A self-driving car using Deep learning

Course- CS G520 ADVANCED DATA MINING

Guided By
HEMANT RATHORE

Kanika Gera
2018H1030041G
h20180041@goa.bits-pilani.ac.in

Kunwar Vikrant
2018H1030050G
h20180050@goa.bits-pilani.ac.in

Satyajeet kumar
2018H1030067G
h20180067@goa.bits-pilani.ac.in

Problem Statement

Automatic navigation is an open challenging issue in the real-life traffic situation. Due to the complexity of the traffic situations and the uncertainty of the edge cases, it is hard to devise a general motion planning system for an autonomous vehicle.
There are various features of self-driving car including steering wheel, brakes, accelerometer, depending on various sensory inputs like front camera, side-view camera, dashboard camera, GPS coordinates, map-routes, etc that can be used for developing a full-fledged self-driving car.

Problem Definition

Problem is to implement a self driving car‘s steering wheel component with front board video as sensory input. We shall implement the software component of deciding the angle of rotation of the steering wheel according to the curvature of the road, and showcase our work using OpenCV.

We shall make use of various deep learning methods used to solve similar problems and compare their performances in this domain. Our goal will be to reduce the training log loss and improve accuracy.

Literature Review

1. End to End Learning for Self-Driving Cars **NVIDIA CORP.**

Objective

The aim of this paper was to develop an end-to-end model to automate various functions of a vehicle such as steering, brakes, and acceleration. The groundwork for this project was done over 10 years ago in a Defence Advanced Research Projects Agency (DARPA). In order to achieve this, a CNN model was used which could map raw pixels from a single front-facing camera to the angle of the steering of the vehicle.

This system was able to learn the useful steps for processing such as road features with only the human steering angle as the training signal. The model was never trained to specifically detect features such as the outline of the lanes. This resulted in improved performance because the internal components self-optimize to improve the overall system performance instead of focusing on improving the performance of some specific criteria like road edges detection.

Data Collection

Training data was collected by driving on a wide variety of roads and in a diverse set of lighting and weather conditions. Most road data was collected in central New Jersey, although highway data was also collected from Illinois, Michigan, Pennsylvania, and New York.

Network Architecture

The network consisted of 9 layers, including a normalization layer, 5 convolutional layers, and 3 fully connected layers. The convolutional layers were designed to perform feature extraction and were chosen through a series of experiments that varied layer configurations. We use strides in the convolutions in the first three convolutional layers with a 2×2 stride and a 5×5 kernel and a non-stride convolution with a 3×3 kernel size in the last two convolutional layers.

Results

The system learns, for example, to detect the outline of a road without the need for explicit labels during training. The System was able to achieve an autonomy value of over 90%.

2. 3D Convolutional Neural Networks for Human Action Recognition

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.442.8617&rep=rep1&type=pdf

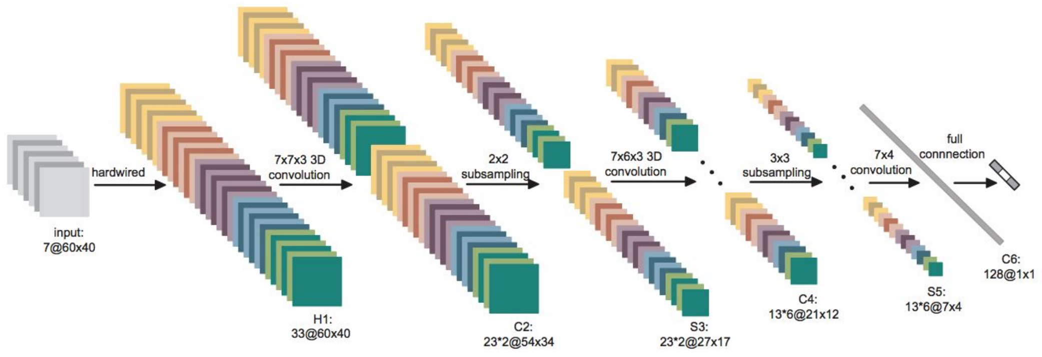
3D Convolutional NNs are introduced in this paper.

In, 2D convolution is performed at the convolutional layers to extract features from the local neighborhood on feature maps in the previous layer. Then an additive bias is applied and the result is passed through a sigmoid function.

In 2D CNNs, convolutions are applied on the 2D feature maps to compute features from the spatial dimensions only. When applied to video analysis problems, it is desirable to capture the motion information encoded in multiple contiguous frames. They have proposed to perform 3D convolutions in the convolution stages of CNNs to compute features from both spatial and temporal dimensions.

The 3D convolution is achieved by convolving a 3D kernel to the cube formed by stacking multiple contiguous frames together. By this construction, the feature maps in the convolution layer is connected to multiple contiguous frames in the previous layer, thereby capturing motion information.

Based on the 3D convolution described above, a variety of CNN architectures can be devised. Following 3D CNN architecture that was developed for human action recognition on the TRECVID data set.



The architecture shown in above figure, they have considered 7 frames of size 60×40 centered on the current frame as inputs to the 3D CNN model.

A 3D CNN architecture for human action recognition. This architecture consists of 1 hardwired layer, 3 convolution layers, 2 subsampling layers, and 1 full connection layer.

Thus they have developed a model that constructs features from both spatial and temporal dimensions using 3D convolution. The developed deep architecture generates multiple channels of information from adjacent input frames and performs convolution and subsampling separately in each channel. The final feature representation is computed by combining information from all channels.

3.Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting

https://arxiv.org/pdf/1506.04214.pdf

This paper proposes an extension of LSTM called ConvLSTM to solve the
problem considering spatiotemporal properties. LSTM alone can solve the time relationship problem but was unable to consider spatial properties.
Here the ConvLSTM is introduced to solve the problem of precipitation nowcasting using an end to end model.

In the proposed model i.e. ConvLSTM, internal multiplication matrices are replaced with convolution operation. As a result, the data that flows through the ConvLSTM cells keeps the input dimension instead of being just a 1D vector with features.

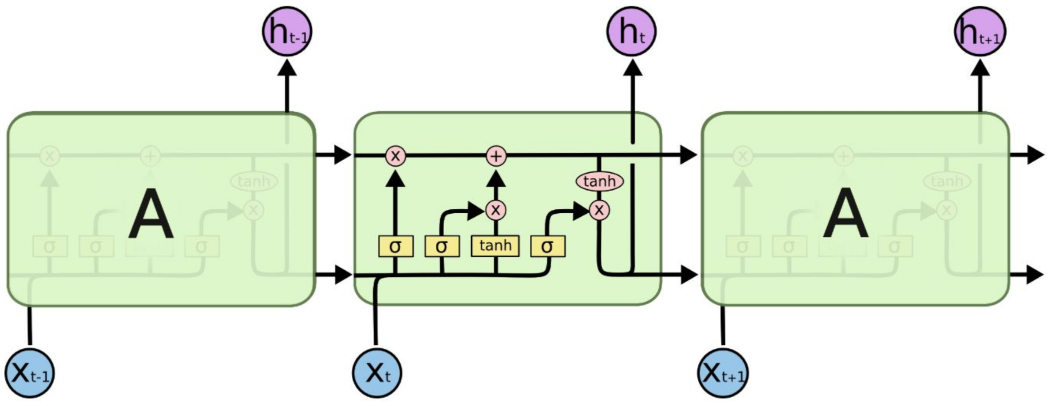


Figure 1 - Repeating Module in LSTM

The ConvLSTM, the ⊗ is replaced by ‘*’ i.e. Convolution Operator and ⊕ is replaced by ‘o’ i.e. Hadamard Operator. The equations are thus:

$$
\begin{aligned}
i_t &= \sigma(W_{xi} * \mathcal{X}_t + W_{hi} * \mathcal{H}_{t-1} + W_{ci} \circ \mathcal{C}_{t-1} + b_i) \\
f_t &= \sigma(W_{xf} * \mathcal{X}_t + W_{hf} * \mathcal{H}_{t-1} + W_{cf} \circ \mathcal{C}_{t-1} + b_f) \\
\mathcal{C}_t &= f_t \circ \mathcal{C}_{t-1} + i_t \circ \tanh(W_{xc} * \mathcal{X}_t + W_{hc} * \mathcal{H}_{t-1} + b_c) \\
o_t &= \sigma(W_{xo} * \mathcal{X}_t + W_{ho} * \mathcal{H}_{t-1} + W_{co} \circ \mathcal{C}_t + b_o) \\
\mathcal{H}_t &= o_t \circ \tanh(\mathcal{C}_t)
\end{aligned}
$$

Here ft is intermediate output coming from forgetting gate structure i .e.

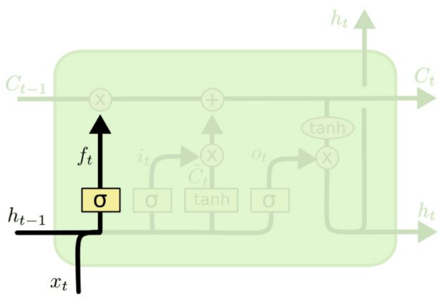


Figure 2: Forget Gate Structure

Ht-1 / ht-1 (in image) is output at a previous time and xt is input at given time t.

It is an intermediate input in the input gate structure as shown in Figure 3, and Ct is the cell output.

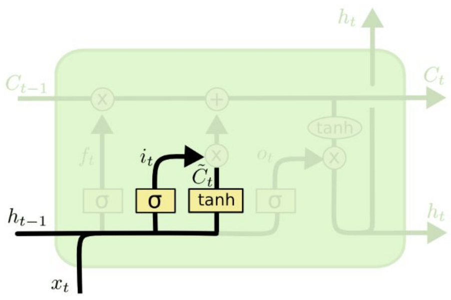


Figure 3: Input Gate Structure

Ot is input in Output Gate structure, and Ht is output at the current instant t.

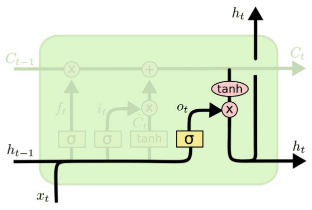


Figure 4 : Output Gate Structure

4. ImageNet Classification with Deep Convolutional Neural Networks

Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton
University of Toronto

Objective

The objective of this paper was to train a large, deep convolutional neural network to classify the 1.2 million high-resolution images in the ImageNet LSVRC-2010 contest into the 1000 different classes. To make training faster, a very efficient GPU implementation of the convolution operation was used. To reduce overfitting in the fully-connected layers we employed a recently-developed regularization method called "dropout" that proved to be very effective. The authors wrote a highly-optimized GPU implementation of 2D convolution and all the other operations inherent in training convolutional neural networks.

Dataset

ImageNet is a dataset of over 15 million labeled high-resolution images belonging to roughly 22,000categories. The images were collected from the web and labeled by human labelers using Amazon's Mechanical Turk crowd-sourcing tool. Starting in 2010, as part of the Pascal Visual Object Challenge, an annual competition called the ImageNet Large-Scale Visual Recognition Challenge (ILSVRC) has been held.

Architecture

The network contains five convolutional and three fully-connected layers, and this depth seems to be important: removing any convolutional layer (each of which contains no more than 1% of the model's parameters) resulted in inferior performance.

Results

The network achieves top-1 and top-5 test set error rates of **37.5%** and **17.0%**5. The best performance achieved during the ILSVRC- 2010 competition was 47.1% and 28.2% with an approach that averages the predictions produced from six sparse-coding models trained on different features , and since then the best-published results are 45.7% and 25.7% with an approach that averages the predictions of two classifiers trained on Fisher Vectors (FVs) computed from two types of densely-sampled features

5. Autonomous Off-Road Vehicle Control Using End-to-End Learning Net-Scale Technologies, Inc.

Objective

The purpose of this project was to perform a short term and strongly focused effort to explore the benefits of end-to-end machine learning for this application. A test vehicle was built, consisting of a commercially available radio-controlled model truck which was equipped with two lightweight video cameras and other sensors. The vehicle's task was to drive on unknown open terrain while avoiding any obstacles such as rocks, trees, ditches, and ponds.

Data Collection

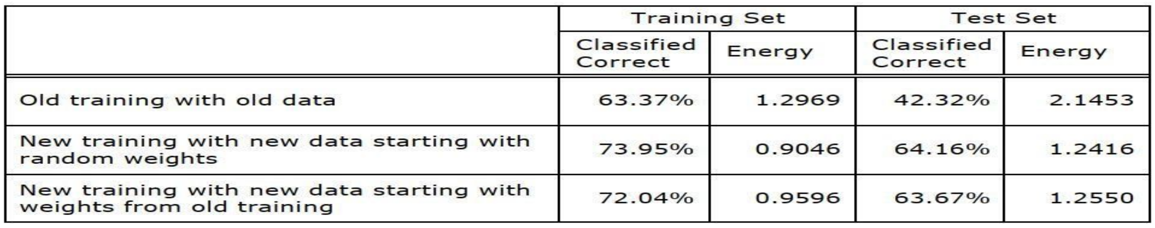
Data collection was done by a human driver who drove the vehicle remotely from the PC with the joystick. The driver could see the image of one video camera (no stereo vision) while driving. During each training run the PC recorded the output of the two video cameras together with the steering wheel position and throttle of the joystick and other sensors. Only the data from the video cameras and the steering wheel angle was used for supervised learning. The other sensors were only needed during reinforcement learning.

Architecture

The actual network used for the supervised learning in this project was organized into six layers with feature maps where feature maps become progressively smaller with each layer. The input layer consists of the raw image data that was sub-sampled to 149*58 pixels. The output consists of two 1*1 feature maps which represent the two driving directions left and right.

Results

The following results were obtained. Here 'Energy' represents the Euclidean distance between the output produced by the network and the target output averaged over all input patterns (frames).



	Training Set		Test Set	
	Classified Correct	Energy	Classified Correct	Energy
Old training with old data	63.37%	1.2969	42.32%	2.1453
New training with new data starting with random weights	73.95%	0.9046	64.16%	1.2416
New training with new data starting with weights from old training	72.04%	0.9596	63.67%	1.2550

6. Alvinn : An Autonomous Land Vehicle in A Neural Network

Dean A. Pomerleau Computer Science Department Carnegie Mellon University Pittsburgh, PA 15213

Objective

ALVINN (Autonomous Land Vehicle In a Neural Network) is a connectionist approach to the navigational task of road following. Specifically, ALVINN was an artificial neural network designed to control the NAVLAB, the Carnegie Mellon autonomous navigation test vehicle

Data Collection

Since it was during the 80's, data collection was difficult back then, hence the department of CMU developed a simulated road generator which created road images to be used as training exemplars for the network. the road generator also created corresponding simulated range finder images

Network training was performed using these artificial road "snapshots" Training involved creating a set of 1200 road snapshots depicting roads with a wide variety of retinal orientations and positions, under a variety of lighting conditions and with realistic noise levels.

Architecture

ALVINN's architecture consisted of a single hidden layer back-propagation network. The input layer is divided into three sets of units: two "retinas" and a single intensity feedback unit. The first retina,

consisting of 3002 units, receives video camera input from a road scene. The second retina, consisting of 8x32 units, receives input from a laser range finder. The output layer consists of 46 units, divided into two groups. The first set of 45 units is a linear representation of the tum curvature along which the vehicle should travel in order to head towards the road center.

Results

After 40 epochs of training on the 1200 simulated road snapshots, the network correctly dictated a turn curvature within two units of the correct answer approximately 90% of the time on novel simulated road images.

7. Learning a Driving Simulator

Eder Santana ⁎**George Hotz** University of Florida comma.ai eder@comma.ai

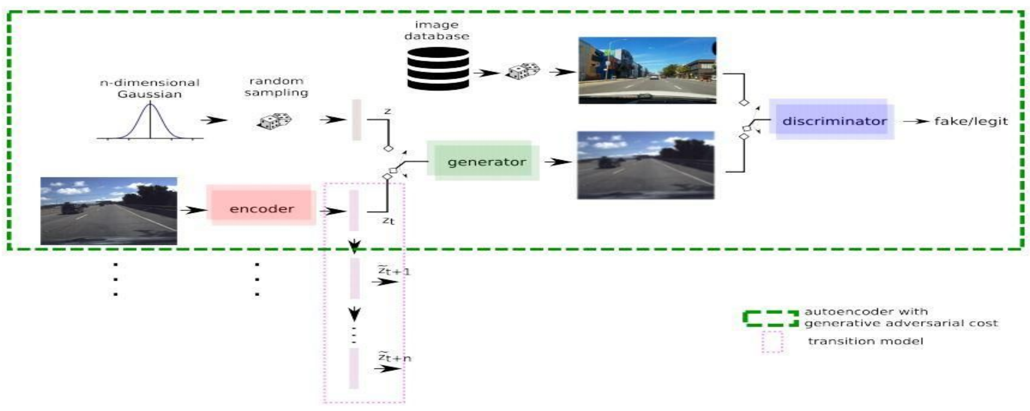
Objective

The paper focuses on generating video streams of a front-facing camera mounted in the car windshield. This paper contributes to this learned video simulation literature. No graphics engine or world model assumptions are made. We show that it is possible to train a model to do realistic-looking video predictions while calculating a low dimensional compact representation and action conditioned transitions. We leave controls on the simulated world for future work. In the next section, we describe the dataset we used to study video prediction of real-world road videos.

Dataset

The dataset was mounted using a Point Grey camera in the windshield of an Acura ILX 2016, capturing pictures of the road at 20 Hz. In the released dataset there is a total of 7.25 hours of driving data divided into 11 videos. The released video frames are 160 × 320 pixels regions from the middle of the captured screen. Beyond video, the dataset also has several sensors that were measured in different frequencies and interpolated to 100Hz. Example data coming from sensors are the car speed, steering angle, GPS, gyroscope, IMU, etc. Further details about the dataset and measurement equipment can be found online in the companion website.

Architecture



Results

Models were trained using (a) generative adversarial networks cost function (b) mean square error. Both models have MSE in the order of 10^-2.

8. MIT Advanced Vehicle Technology Study

Objective

The MIT Advanced Vehicle Technology (MIT-AVT) study seeks to collect and analyze large-scale naturalistic data of semi-autonomous driving in order to better characterize the state of current technology use, to extract insight on how automation-enabled technologies impact human-machine interaction across a range of environments. The goal is to propose, design, and build systems grounded in this understanding, so that shared autonomy between human and vehicle AI does not lead to a series of unintended consequences.

Dataset

ImageNet is an image dataset based on WordNet where 100,000 synonym sets (or "synsets") each define a unique meaningful concept. The goal for ImageNet is to have 1000 annotated images for each of the 100,000 synsets. Currently, it has 21,841 synsets with images and a total of 14,197,122 images. This dataset is commonly used to train a neural network for image classification and object detection tasks. The best performing networks are highlighted as part of the annual ImageNet Large Scale Visual Recognition Competition (ILSVRC). Several deep learning models were trained and tested on these datasets.

9. End-to-End Learning of Driving Models with Surround-View Cameras and Route Planners

https://arxiv.org/pdf/1803.10158.pdf

In this paper, Baidu has developed an automated driving training model based on the Conv-LSTM network structure. The idea was to use feature extraction capabilities from convolution network and memory ability from the LSTM network.

In their experiment a synchronized sample is developed that contains four frames at a resolution of 256 × 256 for the corresponding front, left, right and rear-facing cameras that a rendered image at 256×256 pixels for TomTom route planner.*TomTom Route Planner is a route-finding program that covers more than 40 European countries.*
They had trained a model using Adam Optimizer with an initial learning rate of 10^{-4} and a batch size of 16 for 5 epochs. The structure of the network used to train is not public.

10. What is the Best Multi-Stage Architecture for Object Recognition?

Kevin Jarrett, Koray Kavukcuoglu, Marc'Aurelio Ranzato and Yann LeCunThe Courant Institute of Mathematical Sciences New York University, 715 Broadway, New York, NY 10003, USA koray@cs.nyu.edu

Objective

This paper addresses three questions: 1. How do the nonlinearities that follow the filter banks influence the recognition accuracy? 2. Does learning the filter banks in an unsupervised or supervised manner improve the performance over hard-wired filters or even random filters? 3. Is there any advantage to using an architecture with two successive stages of feature extraction, rather than with a single stage.

Dataset

Results are presented on the well-known Caltech-101 dataset, on the NORB object dataset, and on the MNIST dataset of handwritten digits.

Architecture

The hierarchy stacks one or several feature extraction stages, each of which consists of a filter bank layer, non-linear transformation layers, and a pooling layer that combines filter responses over local neighborhoods using an average or max operation, thereby achieving invariance to small distortions.

Results

A two-stage system with random filters yields an almost 63% recognition rate on Caltech-101, provided that the proper non-linearities and pooling layers are used. With supervised refinement, the system achieves state-of-the-art performance on the NORB dataset (5.6%) and unsupervised pre-training followed by supervised refinement produces good accuracy on Caltech-101 (> 65%), and the lowest known error rate on the undistorted, unprocessed MNIST dataset (0.53%).

Action Plan

Approach - 1

Develop an end to end Convolution Neural Network to implement steering wheel rotation in a self-driving car using architecture in reference 1. In addition to the architecture perform parameter tuning like adding dropout layers, changing Adam optimizer, etc.

Approach - 2

Use of temporal-spatial information to improve the self-driving car with the help of a 3D Convolution network. Reference 2 shows that 3D Convolution has outstanding capabilities for the extraction of spatial features.

Approach - 3

Conv-LSTM can make use of timing relationships with the help of the LSTM component and can characterize local spatial features with the help of CNN. A similar idea will be used for our problem statement.

PHASE - 2

Objective

The objective of this phase is to develop an end to end Convolution Neural Network to implement steering wheel rotation in a self-driving car. In addition to the architecture, we performed hyper-parameter tuning like adding dropout layers, changing optimizer, etc.

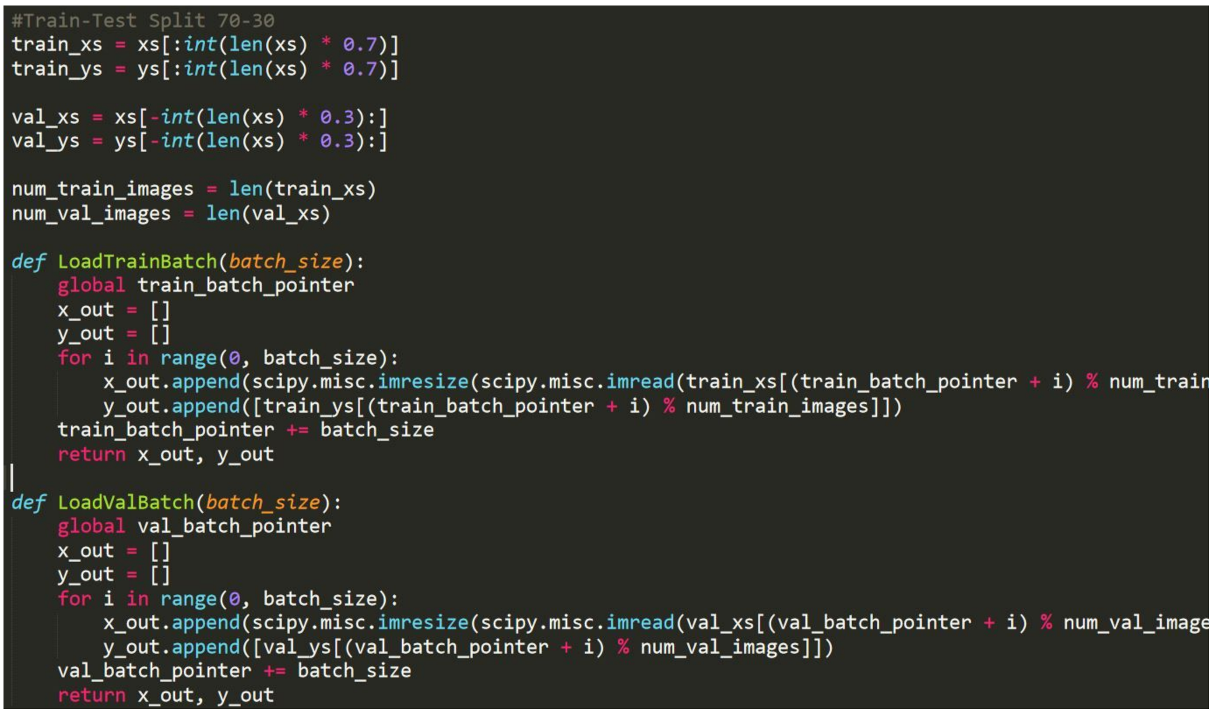
Data

We have used Sully Chen driving datasets for our self-driving car model. Dataset consists of images recorded from a car dashcam with labeled steering angles.

Data Preprocessing

For the preprocessing step, we first loaded all out images and resized them to 66*200 pixels. Then we normalized the pixel values of the images using the MinMaxScaler technique.

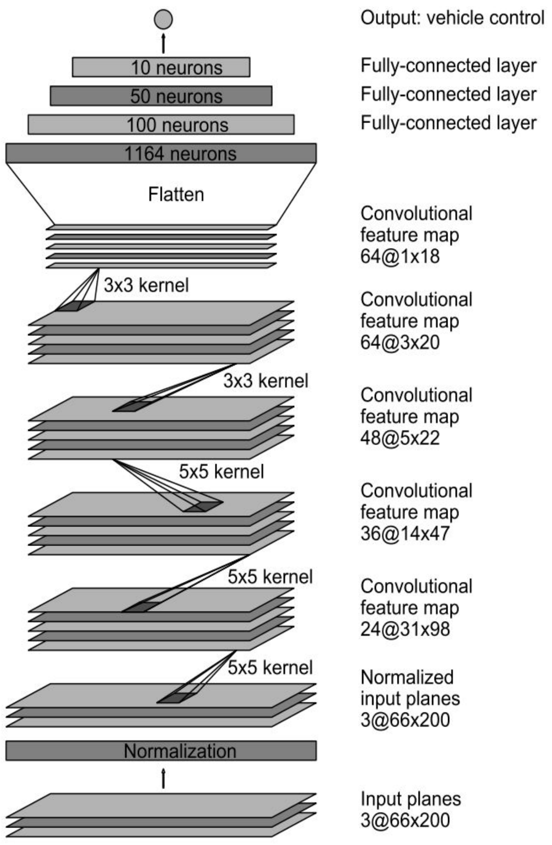
Code



```
#Train-Test Split 70-30
train_xs = xs[:int(len(xs) * 0.7)]
train_ys = ys[:int(len(xs) * 0.7)]

val_xs = xs[-int(len(xs) * 0.3):]
val_ys = ys[-int(len(xs) * 0.3):]

num_train_images = len(train_xs)
num_val_images = len(val_xs)

def LoadTrainBatch(batch_size):
    global train_batch_pointer
    x_out = []
    y_out = []
    for i in range(0, batch_size):
        x_out.append(scipy.misc.imresize(scipy.misc.imread(train_xs[(train_batch_pointer + i) % num_train
        y_out.append([train_ys[(train_batch_pointer + i) % num_train_images]])
    train_batch_pointer += batch_size
    return x_out, y_out

def LoadValBatch(batch_size):
    global val_batch_pointer
    x_out = []
    y_out = []
    for i in range(0, batch_size):
        x_out.append(scipy.misc.imresize(scipy.misc.imread(val_xs[(val_batch_pointer + i) % num_val_image
        y_out.append([val_ys[(val_batch_pointer + i) % num_val_images]])
    val_batch_pointer += batch_size
    return x_out, y_out
```

We split our dataset into a 70:30 ratio of the training set and Cross-Validation set. For training, we used a batch size of 100 and number of epochs as 30 for training the model.

Network Architecture



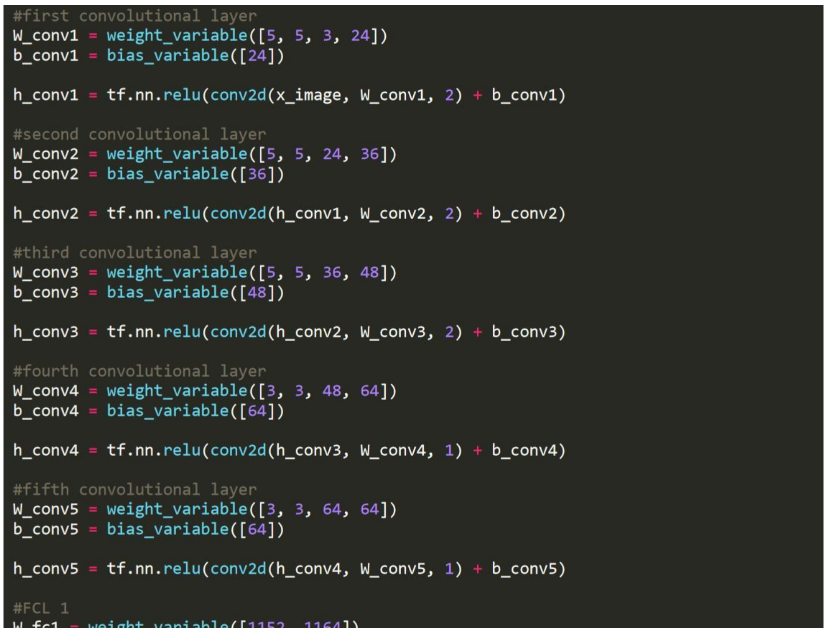
Our model consisted of 5 Convolutional Layers with Kernel size of 5 X 5 and 3X3 as shown in the figure above. Implementation of this was done using google TensorFlow library. In addition to using the convolution layer, we added untied bias on each neural network, with the size equal to the size of the output. The dropout layer was added after each fully connected layer, where we took a dropout rate equal to 0.5. Various Adam optimizer values were used to experiment, including 1^e-4, 1^e-3.

Input Layer

Input Kernel is of shape [Batch_Size, Width, Height, Channels], we took 100 as batch size 100 images of 66X200X3 are sent together in the network.
Therefore input tuple is of size **[batch_size, 66,200,3]**.

Convolution Layers

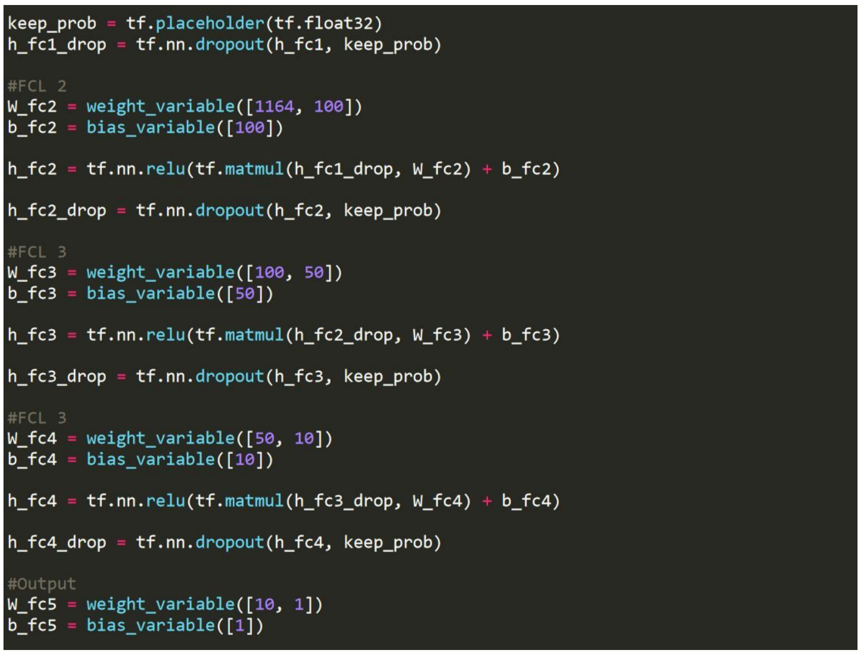
The kernel in 2D Convolution is of shape **[W_kernel, H_kernel, in_channels, out_channels],** We used 2D Convolution kernels as described in paper 1, the shape was [5,5,3,24], and so on as shown in the figure.



```
#first convolutional layer
W_conv1 = weight_variable([5, 5, 3, 24])
b_conv1 = bias_variable([24])

h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1, 2) + b_conv1)

#second convolutional layer
W_conv2 = weight_variable([5, 5, 24, 36])
b_conv2 = bias_variable([36])

h_conv2 = tf.nn.relu(conv2d(h_conv1, W_conv2, 2) + b_conv2)

#third convolutional layer
W_conv3 = weight_variable([5, 5, 36, 48])
b_conv3 = bias_variable([48])

h_conv3 = tf.nn.relu(conv2d(h_conv2, W_conv3, 2) + b_conv3)

#fourth convolutional layer
W_conv4 = weight_variable([3, 3, 48, 64])
b_conv4 = bias_variable([64])

h_conv4 = tf.nn.relu(conv2d(h_conv3, W_conv4, 1) + b_conv4)

#fifth convolutional layer
W_conv5 = weight_variable([3, 3, 64, 64])
b_conv5 = bias_variable([64])

h_conv5 = tf.nn.relu(conv2d(h_conv4, W_conv5, 1) + b_conv5)

#FCL 1
W_fc1 = weight_variable([1152, 1164])
```

Fully Connected Layers

Fully Connected Layer of 1164 neurons is used first. Last convolution layer was of shape [batch_size,1,18,64]. Hence 1152 X 1164 trainable parameters exist during flattening. All other FC layer's structure is shown in the figure



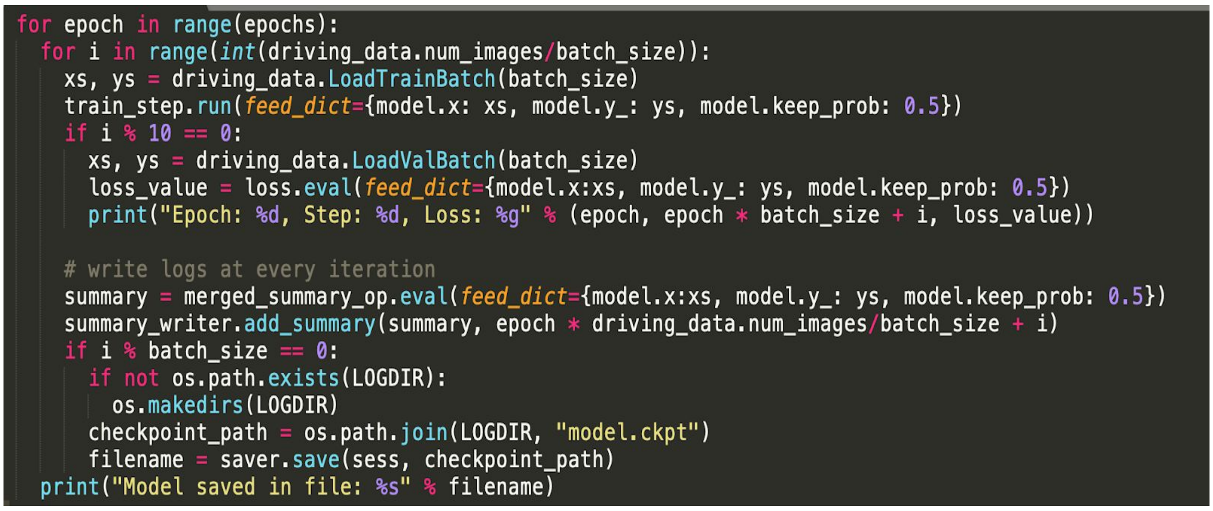
```
keep_prob = tf.placeholder(tf.float32)
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

#FCL 2
W_fc2 = weight_variable([1164, 100])
b_fc2 = bias_variable([100])

h_fc2 = tf.nn.relu(tf.matmul(h_fc1_drop, W_fc2) + b_fc2)

h_fc2_drop = tf.nn.dropout(h_fc2, keep_prob)

#FCL 3
W_fc3 = weight_variable([100, 50])
b_fc3 = bias_variable([50])

h_fc3 = tf.nn.relu(tf.matmul(h_fc2_drop, W_fc3) + b_fc3)

h_fc3_drop = tf.nn.dropout(h_fc3, keep_prob)

#FCL 3
W_fc4 = weight_variable([50, 10])
b_fc4 = bias_variable([10])

h_fc4 = tf.nn.relu(tf.matmul(h_fc3_drop, W_fc4) + b_fc4)

h_fc4_drop = tf.nn.dropout(h_fc4, keep_prob)

#Output
W_fc5 = weight_variable([10, 1])
b_fc5 = bias_variable([1])
```

Training

Machine Used :

2.3 GHz Intel Core i5 X 8 core processor, was initially used to train model, which took about 12-16 hours. Later we switched to Google Cloud 16 core processor, 16 GB RAM with NVIDIA TESLA v100 GPU, which reduced time to 3 to 4 hours of training.

Training Code:

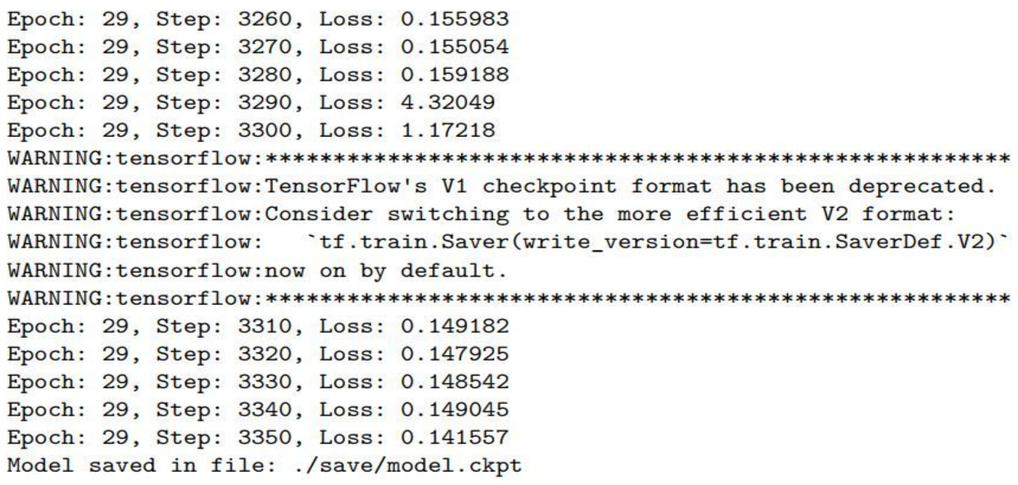


```
for epoch in range(epochs):
  for i in range(int(driving_data.num_images/batch_size)):
    xs, ys = driving_data.LoadTrainBatch(batch_size)
    train_step.run(feed_dict={model.x: xs, model.y_: ys, model.keep_prob: 0.5})
    if i % 10 == 0:
      xs, ys = driving_data.LoadValBatch(batch_size)
      loss_value = loss.eval(feed_dict={model.x:xs, model.y_: ys, model.keep_prob: 0.5})
      print("Epoch: %d, Step: %d, Loss: %g" % (epoch, epoch * batch_size + i, loss_value))

    # write logs at every iteration
    summary = merged_summary_op.eval(feed_dict={model.x:xs, model.y_: ys, model.keep_prob: 0.5})
    summary_writer.add_summary(summary, epoch * driving_data.num_images/batch_size + i)
    if i % batch_size == 0:
      if not os.path.exists(LOGDIR):
        os.makedirs(LOGDIR)
      checkpoint_path = os.path.join(LOGDIR, "model.ckpt")
      filename = saver.save(sess, checkpoint_path)
  print("Model saved in file: %s" % filename)
```

Results

We successfully trained the model to predict the steering angle according to the image of the road given at an instance. The training loss achieved is 0.14. The best result was achieved with the value of Adam optimizer e^-4.

Training Loss: 0.14



```
Epoch: 29, Step: 3260, Loss: 0.155983
Epoch: 29, Step: 3270, Loss: 0.155054
Epoch: 29, Step: 3280, Loss: 0.159188
Epoch: 29, Step: 3290, Loss: 4.32049
Epoch: 29, Step: 3300, Loss: 1.17218
WARNING:tensorflow:*******************************************************
WARNING:tensorflow:TensorFlow's V1 checkpoint format has been deprecated.
WARNING:tensorflow:Consider switching to the more efficient V2 format:
WARNING:tensorflow:   `tf.train.Saver(write_version=tf.train.SaverDef.V2)`
WARNING:tensorflow:now on by default.
WARNING:tensorflow:*******************************************************
Epoch: 29, Step: 3310, Loss: 0.149182
Epoch: 29, Step: 3320, Loss: 0.147925
Epoch: 29, Step: 3330, Loss: 0.148542
Epoch: 29, Step: 3340, Loss: 0.149045
Epoch: 29, Step: 3350, Loss: 0.141557
Model saved in file: ./save/model.ckpt
```

Plot

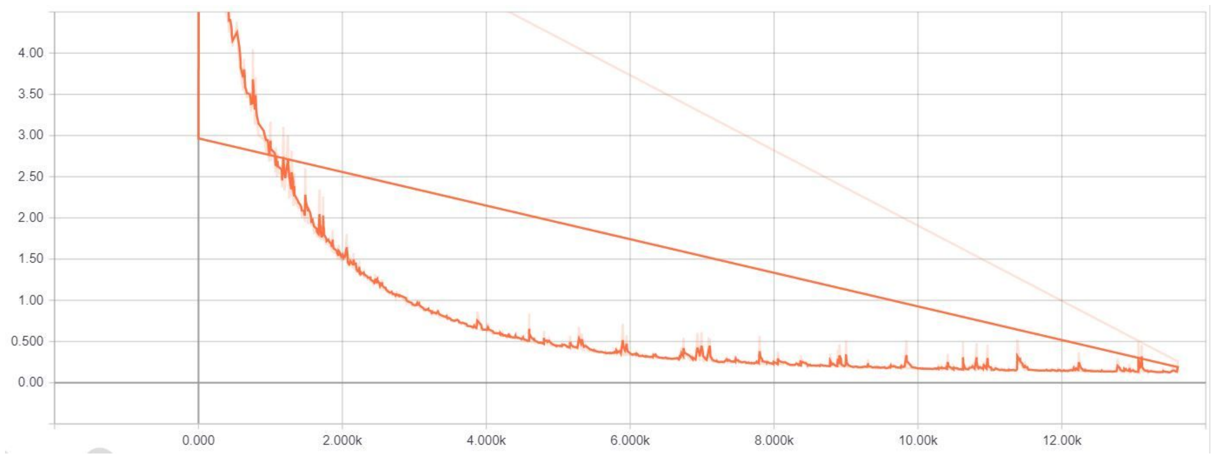


Figure: Validation Loss Graph for 2D CNN Architecture

PHASE- 3

Objective

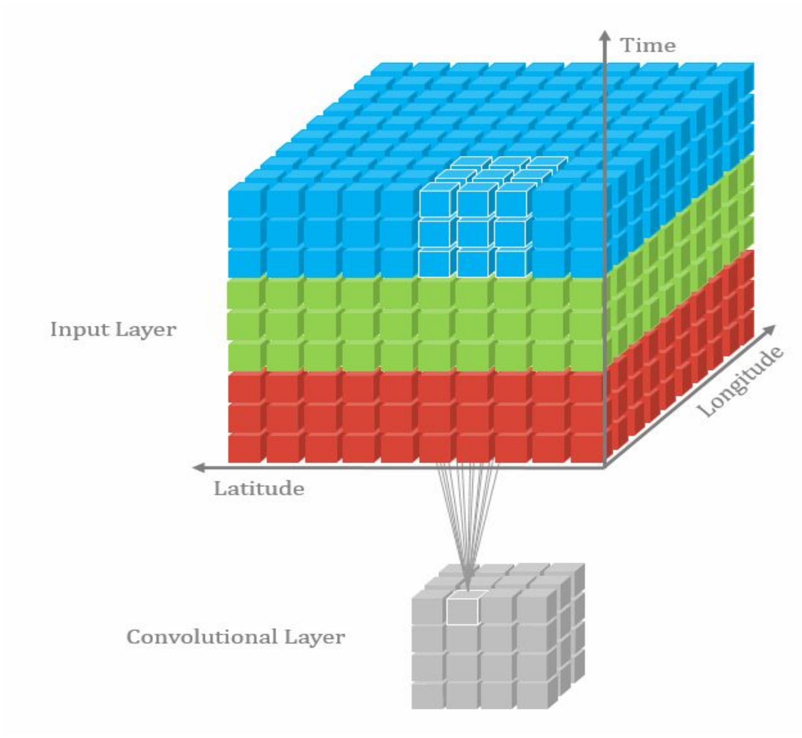
The objective of this phase is to develop an end to end 3D Convolution Neural Network that can capture temporal-spatial information to implement steering wheel rotation in a self-driving car.

Building Model

For the purpose of this phase, we are changing our model. We are using 3D-CNNs in this phase to observe if we can achieve better results using different architecture.

The goal of 3D CNN is to take as an input a video and extract features from it. While ConvNets extract the graphical characteristics of a single image and put them in a vector (a low-level representation), 3D CNNs extract the graphical characteristics of a set of images.

3D CNNs take into account a temporal dimension i.e the order of the images in the video. From a set of images, 3D CNNs find a low-level representation of a set of images, and this representation is useful to find the right label of the video. In order to extract such features, 3D convolution uses 3Dconvolution operations



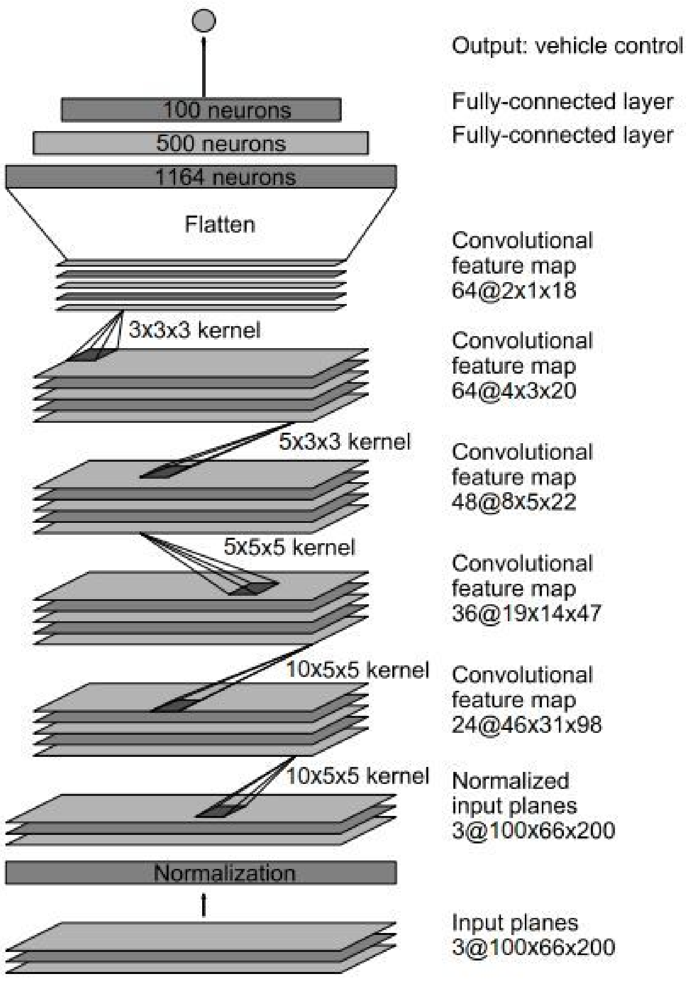
Data

We have used Sully Chen driving datasets for our self-driving car model. Dataset consists of images recorded from a car dashcam with labeled steering angles.

Data Preprocessing

Images are resized to 66 x 200. MinMaxScaler technique is used to normalize the pixel values of the images.

Network Structure



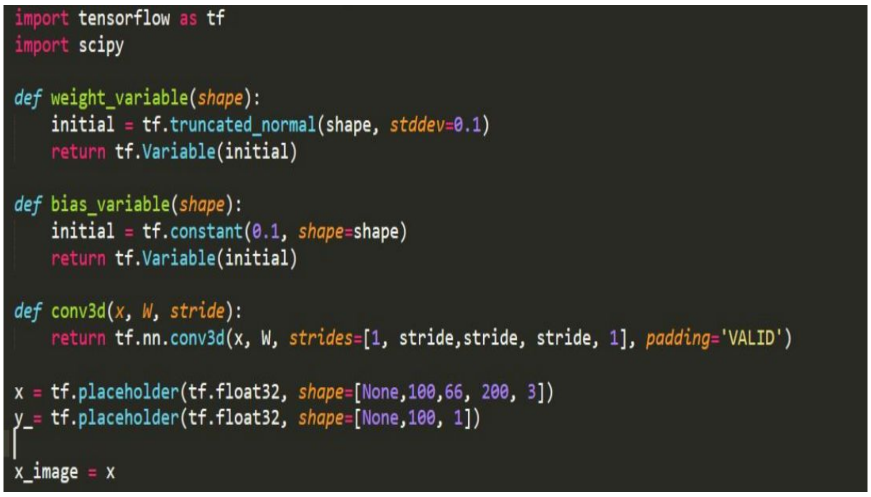
Output: vehicle control
100 neurons
Fully-connected layer
500 neurons
Fully-connected layer
1164 neurons
Flatten
Convolutional feature map 64@2x1x18
3x3x3 kernel
Convolutional feature map 64@4x3x20
5x3x3 kernel
Convolutional feature map 48@8x5x22
5x5x5 kernel
Convolutional feature map 36@19x14x47
10x5x5 kernel
Convolutional feature map 24@46x31x98
10x5x5 kernel
Normalized input planes 3@100x66x200
Normalization
Input planes 3@100x66x200

Figure - 3D CNN Network Architecture

Input Layer

3D Convolution has one extra dimension, hence images are reshaped according to [Batch_Size, Time, Width, Height, Channels], we took 100 images for time dimension i.e. 100 images of 66X200X3 are sent together in the network.

Therefore input tuple is of size **[batch_size, 100, 66,200,3]**.

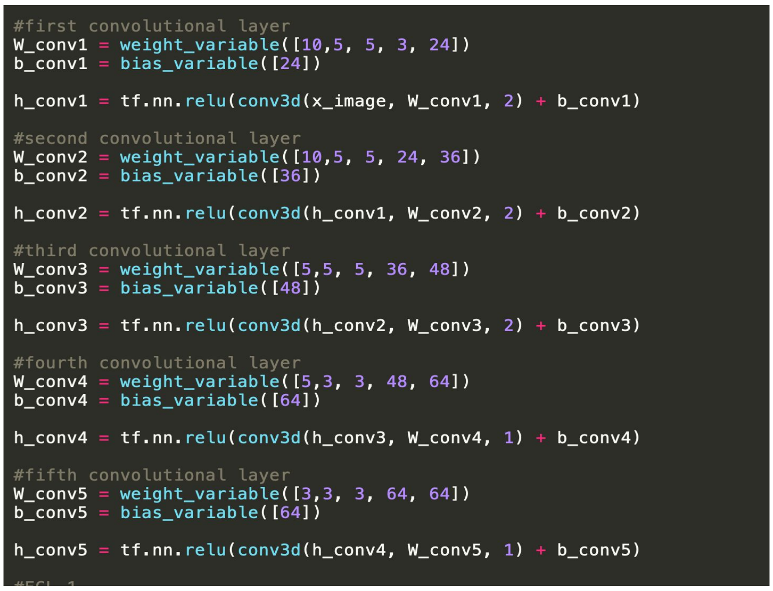


```python
import tensorflow as tf
import scipy

def weight_variable(shape):
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)

def bias_variable(shape):
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)

def conv3d(x, W, stride):
    return tf.nn.conv3d(x, W, strides=[1, stride,stride, stride, 1], padding='VALID')

x = tf.placeholder(tf.float32, shape=[None,100,66, 200, 3])
y_= tf.placeholder(tf.float32, shape=[None,100, 1])

x_image = x
```

Figure - Input Layer Code

Convolution Layers

The kernel in 3D Convolution is of shape **[T_kernel, W_kernel, H_kernel, in_channels, out_channels],** where T_kernel is time dimension shape of the kernel. In phase-2 we used 2D Convolution where kernel shape was [5,5,3,24], here we added one extra dimension i.e. T_kernel, the shape of the new convolution layer-1 is thus [10,5, 5, 3, 24] and so on as shown in the figure.

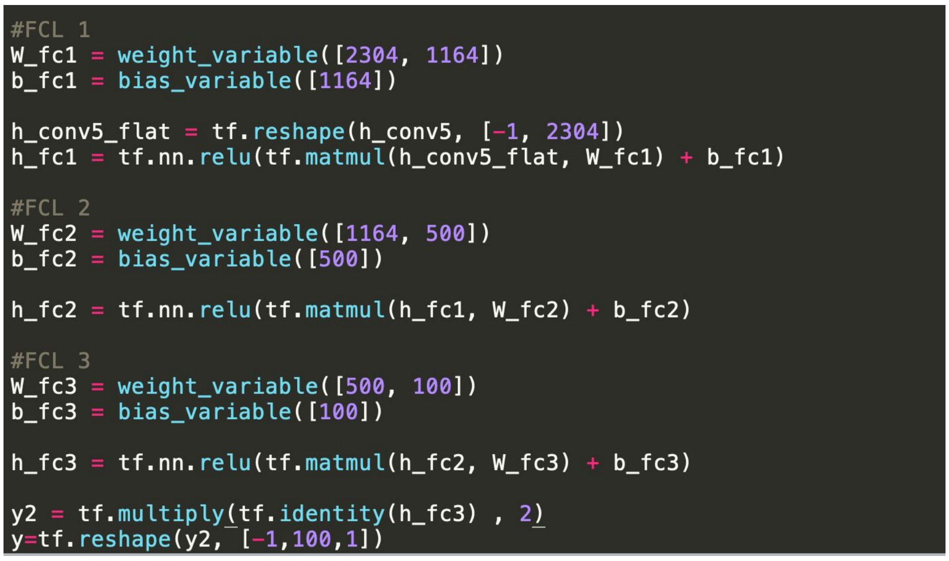


```python
#first convolutional layer
W_conv1 = weight_variable([10,5, 5, 3, 24])
b_conv1 = bias_variable([24])

h_conv1 = tf.nn.relu(conv3d(x_image, W_conv1, 2) + b_conv1)

#second convolutional layer
W_conv2 = weight_variable([10,5, 5, 24, 36])
b_conv2 = bias_variable([36])

h_conv2 = tf.nn.relu(conv3d(h_conv1, W_conv2, 2) + b_conv2)

#third convolutional layer
W_conv3 = weight_variable([5,5, 5, 36, 48])
b_conv3 = bias_variable([48])

h_conv3 = tf.nn.relu(conv3d(h_conv2, W_conv3, 2) + b_conv3)

#fourth convolutional layer
W_conv4 = weight_variable([5,3, 3, 48, 64])
b_conv4 = bias_variable([64])

h_conv4 = tf.nn.relu(conv3d(h_conv3, W_conv4, 1) + b_conv4)

#fifth convolutional layer
W_conv5 = weight_variable([3,3, 3, 64, 64])
b_conv5 = bias_variable([64])

h_conv5 = tf.nn.relu(conv3d(h_conv4, W_conv5, 1) + b_conv5)
```

Figure - 3D Convolution Layers

Fully Connected layers

Fully Connected Layer of 1164 neurons is used first. Last convolution layer was of shape [batch_size,2,1,18,64]. Hence 2304 X 1164 trainable parameters exist during flattening. All other FC layer’s structure is shown in the figure.



```
#FCL 1
W_fc1 = weight_variable([2304, 1164])
b_fc1 = bias_variable([1164])

h_conv5_flat = tf.reshape(h_conv5, [-1, 2304])
h_fc1 = tf.nn.relu(tf.matmul(h_conv5_flat, W_fc1) + b_fc1)

#FCL 2
W_fc2 = weight_variable([1164, 500])
b_fc2 = bias_variable([500])

h_fc2 = tf.nn.relu(tf.matmul(h_fc1, W_fc2) + b_fc2)

#FCL 3
W_fc3 = weight_variable([500, 100])
b_fc3 = bias_variable([100])

h_fc3 = tf.nn.relu(tf.matmul(h_fc2, W_fc3) + b_fc3)

y2 = tf.multiply(tf.identity(h_fc3) , 2)
y=tf.reshape(y2, [-1,100,1])
```

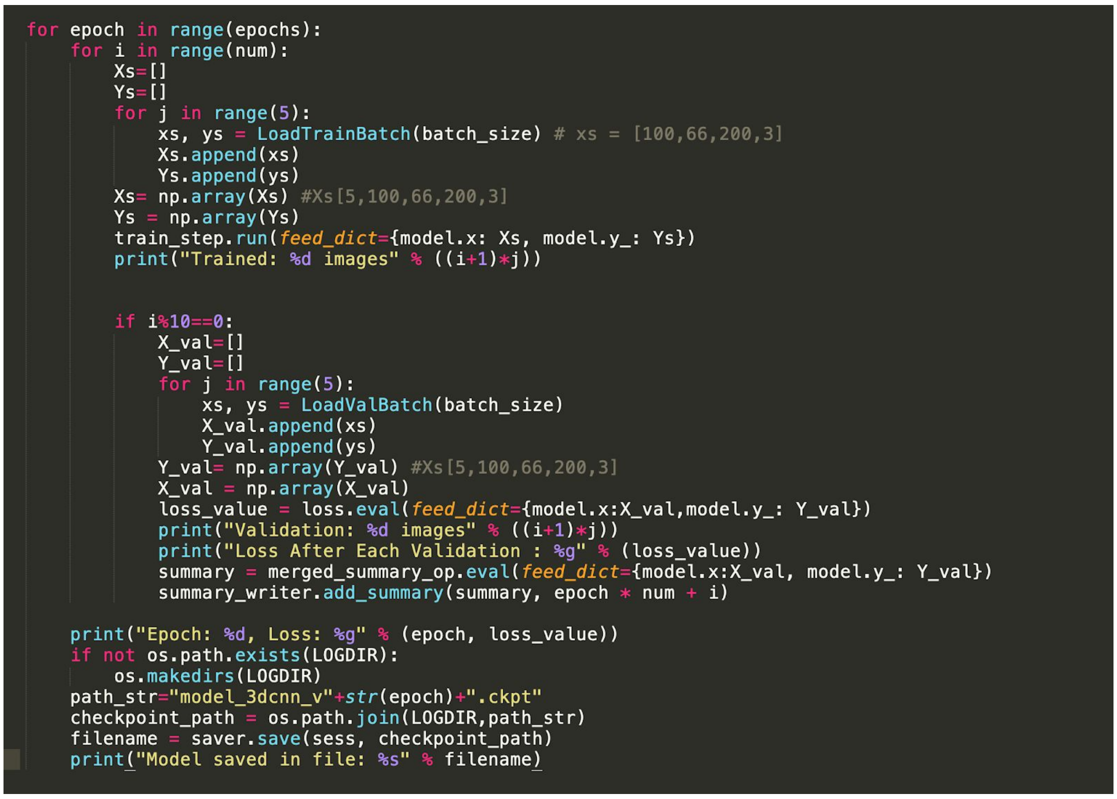
Figure - Fully Connected Layers

Training

Machine Used :

- 2.3 GHz Intel Core i5 X 8 core processor, was initially used to train model, which took about 12-16 hours.
- Google Cloud 16 core processor skylake processor, 16 Gb RAM with NVIDIA TESLA T4 GPU, which reduced time to 4 to 4.5 hours of training.

Training Code:



```
for epoch in range(epochs):
    for i in range(num):
        Xs=[]
        Ys=[]
        for j in range(5):
            xs, ys = LoadTrainBatch(batch_size) # xs = [100,66,200,3]
            Xs.append(xs)
            Ys.append(ys)
        Xs= np.array(Xs) #Xs[5,100,66,200,3]
        Ys = np.array(Ys)
        train_step.run(feed_dict={model.x: Xs, model.y_: Ys})
        print("Trained: %d images" % ((i+1)*j))


        if i%10==0:
            X_val=[]
            Y_val=[]
            for j in range(5):
                xs, ys = LoadValBatch(batch_size)
                X_val.append(xs)
                Y_val.append(ys)
            Y_val= np.array(Y_val) #Xs[5,100,66,200,3]
            X_val = np.array(X_val)
            loss_value = loss.eval(feed_dict={model.x:X_val,model.y_: Y_val})
            print("Validation: %d images" % ((i+1)*j))
            print("Loss After Each Validation : %g" % (loss_value))
            summary = merged_summary_op.eval(feed_dict={model.x:X_val, model.y_: Y_val})
            summary_writer.add_summary(summary, epoch * num + i)

    print("Epoch: %d, Loss: %g" % (epoch, loss_value))
    if not os.path.exists(LOGDIR):
        os.makedirs(LOGDIR)
    path_str="model_3dcnn_v"+str(epoch)+".ckpt"
    checkpoint_path = os.path.join(LOGDIR,path_str)
    filename = saver.save(sess, checkpoint_path)
    print("Model saved in file: %s" % filename)
```

Results

We successfully trained the model to predict the steering angle according to the image of the road given at an instance. The training loss achieved is 12.07 which is more than the training loss achieved in 2D CNN i.e. 0.14.

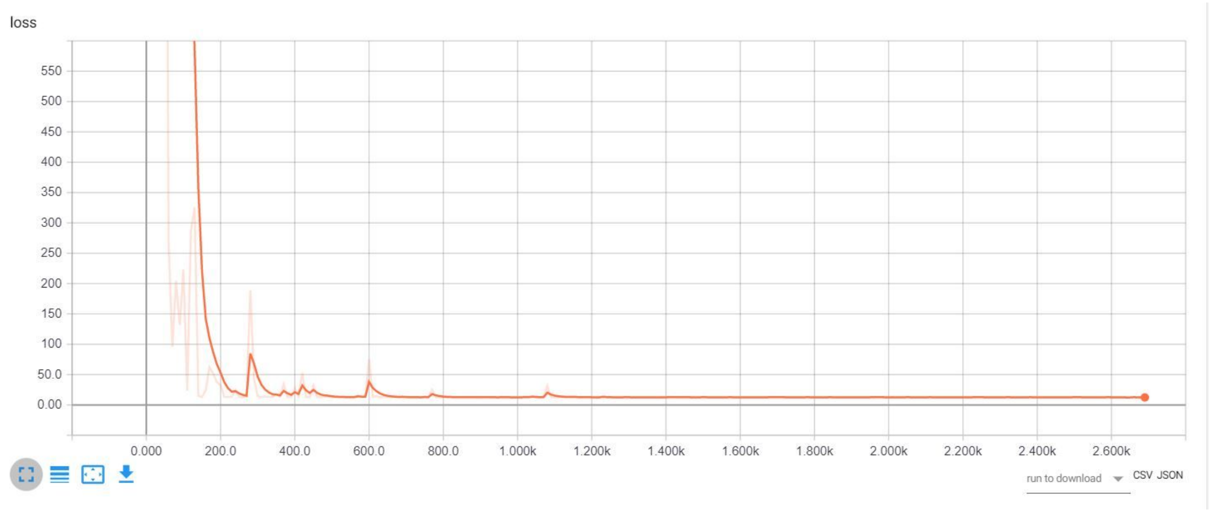


Figure - Validation Loss Graph

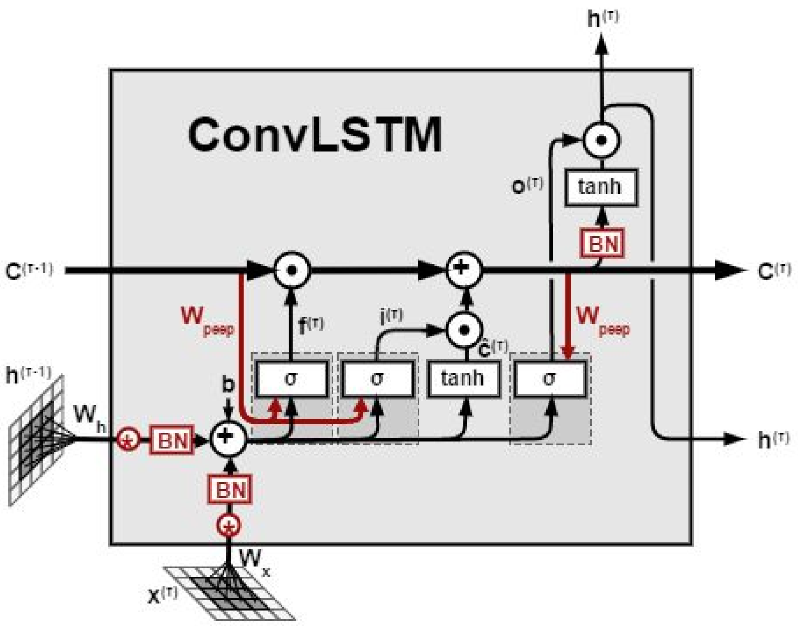
PHASE- 4

Objective

The objective of this phase is to develop an end to end Conv2D-LSTM model architecture. The use of Conv2d-LSTM will allow the model to capture temporal features present in the input as well.

Building Model

For the purpose of this phase, we are changing our model to Conv2d-LSTM to observe if we can achieve better results using different architecture.
The CNN LSTM architecture involves using Convolutional Neural Network (CNN) layers for feature extraction on input data combined with LSTMs to support sequence prediction.
ConvLSTM is a Recurrent layer, just like the LSTM, but internal matrix multiplications are exchanged with convolution operations. As a result, the data that flows through the ConvLSTM cells keeps the input dimension instead of being just a 1D vector with features.



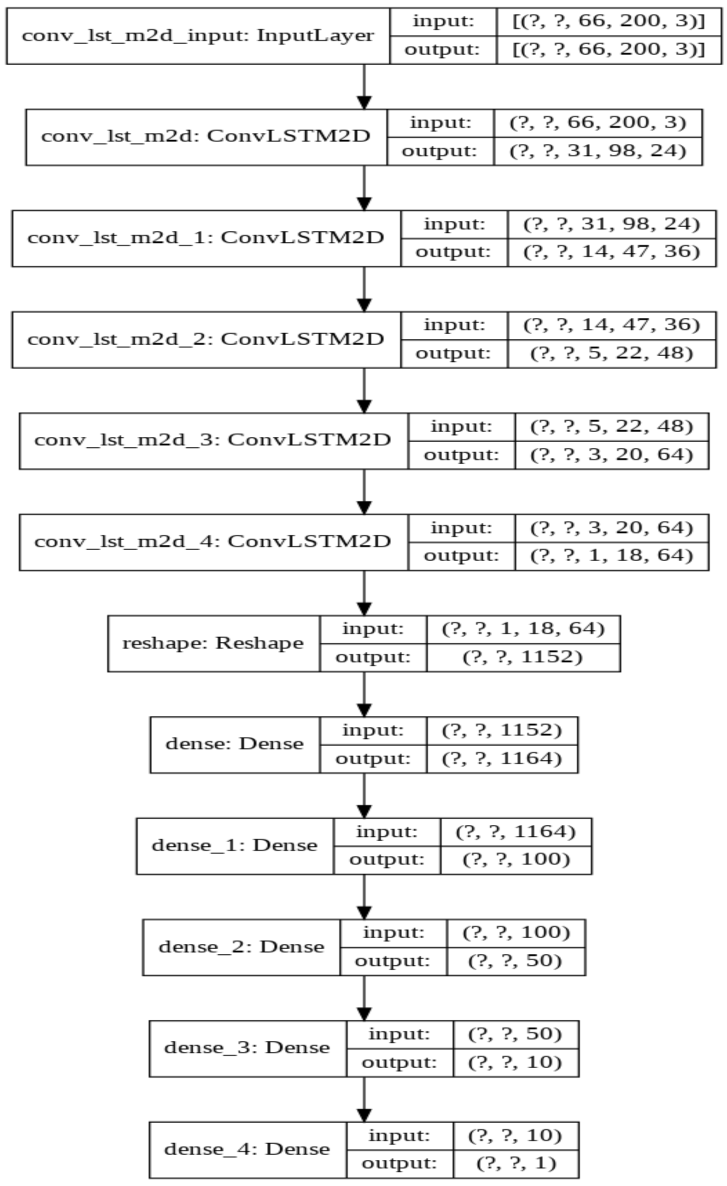
Data

We have used Sully Chen driving datasets for our self-driving car model. Dataset consists of images recorded from a car dashcam with labeled steering angles.

Data Preprocessing

Images are resized to 66 x 200. MinMaxScaler technique is used to normalize the pixel values of the images.

Network Structure



conv_lst_m2d_input: InputLayer | input: [(?, ?, 66, 200, 3)] | output: [(?, ?, 66, 200, 3)]
conv_lst_m2d: ConvLSTM2D | input: (?, ?, 66, 200, 3) | output: (?, ?, 31, 98, 24)
conv_lst_m2d_1: ConvLSTM2D | input: (?, ?, 31, 98, 24) | output: (?, ?, 14, 47, 36)
conv_lst_m2d_2: ConvLSTM2D | input: (?, ?, 14, 47, 36) | output: (?, ?, 5, 22, 48)
conv_lst_m2d_3: ConvLSTM2D | input: (?, ?, 5, 22, 48) | output: (?, ?, 3, 20, 64)
conv_lst_m2d_4: ConvLSTM2D | input: (?, ?, 3, 20, 64) | output: (?, ?, 1, 18, 64)
reshape: Reshape | input: (?, ?, 1, 18, 64) | output: (?, ?, 1152)
dense: Dense | input: (?, ?, 1152) | output: (?, ?, 1164)
dense_1: Dense | input: (?, ?, 1164) | output: (?, ?, 100)
dense_2: Dense | input: (?, ?, 100) | output: (?, ?, 50)
dense_3: Dense | input: (?, ?, 50) | output: (?, ?, 10)
dense_4: Dense | input: (?, ?, 10) | output: (?, ?, 1)

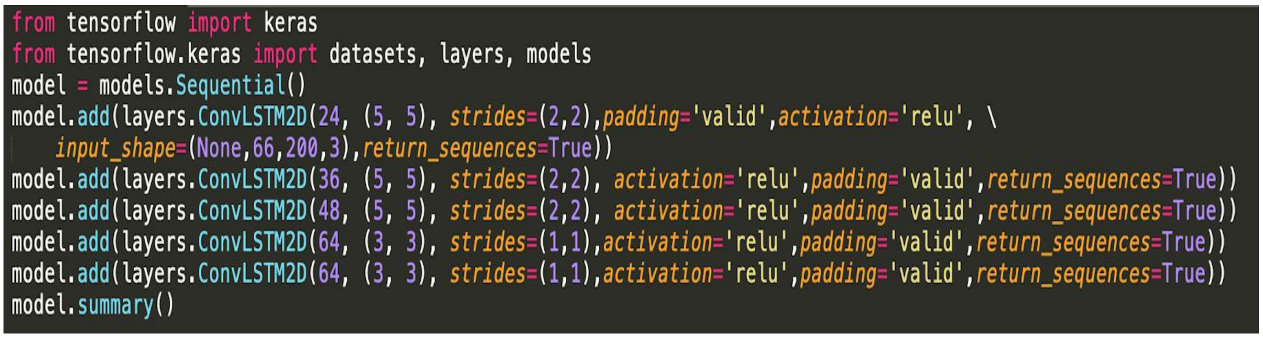
Input Layer

3D Convolution has one extra dimension, hence images are reshaped according to [Batch_Size, Time, Width, Height, Channels], we took 100 images for time dimension i.e. 100 images of 66X200X3 are sent together in the network.

Therefore input tuple is of size **[None, None, 66,200,3]**.

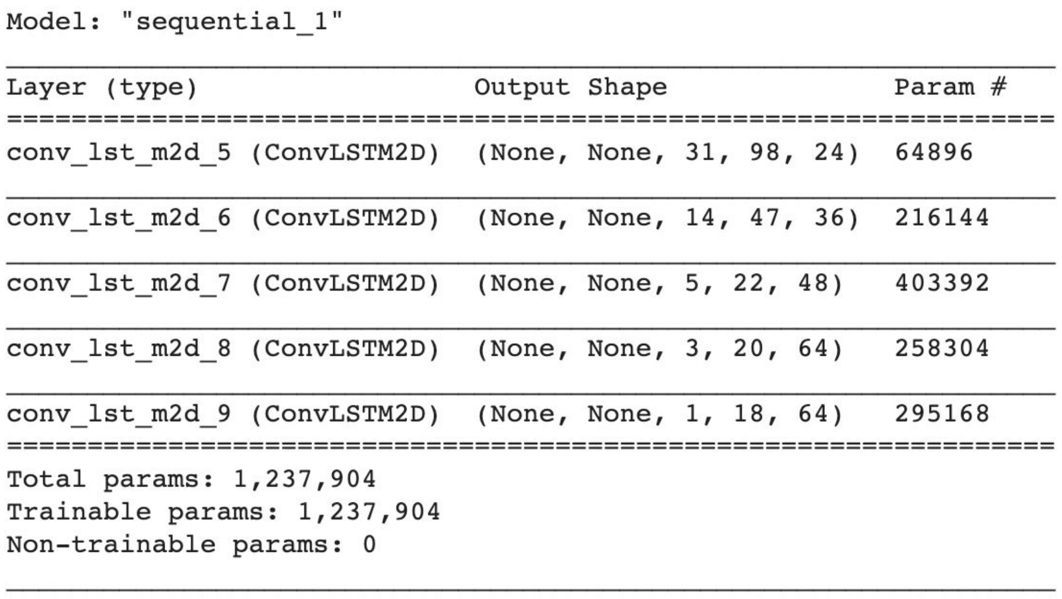
Convolutional- LSTM Layers

The kernel used here are of 2D Convolution is of shape **[W_kernel, H_kernel, in_channels, out_channels],** we have used convolution layers used in phase 2, as internal functions inside LSTM layer.



```
from tensorflow import keras
from tensorflow.keras import datasets, layers, models
model = models.Sequential()
model.add(layers.ConvLSTM2D(24, (5, 5), strides=(2,2),padding='valid',activation='relu', \
    input_shape=(None,66,200,3),return_sequences=True))
model.add(layers.ConvLSTM2D(36, (5, 5), strides=(2,2), activation='relu',padding='valid',return_sequences=True))
model.add(layers.ConvLSTM2D(48, (5, 5), strides=(2,2), activation='relu',padding='valid',return_sequences=True))
model.add(layers.ConvLSTM2D(64, (3, 3), strides=(1,1),activation='relu',padding='valid',return_sequences=True))
model.add(layers.ConvLSTM2D(64, (3, 3), strides=(1,1),activation='relu',padding='valid',return_sequences=True))
model.summary()
```

Figure - Conv2D LSTM Layers

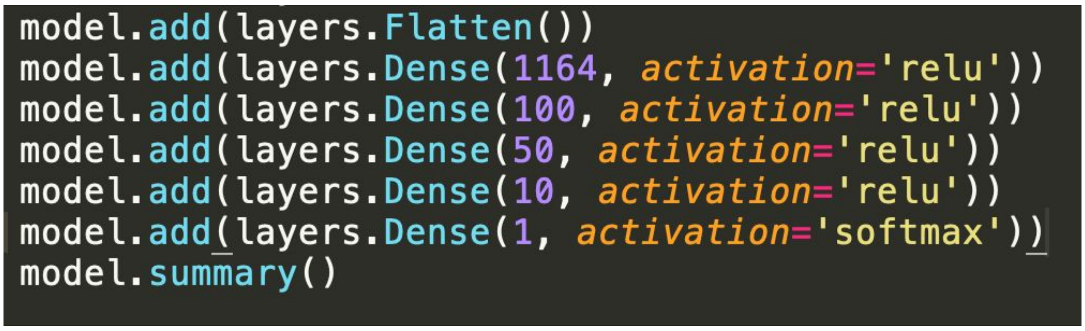


```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv_lst_m2d_5 (ConvLSTM2D)  (None, None, 31, 98, 24)  64896
_________________________________________________________________
conv_lst_m2d_6 (ConvLSTM2D)  (None, None, 14, 47, 36)  216144
_________________________________________________________________
conv_lst_m2d_7 (ConvLSTM2D)  (None, None, 5, 22, 48)   403392
_________________________________________________________________
conv_lst_m2d_8 (ConvLSTM2D)  (None, None, 3, 20, 64)   258304
_________________________________________________________________
conv_lst_m2d_9 (ConvLSTM2D)  (None, None, 1, 18, 64)   295168
=================================================================
Total params: 1,237,904
Trainable params: 1,237,904
Non-trainable params: 0
_________________________________________________________________
```

Figure - Model Summary After Adding ConvLSTM

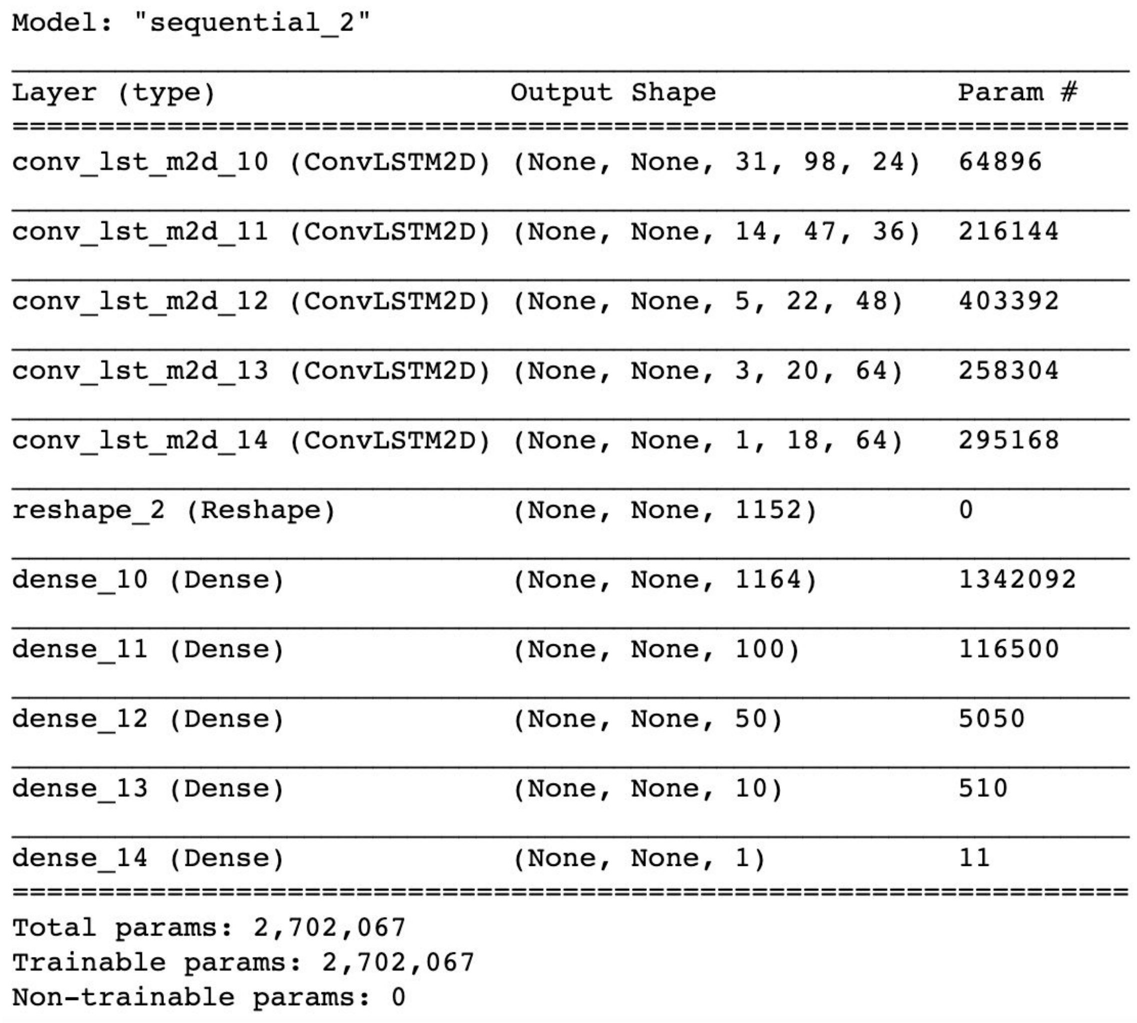
Fully Connected layers

Fully Connected Layer of 1164 neurons is used first. Last convolution layer was of shape [batch_size,100,1,18,64]. Hence (1X18X64)1152 X 1164 trainable parameters exist during flattening. All other FC layer’s structure is shown in the figure.



```
model.add(layers.Flatten())
model.add(layers.Dense(1164, activation='relu'))
model.add(layers.Dense(100, activation='relu'))
model.add(layers.Dense(50, activation='relu'))
model.add(layers.Dense(10, activation='relu'))
model.add(layers.Dense(1, activation='softmax'))
model.summary()
```

Figure - Fully Connected Layers



```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv_lst_m2d_10 (ConvLSTM2D) (None, None, 31, 98, 24)  64896
_________________________________________________________________
conv_lst_m2d_11 (ConvLSTM2D) (None, None, 14, 47, 36)  216144
_________________________________________________________________
conv_lst_m2d_12 (ConvLSTM2D) (None, None, 5, 22, 48)   403392
_________________________________________________________________
conv_lst_m2d_13 (ConvLSTM2D) (None, None, 3, 20, 64)   258304
_________________________________________________________________
conv_lst_m2d_14 (ConvLSTM2D) (None, None, 1, 18, 64)   295168
_________________________________________________________________
reshape_2 (Reshape)          (None, None, 1152)        0
_________________________________________________________________
dense_10 (Dense)             (None, None, 1164)        1342092
_________________________________________________________________
dense_11 (Dense)             (None, None, 100)         116500
_________________________________________________________________
dense_12 (Dense)             (None, None, 50)          5050
_________________________________________________________________
dense_13 (Dense)             (None, None, 10)          510
_________________________________________________________________
dense_14 (Dense)             (None, None, 1)           11
=================================================================
Total params: 2,702,067
Trainable params: 2,702,067
Non-trainable params: 0
```

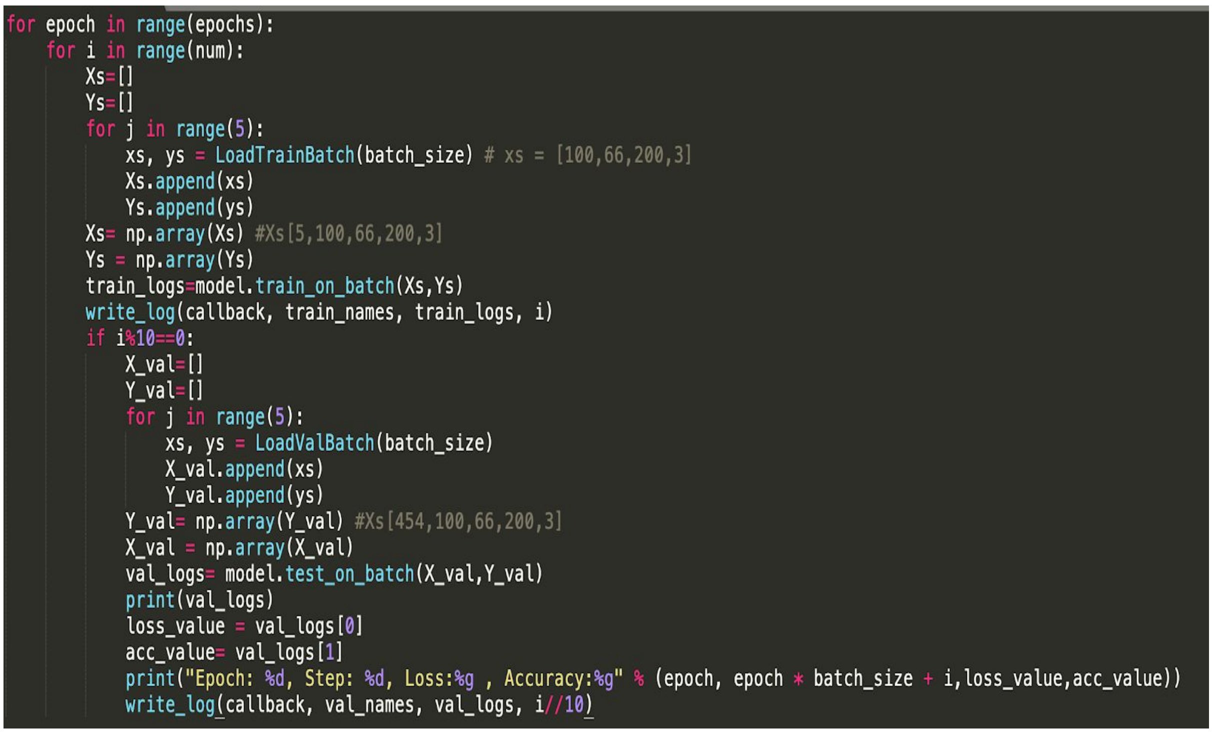
Figure -Complete Model Summary

Training

Machine Used :

- 2.3 GHz Intel Core i5 X 8 core processor, was initially used to train model, which took about 12-16 hours.
- Google Cloud 16 core processor skylake processor, 16 Gb RAM with NVIDIA TESLA T4 GPU, which reduced time to 4 to 4.5 hours of training.

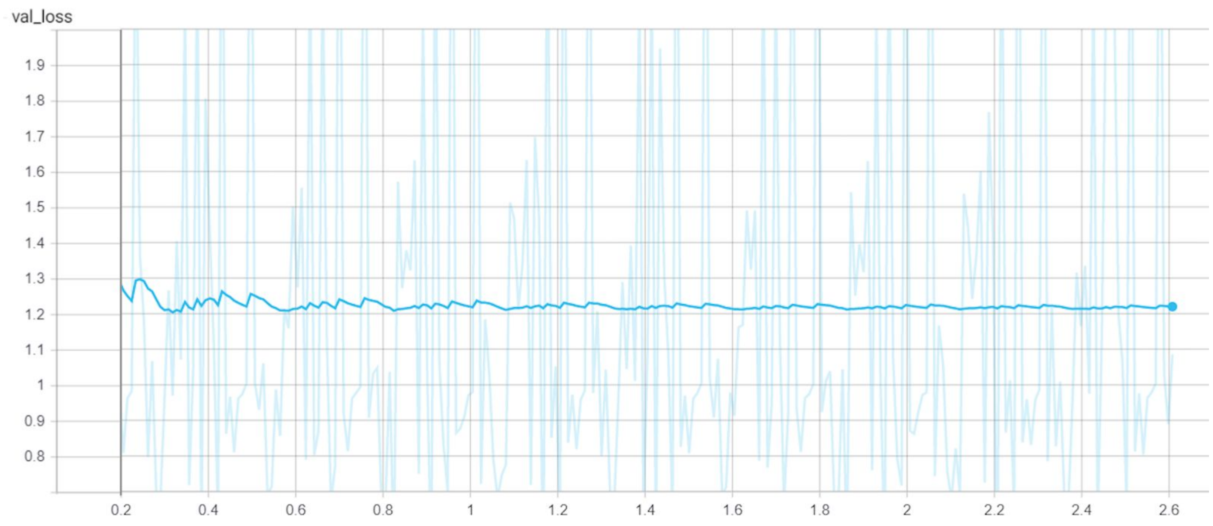
Training code:



```
for epoch in range(epochs):
    for i in range(num):
        Xs=[]
        Ys=[]
        for j in range(5):
            xs, ys = LoadTrainBatch(batch_size) # xs = [100,66,200,3]
            Xs.append(xs)
            Ys.append(ys)
        Xs= np.array(Xs) #Xs[5,100,66,200,3]
        Ys = np.array(Ys)
        train_logs=model.train_on_batch(Xs,Ys)
        write_log(callback, train_names, train_logs, i)
        if i%10==0:
            X_val=[]
            Y_val=[]
            for j in range(5):
                xs, ys = LoadValBatch(batch_size)
                X_val.append(xs)
                Y_val.append(ys)
            Y_val= np.array(Y_val) #Xs[454,100,66,200,3]
            X_val = np.array(X_val)
            val_logs= model.test_on_batch(X_val,Y_val)
            print(val_logs)
            loss_value = val_logs[0]
            acc_value= val_logs[1]
            print("Epoch: %d, Step: %d, Loss:%g , Accuracy:%g" % (epoch, epoch * batch_size + i,loss_value,acc_value))
            write_log(callback, val_names, val_logs, i//10)
```

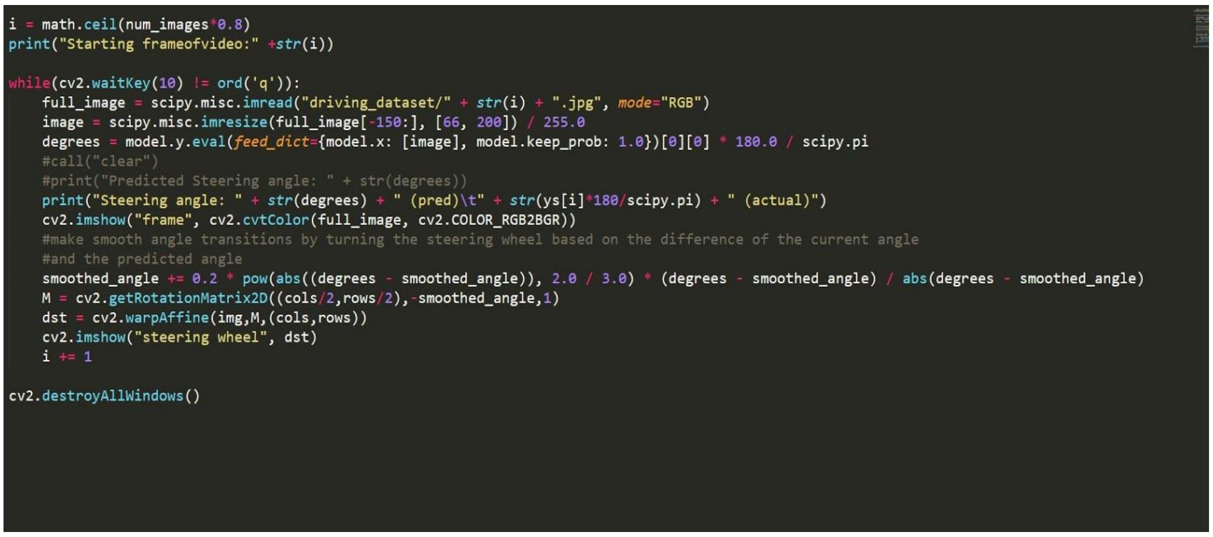
Results

We successfully trained the model to predict the steering angle according to the image of the road given at an instance. The training loss achieved is 1.2 which is more than the training loss achieved in 2D CNN i.e. 0.14.



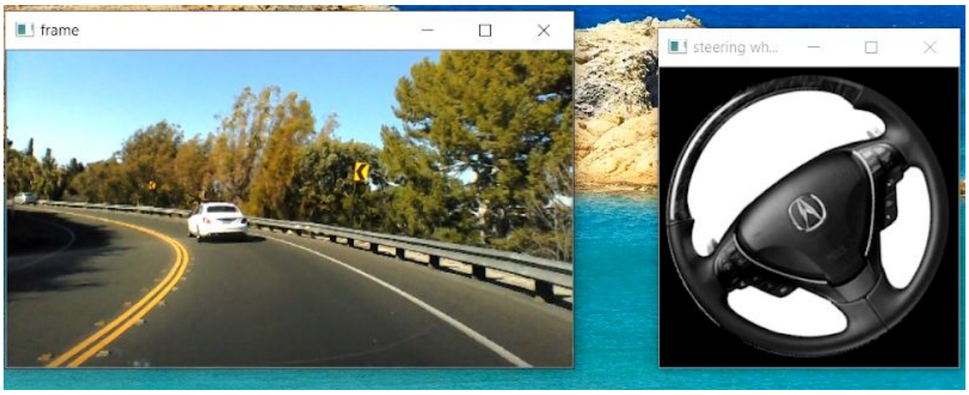
Visualization

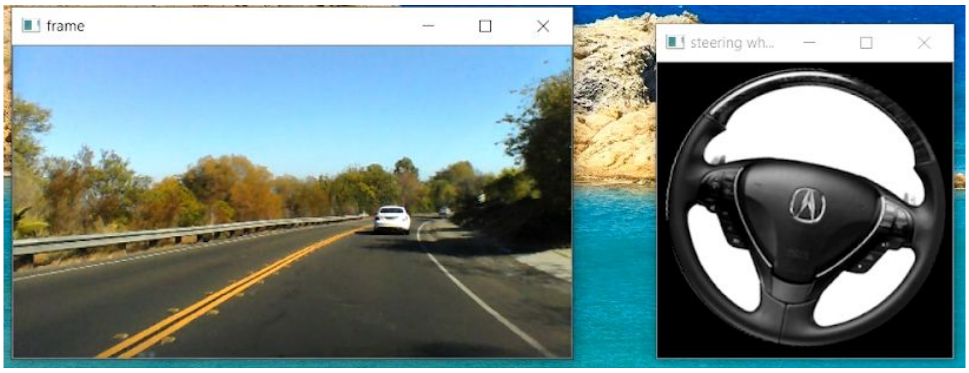
We ran the model on images of the road, and visualized the rotation of the steering wheel according to the input, using the OpenCV library.



```
i = math.ceil(num_images*0.8)
print("Starting frameofvideo:" +str(i))

while(cv2.waitKey(10) != ord('q')):
    full_image = scipy.misc.imread("driving_dataset/" + str(i) + ".jpg", mode="RGB")
    image = scipy.misc.imresize(full_image[-150:], [66, 200]) / 255.0
    degrees = model.y.eval(feed_dict={model.x: [image], model.keep_prob: 1.0})[0][0] * 180.0 / scipy.pi
    #call("clear")
    #print("Predicted Steering angle: " + str(degrees))
    print("Steering angle: " + str(degrees) + " (pred)\t" + str(ys[i]*180/scipy.pi) + " (actual)")
    cv2.imshow("frame", cv2.cvtColor(full_image, cv2.COLOR_RGB2BGR))
    #make smooth angle transitions by turning the steering wheel based on the difference of the current angle
    #and the predicted angle
    smoothed_angle += 0.2 * pow(abs((degrees - smoothed_angle)), 2.0 / 3.0) * (degrees - smoothed_angle) / abs(degrees - smoothed_angle)
    M = cv2.getRotationMatrix2D((cols/2,rows/2),-smoothed_angle,1)
    dst = cv2.warpAffine(img,M,(cols,rows))
    cv2.imshow("steering wheel", dst)
    i += 1

cv2.destroyAllWindows()
```

Figure - Code for Visualization of Automated Steering Wheel





Model comparison

Model	Loss(Mean Squared Error)
Conv2D	0.14
Conv3D	12.07
Conv-LSTM	1.2

Conclusion

We tried three different models to build our automation system and we observed the following points

- The initial Conv2D model with hyperparameter tuning gave the best results among all of the models.
- The second model we tried was Conv3D, this model performed significantly worse than the first and the last model. Also, the training time taken by the 3D-CNNs was significantly larger as compared to the other two models.
- The last model we tried was the Conv-LSTM model which did not perform as good as the first model but still performed much better than the Conv3d model.

Certain inferences can be drawn from our observation. The most important one being that using sequence information among images did not yield a better result so more experimentation can be done in the areas of the end-to-end learning model we started with to achieve even better results.

References

https://images.nvidia.com/content/tegra/automotive/images/2016/solutions/pdf/end-to-end-dl-using-px.pdf
End to End Learning for Self-Driving Cars(NVIDIA CORP.)

http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.
Advances in Neural Information Processing Systems 25, pages
1097–1105. Curran Associates, Inc., 2012

http://www.image-net.org/challenges/LSVRC
Large scale visual recognition challenge (ILSVRC).

http://net-scale.com/doc/net-scale-dave-report.pdf.
Net-Scale Technologies, Inc. Autonomous off-road vehicle control using end-to-end learning, July 2004.

http://repository.cmu.edu/cgi/viewcontent.cgi?article=2874&context=compsci.
ALVINN, an autonomous land vehicle in a neural network. Technical report,
Carnegie Mellon University, 1989.

https://arxiv.org/abs/1608.01230
Learning a Driving Simulator

https://arxiv.org/abs/1711.06976
MIT Advanced Vehicle Technology Study