Predict Steering Angles in Self-Driving Cars

Neha Yadav
University of Massachusetts Amherst
Amherst MA
nyadav@cs.umass.edu

Rishi Mody
University of Massachusetts Amherst
Amherst MA
rmody@cs.umass.edu

Abstract

The advance in technology over recent years has led major automobile manufactures to invest in making their cars technologically more furnished. A major push has been in the area of making cars autonomous and self-driving. Autonomous cars would not only make driving simpler but also safer.

Last year, Udacity released a dataset which among other data, contains a set of images annotated with the steering angle applied by a human during driving. The Udacity challenge [1] is aimed at predicting steering angle based on only the provided images of the road captured using cameras placed behind the car's windshield.

We design and explore multiple models to perform high quality prediction of steering angles based on these images using different deep learning techniques including deep Convolutional Neural Networks and hybrid models built on top of pre-trained networks like VGG16 on imagenet via Transfer Learning [11].

1. Introduction

The push for self-driving vehicles has increased dramatically over the last few years after the success of the DARPA Grand Challenge [2]. An autonomous vehicle works seamlessly when multiple components come together and work in synergy. The most important parts are the various sensors and the AI software powering the vehicle. One of the most critical functions of the AI is to predict the steering angle of the vehicle for the stretch of road lying immediately in front of it and accordingly even steer the car. Increase in computational capabilities over the years allows us to train deep neural networks to become the "brain" of these cars which then understands the surroundings of the car and makes navigation decisions in real time.

[1] https://medium.com/udacity/challenge-2-using-deep-learning-to-predict-steering-angles-f42004a36ff3

[2] https://en.wikipedia.org/wiki/DARPA$_{G}rand_{C}hallenge$

Udacity held a series of challenges last year to create an open source self-driving car. They released a data set of images taken while a car was being manually driven, annotated with the corresponding steering angle applied by the human driver. The goal of one of the challenges was to find a model that, given an image taken while driving, will minimize the RMSE (root mean square error) between what the model predicts and the actual steering angle produced by a human driver.

In this project, we explore a variety of methods including deep convolutional neural networks, models based on pre-trained networks like VGG16, etc. to predict steering angle values. Predicting the steering angle is one of the most important parts of creating a self-driving car. This task would allow us to explore the full power of neural networks, using only steering angle as the training signal, deep neural networks can automatically extract features from the images to help understand the position of the car with respect to the road to make the prediction.

2. Problem Statement

A self-driving car is loaded with cameras which capture images of the road in front of it. Using these images in real-time the model should be able to analyze the direction of the road and predict how much the steering wheel must be turned to follow it. For this project, we have a data set of pre-captured images for which we will predict the angle.

3. Related Work

Research on autonomous vehicle navigation was pioneered by Pomerleau(1989) [6] when he built the Autonomous Land Vehicle in a Neural Network. The model comprised of a fully-connected network which would be considered a very basic version if compared to the large models in use today.Though the network could be applied to only a few simple scenarios with minimal obstacles this paper laid the foundation of end-to-end autonomous navigation.

Recently, an NVIDIA team consisting of Bojarski, Del

Testa et al [5]. carried out a study to design an end-to-end learning of self-driving cars. They trained a convolutional neural network(CNN) to map raw pixels from a single front facing camera directly to steering commands. They found that this method was surprisingly accurate, without much training from humans the model managed to learn to drive in traffic on local roads some of which had lane markings while some did not as well as on highways. Overall this framework was successful in relatively simple real-world scenarios, such as highway lane-following and driving in flat, obstacle-free courses.

The use of complicated neural network structures has increased in the process of classification of videos and object detection. These advancements are getting translated and transferred to challenges of autonomous driving.

Comma.ai [8] has proposed to learn a driving simulator that uses Generative Adversarial Networks (GANs) [3] and a Variational Auto-encoder (VAE) [2]. Their approach is able to keep predicting realistic looking video for several frames based on previous frames despite the transition model being optimized without a cost function in the pixel space. Moreover, deep reinforcement learning (RL) has also been applied to autonomous driving [7], [9]. RL has not been successful for automotive applications until some recent work, which shows the deep learning algorithm's ability to learn good representations of the environment. Inspired by the success of deep reinforcement learning in learning of games, [7] has proposed a framework for autonomous driving using deep RL it is an end-to-end Deep Reinforcement learning pipeline for autonomous driving which integrates RNNs to account for POMDP scenarios. The framework was tested for lane keep assist algorithm.

4. Dataset

The dataset has been provided by Udacity as a part of its Self-Driving Car challenge. They have generated images using NVIDIA's DAVE-2 System which uses 3 cameras placed behind the windshield of the car. A time stamped video is captured along with the steering angle applied by the human driver. The video is captured in varying conditions of light and traffic.

The data is given in a ROSbag format for which we used a reader developed by rwightman [12]. The reader loads the ROS bag, extract images as well as angles from the ROS bag. CH2_002 bag are used for training our model and CH_001 for evaluating its performance. Training data set contains 101397 frames and corresponding labels including steering angle, torque and speed.And there is also a test set which contains 5615 frames from the center camera. The original resolution of the image is 640x480.

Training images come from 5 different driving videos:

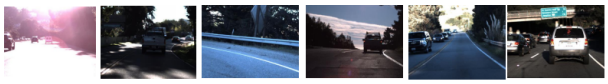


Fig. 1. Training data set. Typical images for different light, traffic and driving conditions. (a) Direct sunlight, (b) shadow, (c) sharp left turn, (d) uphill, (e) straight, (f) heavy traffic

Figure 1: Sample images from the data set

- 221 seconds, direct sunlight, many lighting changes. Good turns in beginning, discontinuous shoulder lines, ends in lane merge, divided highway
- discontinuous shoulder lines, ends in lane merge, divided highway 791 seconds, two lane road, shadows are prevalent, traffic signal (green), very tight turns where center camera cant see much of the road, direct sunlight, fast elevation changes leading to steep gains/losses over summit. Turns into divided highway around 350s, quickly returns to 2 lanes.
- 99 seconds, divided highway segment of return trip over the summit
- 212 seconds, guardrail and two lane road, shadows in beginning may make training difficult, mostly normalizes towards the end
- 371 seconds, divided multi lane highway with a fair amount of traffic

4.1. Data Augmentation Methods

A typical convolutional neural network can have up to a million parameters, and tuning these parameters requires millions of training instances of uncorrelated data, which may not always be possible and in some cases cost prohibitive. Due to limited data, deep neural networks have tendency to over fit the data. To avoid over-fitting, augmentation is one of the technique that is used widely. For example, for a more generalized dataset we will require the car to be driven under different weather, lighting, traffic and road conditions etc.

As we would be generating thousands of new instances of the image in real time, it is not practical to store these images on to the disk. Therefore we are utilizing keras generators to read data from the file, augment on the fly and use it to train the model. We would be generating augmented images using the following data augmentation techniques.

4.1.1 Horizontal and Vertical Shifts

In order to simulate the effect of car being at different positions on the road, we shifted the camera images horizontally and added an offset corresponding to the shift to the steering angle.

Shifting the image up/down should cause the model to believe it is on the upward/downward slope. Therefore,

camera images are shifted vertically by a random number to simulate the effect of driving up or down the slope.

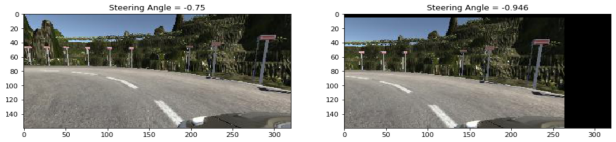


Figure 2: Original and horizontal shifted images

4.1.2 Brightness Augmentation

The motive behind brightness augmentation is to simulate the effect of different light conditions mainly day and night. Camera images with different brightness are generated by first converting images to HSV, scaling up or down the V channel and converting back to the RGB channel.

(a) Original Image (b) Brightened Image

Figure 3: Original and brightened images

4.1.3 Horizontally Flip images

As we expect that car to be able to steer itself regardless of its position on he road, we performed horizontal flip on images with the corresponding inversion of steering angle. This neutralized the steering bias in the training data as a majority of the images were of the car turning to the right.

(a) Steering Angle: -0.1256 (b) Steering Angle: 0.1256

Figure 4: Original and horizontally flipped images

4.1.4 Random Darken

As some parts of track could be much darker compared to other parts due to shadow or some other reason, we darkened a proportion of our images by multiplying all RGB color channels by a scalar randomly picked from a range. Figure 5.

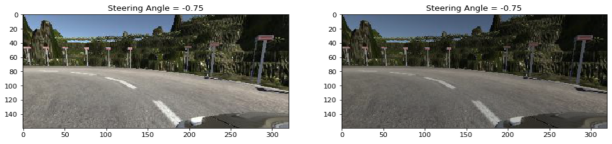


Figure 5: Original and darkened images

4.1.5 Shadow augmentation

Since our car should be able to recognize patches of track that are covered by a shadow, we added shadow augmentation where random shadows are cast across the image. To achieve this task, for a given camera image we chose random points and shaded all points on one side (chosen randomly) of the image.

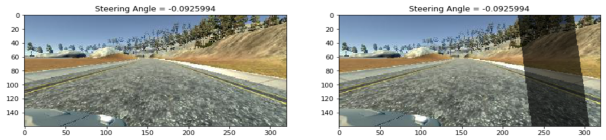


Figure 6: Original and shadowed images

4.2. Prepossessing

Our input image is shaped 160x320x3. Using lambda layer in keras it is cropped vertically to 88x320x3. Further the intensities in image are normalized between -.5 and .5 utilizing the lambda layer functionality in keras. Then images is then re-sized to 66x200 using keras image.resize function.

5. Methods

We developed two types of model structures. One is inspired from the architecture used in NVIDIA's end-to-end learning model [5] for self-driving cars while the other is a hybrid model whose bottom part is based on pre-trained network VGG16 trained on imagenet and stack of fully connected layers on top.

5.1. Deep ConvNet Model

This model of ours is inspired from NVIDIA's [5] CNN architecture from End to End learning from self driving cars.

Our input image as 160x320x3. Using keras lambda layer it is cropped vertically to 80x320 and then re-sized to 66x200.

The model consists of 3 5x5 convolutional layers with stride of 2x2 and network depth of 24,36 and 48 respectively. Then it is followed by 2 3x3 convolutional layers with a 1x1 stride and depth of 64. Each convolution layer

is followed by Batch Normalization. Output of the last convolutional layer is flattened before entering fully connected phase and series of fully connected layers are used.Each stack of fully connected layers consists of 1152, 200, 50 and 10, 1 respectively. Last layer predicts the steering angle. The activation function used across all layers, bar the last one, is ReLU. Evaluation metrics for this model is RMSE with adam as optimizer for the loss function. Overall parameters for this model are 1,229,193.

Batch Normalization [4] after each convolutional layer allowed us to use much higher learning rates and helps make our model more independent from the initialization parameters. It also acted as a regularizer, in some cases eliminating the need for Dropout [10].

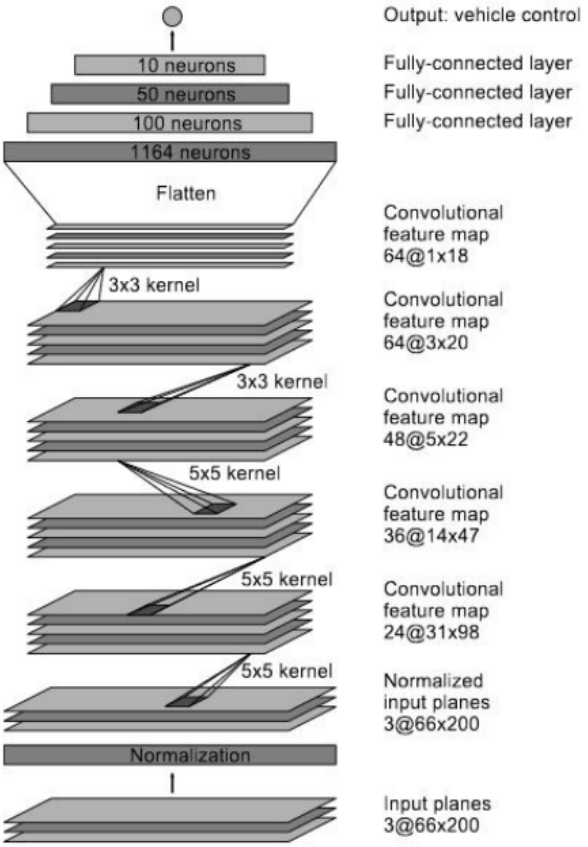


Figure 7: NVIDIA architecture

5.2. Hybrid Model

One common barrier for using deep learning to solve problems is the amount of data needed to train a model. The requirement of large data arises because of the large number of parameters in the model that machines have to learn. However models trained on one task capture relations in the data type and can easily be reused for different problems in the same domain. This technique is referred to as Transfer Learning. With transfer learning, we can take a pre-trained model, which was trained on a large readily available data set (trained for a completely different task, with the same input but different output). Then try to find layers which output reusable features. We use the output of that layer as input features to train a much smaller network that requires a smaller number of parameters. This smaller network only needs to learn the relations for your specific problem having already learned about patterns in the data from the pre-trained model.

Using transfer learning approach, we have built a **Hybrid model**. The bottom part of the model is based on VGG16. We have used Keras implementation of VGG16 model and it was trained on ImageNet. The output of the VGG16 is connected to stack of fully connected layers namely 512,256,64,10 and 1 different units respectively. The output is a single node with tanh activation function.

The architecture of this model can be seen in Figure 8 with overall parameters being 15,129,149. A block consists of dense layer,batch normalization,dropout. Dropout layers after batch normalization should help model prevent overfitting.

Each dense layer has relu as activation function except the last layer where we have used tanh. The idead begind using tanh is that it fits well with the angle distribution as well as generate values that can be positive or negative. The model output is a list of angles between -1 and 1.

Figure 8: Hybrid Model architecture (based on transfer learning technique)

We performed mutiple experiments to choose the number of stacks of fully connected layers and their dimensions for hybrid model. Combinations other than shown in fig 8 either were too slow to train or yielded poor results. Another major challenge to train this network was to fit memory in data. As we were also generating the augmented images on fly it was nearly impractical to load all images in

memory at once. Therefore, we used keras generator functionality where we generated 64 images per batch and total 20480 images per epoc.

5.3. Loss Function and Optimization

Because of the nature of the problem, our loss is computed as root mean squared error(RMSE). RMSE is calculated as the square root of the mean of the squared differences between actual outcomes and predictions. Since the errors are squared before they are averaged, the RMSE gives a relatively high weight to large errors. This means the RMSE is more useful when large errors are particularly undesirable. The benefit of using using RMSE over MSE is that it penalizes larger errors with worse scores.

$$MSE = 1/n \sum (y_i - \hat{y_i})^2 \tag{1}$$

$$RMSE = \sqrt{1/n \sum (y_i - \hat{y_i})^2} \tag{2}$$

To optimize the loss, for our **Deep Convnet Model** we have used 'adam' as optimizer with learning rate of 0.001. Adam combines the best properties of the AdaGrad and RMSProp algorithms to provide an optimization algorithm that can handle sparse gradients on noisy problems.Empirical results demonstrate that Adam works well in practice and compares favorably to other adaptive learning-method algorithms. The other default values of the Keras Adam optimizer showed good results during training ($learning\_rate = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 1e-10$, and decay=learning rate/batch size).

While for our **Hybrid Model**, we needed an optimizer with faster convergence rate. To achieve this we performed multiple experiments using different optimizers and found that 'nadam' works best for our model. Nadam incorporates Nesterov Momentum with adam as we needed an optimizer with faster convergence rate. Nesterov Momentum would speed up the convergence results. We have used keras implementation of Nadam with $learningrate = 1e-6$, $\beta_1 = 0.9$, $\beta_2 = 0.999$,$\epsilon = 1e-10$ and $schedule\_decay = 0.004$

6. Experiments

We processed the .csv file that contains the image file names and their corresponding steering angle and then randomized the data and split into 80/20 ratio. 20% of the data is used for validation purpose. All our models are trained and evaluated in GPU.

In **first phase** of our experiments all our models were trained on training set without using any data augmentation. Though, we performed image pre-processing like cropping the top part of image to remove sky from camera image. This was done considering the fact that sky does not help much in our task.

Further to save time and fine tune number of epochs we used early stopping technique. **Early stopping** is a type of regularization to curb over-fitting of the training data and requires that you monitor the performance of the model on training and a held validation data sets, each epoch. It simply stops training once the validation loss hasn't decreased for a pre decided number of epochs (the parameter is called 'patience'). Keras supports early stopping through an EarlyStopping callback class. We used (monitor='val_loss', min_delta=0.0001,patience=10). 'min_delta' is a threshold to whether quantify a loss at some epoch as improvement or not. 'patience' argument represents the number of epochs before stopping once your loss starts to increase (stops improving).

As a fault tolerance mechanism for long running process like training a deep neural network with millions of parameters, we used application check-pointing. It is an approach where a snapshot of the state of the system is taken in case of system failure. If there is a problem, not all is lost. The checkpoint only includes the model weights. It assumes you know the network structure which could be serialized to file in JSON or YAML format. These weights were used to make predictions as is, or used as the basis for ongoing training. Keras support checkpoint capability by The **ModelCheckpoint** callback. This ensured that our best model is saved for the run during training.

Now since memory is an limited resource and it is not practical to load all these images at once in memory. So we utilized Keras generator to sample images such that all angles have the similar probability no matter how they are represented in the data set. Keras generator is set up such that in the initial phases of learning, the model drops data with lower steering angles with higher probability. This removes any potential for bias towards driving at zero angle.

For optimizing the loss for our model, we used adam as optimizer for Deep Conv Net model, while nadam was the optimizer for Hybrid model. We began with using the default parameters in Keras for these optimizer, after measuring the loss after the first epoch we decreased the learning rate by a factor of 10 and measure the loss again. Once we had a low initial loss, we began training our model. The final parameters used in our models are mentioned in section 5.3 under loss and optimization.

Since our training data is limited and training deep neural network to learn and generalize better in real world scenario, huge amount of data is required. Our car should be able to drive in different conditions. Hence we switched our approach and decided to use data augmentation techniques. Using these data augmentation methods an infinite number of new images can be generated. We performed different augmentation on the training images as described in section 4.

In **second phase** of our experiments, we used data aug-

mentation heavily (examples in section 4.1). Various experiments were performed to optimize the number of images to be generated by generator in one batch and how many images to be generated in total for one epoch. We tried using batch size of 256,512, 128, 64, 32 and found 64 to be most efficient. Total 20480 images were used and we used 50 epochs with early stopping mechanism. After each epoch our models were evaluated against the validation data that we had split earlier.

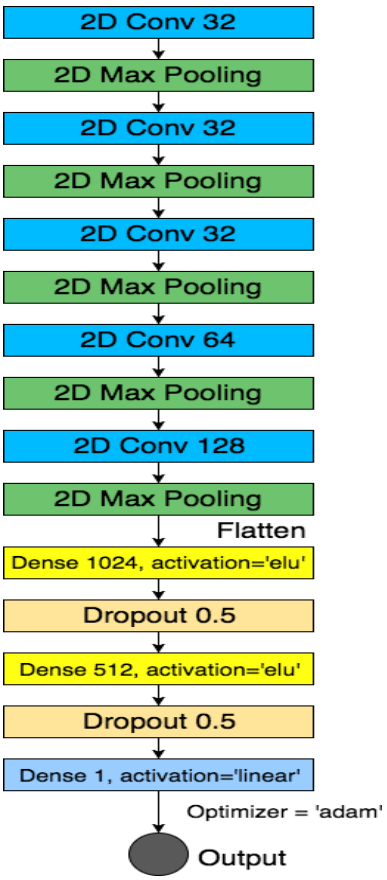


Figure 9: Experimental model architecture

6.1. Experimental Model

In both our models we have used batch normalization layer after convolutional layers. While in our hybrid model,we have used Dropout of 0.5 after batch normalization. Since we were interested to know effect of max pooling layers for spatial reduction (and memory friendliness). We tried another experimental model whose architecture could be seen in Fig 9. This model consists of 5 convolutional layers with Max pooling between them followed by a couple of dense layers and a linear activation to output continuous steering angles. We use the exponential linear unit (ELU) [1] activation throughout with adam as an optimizer. ELU acts like a ReLU unit if x is positive, but for negative values it is a function bounded by a fixed value -1, for =1. This behavior helps to push the mean activation of neurons closer to zero which is beneficial for learning and it helps to learn representations that are more robust to noise. Further ELU units actually seem to learn faster than other units and they are able to learn models which are at least as good as ReLU-like networks. After we realized augmentation was helpful, we trained our experimental model only on augmented images. This model did not perform well compared to our two main models. The RMSE for this model was 0.2031 on test set.

7. Results

NVIDIA, after conducting its experiments had set a benchmark of 0.20 RMSE for their architecture. Benchmark on this data set is set at 0.2067 which can be obtained if all the predicted angles are set to zero. All our models ran on the same data sets (training, validation, and test). The results for each model are listed in table 1 and 2.

Table 1: RMSE for models on Udacity data set

	Training Set	**Validation Set**	**Test Set**
Hybrid	0.0778	0.0736	**0.0708**
Deep Convnet	0.0945	0.1070	0.1043
NVIDIA	0.0750	0.0995	0.0986
Predict 0	0.2716	0.2130	0.2076

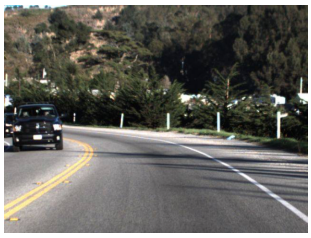
Table 2: RMSE for models on Udacity data set(no augmentation)

	Training Set	**Validation Set**	**Test Set**
Hybrid	0.1138	0.0997	0.0951
Deep Convnet	0.2072	0.1988	0.1907

It is evident from table 1 and 2, that image augmentation significantly improved the performance of our model and helped them generalize better. Fig 9 shows sample images along with true and predicted steering angle.

Results from our hybrid model using pre-trained VGG16 and stack of fully connected layers has RMSE 0.0708 for the test set. RMSE for our deep convnet model inspired from NVIDIA architecture was 0.1043. Retrospectively, had we participated in the Udacity challenge with this model, we would have been placed amongst the top 4 teams.

(a) Predicted: 0.00143, True: 0.00174



(b) Predicted: 0.00757, True: 0.00776

Figure 10: Comparison between true and predicted steering angle

8. Conclusion and Future Work

Predicting the steering angle for self driving cars is a very interesting problem. One of the major challenge to train a deep neural network to perform this task is the amount of training data. Our ultimate aim is that our car should be able to drive under different weather, lighting, traffic and road conditions. Apart from designing an efficient network, the model requires huge amount of data and hours of training. Based on limited number resources and training data our model did fairly well. Among all models hybrid model gave the best results. Image augmentation and generating images on the fly helped significantly by preventing model to over fit and helping the model generalize better.

We did not consider speed, torque and throttle for this task. Incorporating these metrics can help model improve performance and their values can also be predicted moving the model closer to a fully functional self-driving car model.One can also experiment with deep reinforcement learning to solve this problem. Because of the nature of the input, which is a series on consequence images about the same environment evolving by time. Using recurrent neural network or RNN with memory unit can be useful. Generate adversarial models, GANs, could be used to augment data set by generating different conditions on track while driving. For example GAN could be used to generate images on track during adverse weather conditions. To simulate driving in real world with minimal error, there is substantial research that still needs to be done on the subject before models like these can be deployed widely to transport the public.

References

[1] D.-A. Clevert, T. Unterthiner, and S. Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). https://arxiv.org/abs/1511.07289.

[2] C. Doersch. Tutorial on variational autoencoders. https://arxiv.org/abs/1606.05908.

[3] I. J. Goodfellow, J. Pouget-Abadie, et al. Generative adversarial nets. https://arxiv.org/abs/1406.2661v1.

[4] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. https://arxiv.org/abs/1502.03167.

[5] NVIDIA. End to end learning for self-driving cars. https://arxiv.org/abs/1604.07316.

[6] D. A. Pomerleau. Alvinn, an autonomous land vehicle in a neural network. http://repository.cmu.edu/cgi/viewcontent.cgi?article=2874&context=compsci.

[7] A. E. Sallab, M. Abdou, et al. Deep reinforcement learning framework for autonomous driving. https://arxiv.org/abs/1704.02532.

[8] E. Santana and G. Hotz. Learning a driving simulator. https://arxiv.org/abs/1608.01230.

[9] S. Shalev-Shwartz, S. Shammah, et al. Safe, multi-agent, reinforcement learning for autonomous driving. https://arxiv.org/abs/1610.03295.

[10] N. Srivastava, G. Hinton, et al. Dropout: A simple way to prevent neural networks from overfitting. https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf.

[11] L. Torrey and J. Shavlik. Transfer learning. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.146.1515&rep=rep1&type=pdf.

[12] R. Wightman. ROS bag reader by R Wightman. https://github.com/rwightman/udacity-driving-reader.