val


def go(model_name, outdir, epochs=50, inputs='./log/*.jpg', limit=None):

    print('working on model', model_name)

    hf_mkdir(outdir)
    outdir += '/' + model_name
    hf_mkdir(outdir)

    # https://docs.python.org/3/library/datetime.html#strftime-strptime-behavior
    dt = strftime("%Y%m%d%H%M%S")
    fp = outdir + '/' + dt + '_' + model_name;
    model_name = fp + '.h5'

    '''
    modify config.json to select the model to train.
    '''
    # model = models.get_nvidia_model_naoki(conf.num_outputs)
    # interpreter seems to be playing up, dummy assignment to appease
    if(conf.model_name=='nvidia1'):
        model = models.nvidia_model1(conf.num_outputs)
    elif(conf.model_name=='nvidia2'):
        model = models.nvidia_model2(conf.num_outputs)
    elif(conf.model_name == 'nvidia_baseline'):
        model = models.nvidia_baseline(conf.num_outputs)
    elif(conf.model_name == 'alexnet'):
        try:
            model = models.get_alexnet(conf.num_outputs)
        except Exception as e:
            print("Failed to save accuracy/loss graph: " + str(e))
        # adjust image size
        # conf.row = conf.image_width_net = conf.image_width_alexnet
        # conf.col = conf.image_height_net = conf.image_height_alexnet

    else:
        try:
            raise ValueError
        except ValueError:
            print('No valid model name given. Please check command line arguments and model.py')

    callbacks = [
        # running with naoki's model
```

```
        keras.callbacks.EarlyStopping(monitor='val_loss', patience=conf.training_patience, verbose=0),
        keras.callbacks.ModelCheckpoint(model_name, monitor='val_loss', save_best_only=True, verbose=0),
        # keras.callbacks.ModelCheckpoint(('model-{epoch:03d}' +'_' + model_name), monitor='val_loss', save_best_o
nly=True, verbose=0),
    ]

    batch_size = conf.batch_size


    #Train on session images
    train_generator, validation_generator, n_train, n_val = make_generators(inputs, limit=limit, batch_size=batch_siz
e)

    if n_train == 0:
        print('no training data found')
        return

    steps_per_epoch = n_train // batch_size
    validation_steps = n_val // batch_size

    print("steps_per_epoch", steps_per_epoch, "validation_steps", validation_steps)
    s1 = strftime("%Y%m%d%H%M%S")

    #history = model.fit_generator(train_generator,
    #    steps_per_epoch = steps_per_epoch,
    #    validation_data = validation_generator,
    #    validation_steps = validation_steps,
    #    epochs=epochs,
    #    verbose=1,
    #    callbacks=callbacks)
    history = []
    try:
        history = model.fit(train_generator,
            steps_per_epoch = steps_per_epoch,
            validation_data = validation_generator,
            validation_steps = validation_steps,
            epochs=epochs,
            verbose=1,
            callbacks=callbacks)
    except Exception as e:
        print("Failed to run model: " + str(e))

    # e =  "Input to reshape is a tensor with 147456 values, but the requested shape requires a multiple of 27456". errp
r rao with jungle1 dataset
    #   [[node model/flattened/Reshape (defined at /git/sdsandbox/src/train.py:250) ]] [Op:__inference_train_function
_2398]
    #
    # Function call stack:
    # train_function
    s2 = strftime("%Y%m%d%H%M%S")
    FMT = "%Y%m%d%H%M%S"
    tdelta = datetime.strptime(s2, FMT) - datetime.strptime(s1, FMT)
    tdelta = "Total training time: " + str(tdelta)
```

```
    # save info
    log = fp + '.log'
    logfile = open(log, 'w')
    logfile.write("Model name: " + model_name + "\r\n")
    logfile.write(tdelta)
    logfile.write('\r\n');
    logfile.write("Training loss: " + '{0:.3f}'.format(history.history['loss'][-1]) + "\r\n")
    logfile.write("Validation loss: " + '{0:.3f}'.format(history.history['val_loss'][-1]) + "\r\n")
    logfile.write("Training accuracy: " + '{0:.3f}'.format(history.history['acc'][-1]) + "\r\n")
    logfile.write("Validation accuracy: " + '{0:.3f}'.format(history.history['val_acc'][-1]) + "\r\n")
    logfile.close()


    # save history
    histfile = fp + '.history'
    with open(histfile, 'wb') as file_pi:
        pickle.dump(history.history, file_pi)
    try:
        if do_plot:
            fig = plt.figure()
            sp = '(l,vl,a,va)' + '{0:.3f}'.format(history.history['loss'][-1]) \
                + ',' + '{0:.3f}'.format(history.history['val_loss'][-1]) \
                + ',' + '{0:.3f}'.format(history.history['acc'][-1]) \
                + ',' + '{0:.3f}'.format(history.history['val_acc'][-1]) \
                + ' - ' + model_name.split('/')[-1]
            fig.suptitle(sp, fontsize=9)

            ax = fig.add_subplot(111)
            #ax.plot(time, Swdown, '-', label='Swdown')
            ax.plot(history.history['loss'], 'r-', label='Training Loss', )
            ax.plot(history.history['val_loss'], 'c-', label='Validation Loss')
            ax2 = ax.twinx()
            ax2.plot(history.history['acc'], 'm-', label='Training Accuracy')
            ax2.plot(history.history['val_acc'], 'y-', label='Validation Accuracy')
            ax.legend(loc=2) # https://matplotlib.org/3.1.1/api/_as_gen/matplotlib.pyplot.legend.html
            ax.grid()
            ax.set_xlabel("Epoch")
            ax.set_ylabel(r"Loss")
            ax2.set_ylabel(r"Accurary")
            ax.set_ylim(0, 0.2)
            ax2.set_ylim(0.5, 1)
            ax2.legend(loc=1)
            aimg = fp + '_accuracy.png'
            plt.savefig(aimg)
    except Exception as e:
        print("Failed to save accuracy/loss graph: " + str(e))

# moved to helper functions
#def parse_bool(b):
#    return b == "True"

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description='train script')
    parser.add_argument('--model', type=str, help='model name')
```

```
    parser.add_argument('--outdir', type=str, help='output directory')
    parser.add_argument('--epochs', type=int, default=conf.training_default_epochs, help='number of epochs')
    parser.add_argument('--inputs', default='../dataset/unity/genRoad/*.jpg', help='input mask to gather images')
    parser.add_argument('--limit', type=int, default=None, help='max number of images to train with')
    parser.add_argument('--aug', type=parse_bool, default=False, help='image augmentation flag')
    parser.add_argument('--preproc', type=parse_bool, default=True, help='image preprocessing flag')

    args = parser.parse_args()

    conf.aug = args.aug
    conf.preproc = args.preproc
    conf.model_name = args.model
    #print(tf.__version__) 2.2.0
    ag = Augmentation.Augmentation(args.model)
    go(args.model, args.outdir, epochs=args.epochs, limit=args.limit, inputs=args.inputs)

#python train.py ..\outputs\mymodel_aug_90_x4_e200 --epochs=200
```