S.No	Title	Page No.
A	Acknowledgement	2
B	Abstract	3
1	Introduction	4
2	Background Projects	6
3	Problem Statement	91
4	Literature Review	92
5	Requirements	115
6	Methodology and Approach	123
7	Output and Results	150
8	Conclusion	154
9	References	156

ACKNOWLEDGEMENT

We would love to record our deepest gratitude to **Lord Almighty**, for the merciful shower of grace and blessing in all the endeavors to bring out this project successfully.

We would like to express our sincere thanks to our beloved **Vice Chancellor**, **Prof. R. Sethuraman**, for starting this programme and giving us an opportunity to be a part of SASTRA's student community.

We also express our thanks to **Dr. G. Balachandran, Registrar**, SASTRA University, Thanjavur for giving an opportunity to do our B. Tech Degree in ECE.

We sincerely thank our **Dr. V. Ramaswamy, Dean, SRC** for his guidance and encouragement during our study. We also express our thanks to **Dr. A. Alli Rani, Head of the Department, ECE** for her ideas and guidance. We sincerely thank our project co-ordinator **Prof. T. Gayathri Devi, Assistant Professor, Dept of ECE,** who gave us valuable suggestion for the completion of our project.

We place on record our gratitude to **Prof. Fernado Vilarino, Associate Director of CVC , Associate Professor, School of Computing, UAB, Barcelona, Spain** for getting us to this level through his guidance and encouragement throughout this project. We are indebted to him for all that he has helped us with our project.

We faithfully thank all the **teaching and non-teaching staff** of our department who helped us on our efforts during the course of our project. We express deep sense of gratitude to our beloved parents and friends who inspired and encouraged us all along the successful completion of our project work.

We would also like to thank and express our gratitude to our **parents** who were by our side all time helped us to pursue and finish off the project successfully.

ABSTRACT

Unfathomable as it may seem, the world is definitely heading to an age of Self driving cars that can replace human drivers. Imagine a world with no car crashes. Our self-driving vehicles aim to eliminate human driver error — the primary cause of 94 percent of crashes — leading to fewer injuries and fatalities. Imagine widespread use of electric-only vehicles, reducing vehicle emissions. Our self-driving vehicles will all be electric, contributing to a better environment. Imagine not sitting in traffic for what feels like half your life. And imagine a crowded city not filled with congested roads and parking lots and structures but with efficiently moving traffic and more space. Our self-driving vehicles will improve access to mobility for those who currently cannot drive due to age, disability, or otherwise. In this project, we are going to design a system that drives a car autonomously in a simulated environment using keras deep learning library. As an input, we used Udacity Self Driving Car Simulator [1] to generate input data. We use keyboard to drive around the circuit. The simulator records screen shots of the images and the decisions we made: steering angles, throttle and brake, while driving the car in simulator. The mission is to create a deep neural network to emulate the behavior of the human being, and the put this network to run the simulator autonomously.

INTRODUCTION

It is important to understand that a lot of planning has to go into designing a system that drives a car autonomously in a simulated environment. Building a complete neural network model, before implementing it with the simulator is very important in a project that involves precise integration between our model and the simulator. An equally important aspect is gaining as much background knowledge of neural networks and keras.

In order to get a good overview about keras and neural networks, my professor took an intense care on me and gave an Introductory session to me with his student, which makes me to get familiar with keras and neural networks. In order to have good understanding on keras and the concepts of neural networks, my professor encouraged me to do some basic projects. By doing multiple basic projects, it helped me gain an overall picture of what goes into designing a system that drives a car autonomously in a simulated environment from scratch.

This part of my project involved designing a system that drives a car autonomously in a simulated environment. In our model, the architecture has been inspired from paper, End to End Learning for Self- Driving Cars published [2] by NVIDIA [3]. In order to design our model to make it suitable for all driving conditions, we are going to use data augmentation technique to train our model, which produces the steering angle as output.

In order to generate input data, we used Udacity self driving car simulator. We used keyboard to drive around the circuit. The simulator records screen shots of the images and the decisions we made: steering angles, throttle and brake, while driving the car in simulation

To design our model, we used keras deep learning library to build our neural network and used OpenCV for data augmentation in python programming language.

Once the model is trained, we used flask, a web application frame work to deploy our model in Udacity self driving car simulator and to drive our car autonomously.

Background Projects

In order to understand the basics of Neural networks, our professor asked us to implement some basic projects. We implemented multiple basic projects. The complexity of the projects have been increased gradually from project to project.

Diabetes Prediction Model

We are going to predict a model, which can predict whether the person will get affected by diabetes or not within 5 years.

Load Data

Whenever we work with machine learning algorithms that use a stochastic process (e.g. random numbers), it is a good idea to set the random number seed. This is so that you can run the same code again and again and get the same result. This is useful if you need to demonstrate a result, compare algorithms using the same source of randomness or to debug a part of your code. You can initialize the random number generator with any seed you like, for example:

```
from keras.models import Sequential
from keras.layers import Dense
import numpy
# fix random seed for reproducibility
numpy.random.seed(7)
```

Now we can load our data.

In this project, we are going to use the Pima Indians onset of diabetes dataset. This is a standard machine learning dataset from the UCI Machine Learning repository. It describes patient medical record data for Pima Indians and whether they had an onset of diabetes within five years.

As such, it is a binary classification problem (onset of diabetes as 1 or not as 0). All of the input variables that describe each patient are numerical. This makes it easy to use directly with neural networks that expect numerical input and output values, and ideal for our first neural network in Keras.

Download the dataset and place it in your local working directory, the same as your python file. Save it with the file name:

```
pima-indians-diabetes.csv
```

You can now load the file directly using the NumPy function **loadtxt()**. There

are eight input variables and one output variable (the last column). Once loaded we can split the dataset into input variables (X) and the output class variable (Y).

```
# load pima indians dataset
dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset[:,0:8]
Y = dataset[:,8]
```

We have initialized our random number generator to ensure our results are reproducible and loaded our data. We are now ready to define our neural network model.

Note, the dataset has 9 columns and the range 0:8 will select columns from 0 to 7, stopping before index 8.

Define Model

Models in Keras are defined as a sequence of layers.

We create a Sequential model and add layers one at a time until we are happy with our network topology.

The first thing to get right is to ensure the input layer has the right number of inputs. This can be specified when creating the first layer with the **input_dim** argument and setting it to 8 for the 8 input variables. There are heuristics that we can use and often the best network structure is found through a process of trial and error experimentation. Generally, you need a network large enough to capture the structure of the problem if that helps at all.

In this example, we will use a fully-connected network structure with three layers.

Fully connected layers are defined using the Dense class. We can specify the number of neurons in the layer as the first argument, the initialization method as the second argument as **init** and specify the activation function using the **activation** argument.

In this case, we initialize the network weights to a small random number generated from a uniform distribution (‘**uniform**‘), in this case between 0 and 0.05 because that is the default uniform weight initialization in Keras. Another traditional alternative would be ‘**normal’** for small random numbers

generated from a Gaussian distribution.
We will use the rectifier (‘**relu**‘) activation function on the first two layers and the sigmoid function in the output layer. It used to be the case that sigmoid and tanh activation functions were preferred for all layers. These days, better performance is achieved using the rectifier activation function. We use a sigmoid on the output layer to ensure our network output is between 0 and 1 and easy to map to either a probability of class 1 or snap to a hard classification of either class with a default threshold of 0.5.
We can piece it all together by adding each layer. The first layer has 12 neurons and expects 8 input variables. The second hidden layer has 8 neurons and finally, the output layer has 1 neuron to predict the class (onset of diabetes or not).

```
# create model
model = Sequential()
model.add(Dense(12, input_dim=8, activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

Compile Model

Now that the model is defined, we can compile it.

Compiling the model uses the efficient numerical libraries under the covers (the so-called backend) such as Theano or TensorFlow. The backend automatically chooses the best way to represent the network for training and making predictions to run on your hardware, such as CPU or GPU or even distributed.

When compiling, we must specify some additional properties required when training the network. Remember training a network means finding the best set of weights to make predictions for this problem.

We must specify the loss function to use to evaluate a set of weights, the optimizer used to search through different weights for the network and any optional metrics we would like to collect and report during training.

In this case, we will use logarithmic loss, which for a binary classification problem is defined in Keras as “**binary_crossentropy**“. We will also use the

efficient gradient descent algorithm "**adam**" for no other reason that it is an efficient default. Learn more about the Adam optimization algorithm in the paper "Adam: A Method for Stochastic Optimization".
Finally, because it is a classification problem, we will collect and report the classification accuracy as the metric.

```
# Compile model

model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])
```

Fit Model

We have defined our model and compiled it ready for efficient computation. Now it is time to execute the model on some data. We can train or fit our model on our loaded data by calling the **fit()** function on the model. The training process will run for a fixed number of iterations through the dataset called epochs, that we must specify using the **nepochs** argument. We can also set the number of instances that are evaluated before a weight update in the network is performed, called the batch size and set using the **batch_size** argument.

For this problem, we will run for a small number of iterations (150) and use a relatively small batch size of 10. Again, these can be chosen experimentally by trial and error.

```
# Fit the model

model.fit(X, Y, epochs=150, batch_size=10)
```

This is where the work happens on your CPU or GPU. No GPU is required for this project.

Evaluate Model

We have trained our neural network on the entire dataset and we can evaluate the performance of the network on the same dataset.

This will only give us an idea of how well we have modeled the dataset (e.g. train accuracy), but no idea of how well the algorithm might perform on new data. We have done this for simplicity, but ideally, you could separate your data into train and test datasets for training and evaluation of your model.

You can evaluate your model on your training dataset using the **evaluate()** function on your model and pass it the same input and output used to train the model.

This will generate a prediction for each input and output pair and collect scores, including the average loss and any metrics you have configured, such as accuracy.

```
# evaluate the model

scores = model.evaluate(X, Y)

print("\n%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))
```

Our accuracy model accuracy is 78.26%

…

```
Epoch 145/150

768/768 [==============================] - 0s - loss: 0.5105 - acc: 0.7396

Epoch 146/150

768/768 [==============================] - 0s - loss: 0.4900 - acc: 0.7591
```

Epoch 147/150

768/768 [==============================] - 0s - loss: 0.4939 - acc: 0.7565

Epoch 148/150

768/768 [==============================] - 0s - loss: 0.4766 - acc: 0.7773

Epoch 149/150

768/768 [==============================] - 0s - loss: 0.4883 - acc: 0.7591

Epoch 150/150

768/768 [==============================] - 0s - loss: 0.4827 - acc: 0.7656

32/768 [>.....] - ETA: 0s

acc: 78.26%

Predictions

We can adapt the above example and use it to generate predictions on the training dataset, pretending it is a new dataset we have not seen before.

Making predictions is as easy as calling **model.predict()**. We are using a sigmoid activation function on the output layer, so the predictions will be in the range between 0 and 1. We can easily convert them into a crisp binary prediction for this classification task by rounding them.

The complete code that makes predictions for each record in the training data is listed below.

```
# Create first network with Keras

from keras.models import Sequential

from keras.layers import Dense
```

```
import numpy

# fix random seed for reproducibility

seed = 7

numpy.random.seed(seed)

# load pima indians dataset

dataset = numpy.loadtxt("pima-indians-diabetes.csv", delimiter=",")

# split into input (X) and output (Y) variables

X = dataset[:,0:8]

Y = dataset[:,8]

# create model

model = Sequential()

model.add(Dense(12, input_dim=8, init='uniform', activation='relu'))

model.add(Dense(8, init='uniform', activation='relu'))

model.add(Dense(1, init='uniform', activation='sigmoid'))

# Compile model

model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])

# Fit the model

model.fit(X, Y, epochs=150, batch_size=10,  verbose=2)

# calculate predictions
```

predictions = model.predict(X)

 # round predictions

rounded = [round(x[0]) for x in predictions]

print(rounded)

Running this modified code now prints the predictions for each input pattern. We could use these predictions directly in our application if needed.

```
[1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0,
1.0, 0.0, 1.0, 0.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0,
1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0,
0.0, 1.0, 0.0, 1.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0,
1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0,
1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 1.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 1.0,
0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 0.0, 0.0, 1.0, 0.0, 1.0, 1.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0, 0.0, 1.0,
0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0, 0.0, 1.0, 1.0, 1.0, 1.0, 0.0,
0.0, 0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 1.0,
1.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
1.0, 1.0, 0.0, 1.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0, 0.0,
1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 1.0,
0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 1.0, 1.0, 1.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 0.0, 1.0,
1.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 1.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0,
0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0,
0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 0.0, 0.0,
1.0, 0.0, 0.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
1.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0,
```

0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 1.0, 1.0, 0.0, 1.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 1.0, 0.0, 1.0, 0.0, 1.0, 1.0, 1.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0]

CAT –DOG CLASSIFIER

In this project, we are going to design a model which can predict whether the image is cat or dog (cat dog classifier).

from blog.keras.io. It uses data that can be downloaded at: https://www.kaggle.com/c/dogs-vs-cats/data. In our setup, we: - created a data/ folder - created train/ and validation/ subfolders inside data/ - created cats/ and dogs/ subfolders inside train/ and validation/ - put the cat pictures index 0-999 in data/train/cats - put the cat pictures index 1000-1400 in data/validation/cats- put the dogs pictures index 11500-12499 in data/train/dogs - put the dog pictures index 13500-13900 in data/validation/dogs So that we have 1000 training examples for each class, and 401 validation examples for each class.

Import required models and layers

First we need to import required models, layers from keras. As we are dealing with image data , we need to import ImageDataGenerator as it Generate batches of tensor image data with real-time data augmentation. The data will be looped over (in batches).

```
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D
from keras.layers import Activation, Dropout, Flatten, Dense
from keras import backend as K
```

Initialize the model parameters

In keras, we need to give the shape of the input data to first layer of the network.

```
img_width, img_height = 150, 150
```

Initialise the batch size, number the epoches you want to perform and input directories of train and validation data.

```
train_data_dir = 'data/train'
validation_data_dir = 'data/validation'
nb_train_samples = 2000
```

```
nb_validation_samples = 800
epochs = 5
batch_size = 16
if K.image_data_format() == 'channels_first':
input_shape = (3, img_width, img_height)
else:
input_shape = (img_width, img_height, 3)
```

Define Model

we are using the sequential model , which is the stack of multiple layers. construction of model with required layers. In conv2D 32 is the number of hidden layers and (3, 3) is the kernal size. we are relu activation maxpolling2D is used to reduce the features of the data i.e only high influenctial features are extracted and pool_size is the required pool scale i.e pool_size = (horizontal,vertical). Dropout is a regularization technique, which aims to reduce the complexity of the model with the goal to prevent overfitting. Dense is a fully connected layer, in which all the neurons in this layer is connected to next layer. Flatten layer is used to flatten the input and it does not affect the batch size. This model is the 12 layer network.

```
model = Sequential()
model.add(Conv2D(32, (3, 3), input_shape=input_shape))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(32, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
 model.add(Flatten())
model.add(Dense(64))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(1))
model.add(Activation('sigmoid'))
```

Compile the Model

Now we are going to compile the model, while compiling the model we need to define the three parameters which are As we are doing binary classification, we are using BI-NARY_CROSSENTROPY. Here we are using rmsprop. As we are dealing with classification problem , we are using accuracy metric.

```
model.compile(loss='binary_crossentropy',optimizer='rmsprop',metrics=['accuracy'])
```

Data Augmentation

We are using ImagedataGenerator as it generate batches of tensor image data with real-time data augmentation. The data will be looped over (in batches). rescale is the value that is used to multiply the give data.

```
train_datagen = ImageDataGenerator(rescale=1. / 255,shear_range=0.2,
zoom_range=0.2,horizontal_flip=True)
test_datagen = ImageDataGenerator(rescale=1. / 255)
train_generator =
train_datagen.flow_from_directory(train_data_dir,target_size
=(img_width, img_height),
batch_size=batch_size,class_mode='binary')
```

Found 2000 images belonging to 2 classes.

```
validation_generator = test_datagen.flow_from_directory(
validation_data_dir,
target_size=(img_width, img_height),
batch_size=batch_size,
class_mode='binary')
```

Found 802 images belonging to 2 classes.

Fit Model

for training the model , FIT function is typically used.

```
model.fit_generator(train_generator,steps_per_epoch=nb_train_samples
// batch_size,epochs=epochs,validation_data=validation_generator,
```

```
validation_steps=nb_validation_samples // batch_size)
Epoch 1/5
125/125 [==============================] - 57s 454ms/step - loss: 0.7213 - acc: 0.5610 Epoch 2/5
125/125 [==============================] - 52s 419ms/step - loss: 0.6611 - acc: 0.6250 Epoch 3/5
125/125 [==============================] - 53s 420ms/step - loss: 0.6496 - acc: 0.6480 Epoch 4/5
125/125 [==============================] - 52s 420ms/step - loss: 0.6299 - acc: 0.6690  Epoch 5/5
125/125 [==============================] - 53s 420ms/step - loss: 0.5968 - acc: 0.6965  Out[13]:
<keras.callbacks.History at 0x7fae30c67f60>
```

If we perform 15 epochs, we will get accuracy around 85.0 %.

PORTO SEGURO DRIVER INSURANCE PREDICTION

ABSTRACT

Nothing ruins the thrill of buying a brand new car more quickly than seeing your new insurance bill. The sting's even more painful when you know you're a good driver. It doesn't seem fair that you have to pay so much if you've been cautious on the road for years. Inaccuracies in car insurance company's claim predictions raise the cost of insurance for good drivers and reduce the price for bad ones. In this project, we are going to build a model that predicts the probability that a driver will initiate an auto insurance claim in the next year using the data, which has information of auto insurance policy holder. We are going to build a model using neural networks in keras having tensor flow as a backend. As, we are using textual data, we are going to use word embedding concept.

A more accurate prediction will allow the insurance company to further tailor their prices, and hopefully make auto insurance coverage more accessible to more drivers. So, the probability of increasing the profits for insurance company is much higher by using the predictions from this model. Now, we are now are going to build a prediction model, which predicts the probability that an auto insurance policy holder files a claim

INTRODUCTION

The heart of any model is neural network, so it is very important to decide which concept you are using while building the layers of neural network. As we are dealing with textual data, we are going to use embedding concept for our neural network.

Word embeddings provide a dense representation of words and their relative meanings. They are an improvement over sparse representations used in simpler bag of word model representations. Word embeddings can be learned from text data and reused among projects. They can also be learned as part of fitting a neural network on text data.

Training data plays a vital role in the performance and efficiency of the model. So, we should use correct data to train our model. In our model we are going use porto seguro's iauto insurance customers data to train our model.

In the training data[1], features that belong to similar groupings are tagged as such in the feature names (e.g., ind, reg, car, calc). In addition, feature names include the postfix bin to indicate binary features and cat to indicate categorical features. Features without these designations are either continuous or ordinal. Values of -1 indicate that the feature was missing from the observation. The target column's signifies whether or not a claim was filed for that policy holder.

[1] Data - https://www.kaggle.com/c/porto-seguro-safe-driver-prediction/data

PROBLEM STATEMENT

In this project, we are going to build a model for kaggle competition[2]. We are going to build a prediction model for Porto seguro, one of Brazil's largest auto and homeowner insurance companies. Inaccuracies in car insurance company's claim predictions raise the cost of insurance for good drivers and reduce the price for bad ones.

In this project, we are going to build a model that predicts the probability that a driver will initiate an auto insurance claim in the next year. While Porto Seguro has used machine learning for the past 20 years, they're looking to Kaggle's machine learning community to explore new, more powerful methods. A more accurate prediction will allow them to further tailor their prices, and hopefully make auto insurance coverage more accessible to more drivers.

Building from bottom up, there are a number of things to consider and get right.

1) In the beginning, it is very important to look at the data to make it suitable for training the neural network, which is called data cleaning
2) we need to choose the neural network which works best for our problem
3) we need to create the model
4) Fit the model
5) Evaluate the model
6) Make predictions

[2] Kaggle - https://www.kaggle.com/c/porto-seguro-safe-driver-prediction

Software Requirements

Anaconda Cloud

Anaconda is an open-source package manager, environment manager, and distribution of the Python and Rvii programming languages. It is commonly used for large scale data processing, scientific computing, and predictive analytics, serving data scientists, developers, business analysts, and those working in DevOps. Anaconda offers a collection of over 720 open-source packages, and is available in both free and paid versions. The Anaconda distribution ships with the conda command-line utility. You can learn more about Anaconda cloud by reading the Anaconda Documentation pages.

Installing Anaconda cloud

The best way to install Anaconda is to download the latest Anaconda installer bash script, verify it, and then run it and you can see this page for installation steps[3].

a. Python

Python is a widely used high-level, general-purpose, interpreted, dynamic programming language. Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than would be possible in languages such as C++ or Java. The language provides constructs intended to enable clear programs on both a small and large scale.

To install python using command line

$ conda install -c anaconda python

b. Pandas

pandas is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language.

To install pandas using command line

$ conda install -c conda-forge pandas

c. Jupyter

The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations

[3] https://www.pugetsystems.com/labs/hpc/How-to-Install-Anaconda-Python-and-First-Steps-for-Linux-and-Windows-917/

and narrative text. Uses include: data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more.

You can find jupyter by launching the anaconda navigator.

d. Numpy

NumPy is the fundamental package for scientific computing with Python. It contains among other things:

• a powerful N-dimensional array object
• sophisticated (broadcasting) functions
• tools for integrating C/C++ and Fortran code

To install Numpy using command line

$ conda install -c anaconda numpy

f. Scikit - learn

Scikit - Learn It is used for machine learning in python. It is simple and efficient tools for data mining and data analysis. It is accessible to everybody, and reusable in various contexts and it can be built on NumPy, SciPy, and matplotlib

To install scikit – learn using command line

$ conda install -c anaconda scikit-learn

g. Tensor flow

Tensor Flow is an open source software library for high performance numerical computation. Its flexible architecture allows easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices. Originally developed by researchers and engineers from the Google Brain team within Google's AI organization, it comes with strong support for machine learning and deep learning and the flexible numerical computation core is used across many other scientific domains. We are not using tensor flow directly. But, we are using it as a backend for keras

To install tensor flow using command line

$ conda install -c anaconda tensorflowgpu

h. Keras

Keras is a high-level neural networks API, written in Python and capable of running on top of Tensoe Flow, CNTN or Theano. It was developed with a focus on enabling fast experimentation. Being able to go from idea to result

with the least possible delay is key to doing good research. Keras allows us for easy and fast prototyping (through user
friendliness, modularity, and extensibility). Supports both convolutional networks and recurrent networks, as well as combinations of the two.
To Runs seamlessly on CPU and GPU.
To install keras using command line
$ conda install -c anaconda keras-gpu

CONCEPTUAL MODEL

Word Embedding

A word embedding is a class of approaches for representing words and documents using a dense vector representation.

It is an improvement over more the traditional bag-of-word model encoding schemes where large sparse vectors were used to represent each word or to score each word within a vector to represent an entire vocabulary. These representations were sparse because the vocabularies were vast and a given word or document would be represented by a large vector comprised mostly of zero values.

Instead, in an embedding, words are represented by dense vectors where a vector represents the projection of the word into a continuous vector space.

The position of a word within the vector space is learned from text and is based on the words that surround the word when it is used. The position of a word in the learned vector space is referred to as its embedding. Two popular examples of methods of learning word embeddings from text include: Word2Vec6, GloVe7.

In addition to these carefully designed methods, a word embedding can be learned as part of a deep learning model. This can be a slower approach, but tailors the model to a specific training dataset.

Keras offers an Embedding layer that can be used for neural networks on text data.

It requires that the input data be integer encoded, so that each word is represented by a unique integer. This data preparation step can be performed using the Tokenizer API also provided with Keras. The Embedding layer is initialized with random weights and will learn an embedding for all of the words in the training dataset. It is a flexible layer that can be used in a variety of ways, such as:

It can be used alone to learn a word embedding that can be saved and used in another model later. It can be used as part of a deep learning model where the embedding is learned along with the model itself. It can be used to load a pre-trained word embedding model, a type of transfer learning. The Embedding layer is defined as the first hidden layer of a network. It must specify 3 arguments:

It must specify 3 arguments:

input_dim: This is the size of the vocabulary in the text data. For example, if your data is integer encoded to values between 0-10, then the size of the vocabulary would be 11 words.

output_dim: This is the size of the vector space in which words will be embedded. It defines the size of the output vectors from this layer for each word. For example, it could be 32 or 100 or even larger. Test different values for your problem.

input_length: This is the length of input sequences, as you would define for any input layer of a Keras model. For example, if all of your input documents are comprised of 1000 words, this would be 1000.

For example, below we define an Embedding layer with a vocabulary of 200 (e.g. integer encoded words from 0 to 199, inclusive), a vector space of 32 dimensions in which words will be embedded, and input documents that have 50 words each

e = Embedding(200, 32, input_length=50)

The Embedding layer has weights that are learned. If you save your model to file, this will include weights for the Embedding layer.

The output of the Embedding layer is a 2D vector with one embedding for each word in the input sequence of words (input document). If you wish to connect a Dense layer directly to an Embedding layer, you must first flatten the 2D output matrix to a 1D vector using the Flatten layer

METHODOLOGY AND APPROACH

Outlined below is the methodology adopted to achieve the end result. The problem we are going to look at in this project is the prediction problem for an insurance company.

In this project we are going to build a model, which will predict the probability that an auto insurance policy holder files a claim. We are going to use the porto seguro's auto insurance policy holders data to train our model. You can download the data from the kaggle website.

Import required functions and classes

Before we get started, let's first import all of the functions and classes we intend to use. Before we do anything, it is a good idea to fix the random number seed to ensure our results are reproducible.

```
import numpy as np
import pandas as pd
np.random.seed(10)
from tensorflow import set_random_seed
set_random_seed(15)
from keras.models import Sequential
from keras.layers import Dense, Activation, Merge, Reshape, Dropout
from keras.layers.embeddings import Embedding
from sklearn.model_selection import StratifiedKFold
```

Data Loading and preprocessing

Before using the data for training the model, it is very important to clean the data and to get rid of unwanted columns to make the data more suitable for training the model.

```
df_train = pd.read_csv('input/portotrain.csv')
df_test = pd.read_csv('input/portotest.csv')
X_train, y_train = df_train.iloc[:,2:], df_train.target
X_test = df_test.iloc[:,1:]
cols_use = [c for c in X_train.columns if (not c.startswith('ps_calc_'))]
X_train = X_train[cols_use]
X_test = X_test[cols_use]
col_vals_dict = {c: list(X_train[c].unique()) for c in X_train.columns if
```

```
c.endswith('_cat')}
embed_cols = []
for c in col_vals_dict:
if len(col_vals_dict[c])>2:
embed_cols.append(c)
print(c + ': %d values' % len(col_vals_dict[c])) #look at value counts to know the print('\n')
```

Building The Model

While building the neural networks in the model, It is very important to choose the best layers for our neural network which will give best results for our model. In this project, we are going to use embedding concept.

```
def build_embedding_network():
models = []
model_ps_ind_02_cat = Sequential()
model_ps_ind_02_cat.add(Embedding(5, 3, input_length=1))
model_ps_ind_02_cat.add(Reshape(target_shape=(3,)))
models.append(model_ps_ind_02_cat)
model_ps_ind_04_cat = Sequential()
model_ps_ind_04_cat.add(Embedding(3, 2, input_length=1))
model_ps_ind_04_cat.add(Reshape(target_shape=(2,)))
models.append(model_ps_ind_04_cat)
model_ps_ind_05_cat = Sequential()
model_ps_ind_05_cat.add(Embedding(8, 5, input_length=1))
model_ps_ind_05_cat.add(Reshape(target_shape=(5,)))
models.append(model_ps_ind_05_cat)
model_ps_car_01_cat = Sequential()
model_ps_car_01_cat.add(Embedding(13, 7, input_length=1))
model_ps_car_01_cat.add(Reshape(target_shape=(7,)))
models.append(model_ps_car_01_cat)
model_ps_car_02_cat = Sequential()
model_ps_car_02_cat.add(Embedding(3, 2, input_length=1))
model_ps_car_02_cat.add(Reshape(target_shape=(2,)))
```

```
models.append(model_ps_car_02_cat)
model_ps_car_03_cat = Sequential()
model_ps_car_03_cat.add(Embedding(3, 2, input_length=1))
model_ps_car_03_cat.add(Reshape(target_shape=(2,)))
models.append(model_ps_car_03_cat)
model_ps_car_04_cat = Sequential()
model_ps_car_04_cat.add(Embedding(10, 5, input_length=1))
model_ps_car_04_cat.add(Reshape(target_shape=(5,)))
models.append(model_ps_car_04_cat)
model_ps_car_05_cat = Sequential()
model_ps_car_05_cat.add(Embedding(3, 2, input_length=1))
model_ps_car_05_cat.add(Reshape(target_shape=(2,)))
models.append(model_ps_car_05_cat)
model_ps_car_06_cat = Sequential()
model_ps_car_06_cat.add(Embedding(18, 8, input_length=1))
model_ps_car_06_cat.add(Reshape(target_shape=(8,)))
models.append(model_ps_car_06_cat)
model_ps_car_07_cat = Sequential()
model_ps_car_07_cat.add(Embedding(3, 2, input_length=1))
model_ps_car_07_cat.add(Reshape(target_shape=(2,)))
models.append(model_ps_car_07_cat)
model_ps_car_09_cat = Sequential()
model_ps_car_09_cat.add(Embedding(6, 3, input_length=1))
model_ps_car_09_cat.add(Reshape(target_shape=(3,)))
models.append(model_ps_car_09_cat)
model_ps_car_10_cat = Sequential()
model_ps_car_10_cat.add(Embedding(3, 2, input_length=1))
model_ps_car_10_cat.add(Reshape(target_shape=(2,)))
models.append(model_ps_car_10_cat)
model_ps_car_11_cat = Sequential()
model_ps_car_11_cat.add(Embedding(104, 10, input_length=1))
model_ps_car_11_cat.add(Reshape(target_shape=(10,)))
models.append(model_ps_car_11_cat)
```

```
model_rest = Sequential()
model_rest.add(Dense(16, input_dim=24))
models.append(model_rest)
model = Sequential()
model.add(Merge(models, mode='concat'))
model.add(Dense(80))
model.add(Activation('relu'))
model.add(Dropout(.35))
model.add(Dense(20))
model.add(Activation('relu'))
model.add(Dropout(.15))
model.add(Dense(10))
model.add(Activation('relu'))
model.add(Dropout(.15))
model.add(Dense(1))
model.add(Activation('sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='adam')
return model
```

Data Conversion

We are going to convert the data to list format to match the network structure.

```
def preproc(X_train, X_val, X_test):
input_list_train = []
input_list_val = []
input_list_test = []
#the cols to be embedded: rescaling to range [0, # values)
for c in embed_cols:
raw_vals = np.unique(X_train[c])
val_map = {}
for i in range(len(raw_vals)):
val_map[raw_vals[i]] = i
```

```
input_list_train.append(X_train[c].map(val_map).values)
input_list_val.append(X_val[c].map(val_map).fillna(0).values)
input_list_test.append(X_test[c].map(val_map).fillna(0).values)
#the rest of the columns
other_cols = [c for c in X_train.columns if (not c in embed_cols)]
input_list_train.append(X_train[other_cols].values)
input_list_val.append(X_val[other_cols].values)
input_list_test.append(X_test[other_cols].values)
return input_list_train, input_list_val, input_list_test
```

Scoring Function

We are going to create a gini scoring function. Below is the code

```
def ginic(actual, pred):
n = len(actual)
a_s = actual[np.argsort(pred)]
a_c = a_s.cumsum()
giniSum = a_c.sum() / a_c[-1] - (n + 1) / 2.0
return giniSum / n
def gini_normalizedc(a, p):
return ginic(a, p) / ginic(a, a)
```

Network Training

Now, we are going to train the model with our training data

```
K = 8
runs_per_fold = 3
n_epochs = 15
cv_ginis = []
full_val_preds = np.zeros(np.shape(X_train)[0])
y_preds = np.zeros((np.shape(X_test)[0],K))
kfold = StratifiedKFold(n_splits = K, random_state = 231, shuffle = True)
for i, (f_ind, outf_ind) in enumerate(kfold.split(X_train, y_train)):
X_train_f, X_val_f = X_train.loc[f_ind].copy(), X_train.loc[outf_ind].copy()
y_train_f, y_val_f = y_train[f_ind], y_train[outf_ind]
```

```
X_test_f = X_test.copy()
#upsampling adapted from kernel:
pos = (pd.Series(y_train_f == 1))
# Add positive examples
X_train_f = pd.concat([X_train_f, X_train_f.loc[pos]], axis=0)
y_train_f = pd.concat([y_train_f, y_train_f.loc[pos]], axis=0)
# Shuffle data
idx = np.arange(len(X_train_f))
np.random.shuffle(idx)
X_train_f = X_train_f.iloc[idx]
y_train_f = y_train_f.iloc[idx]
#preprocessing
proc_X_train_f, proc_X_val_f, proc_X_test_f = preproc(X_train_f, X_val_f, X_test_f)
```

Tracking the prediction for cv scores

```
val_preds = 0
for j in range(runs_per_fold):
NN = build_embedding_network()
NN.fit(proc_X_train_f, y_train_f.values, epochs=n_epochs, batch_size=4096, verbose=val_preds += NN.predict(proc_X_val_f)[:,0] / runs_per_fold
y_preds[:,i] += NN.predict(proc_X_test_f)[:,0] / runs_per_fold
full_val_preds[outf_ind] += val_preds
cv_gini = gini_normalizedc(y_val_f.values, val_preds)
cv_ginis.append(cv_gini)
print ('\nFold %i prediction cv gini: %.5f\n' %(i,cv_gini))
print('Mean out of fold gini: %.5f' % np.mean(cv_ginis))
print('Full validation gini: %.5f' % gini_normalizedc(y_train.values, full_val_preds))
y_pred_final = np.mean(y_preds, axis=1)
df_sub = pd.DataFrame({'id' : df_test.id,
'target' : y_pred_final},
```

```
columns = ['id','target'])
df_sub.to_csv('NN_EntityEmbed_10fold-sub.csv', index=False)
pd.DataFrame(full_val_preds).to_csv('NN_EntityEmbed_10fold-
val_preds.csv',index=False)
```

OUTPUT RESULTS

We can see that model did an excellent job on training and test data. we can see the results below:

```
/home/nikhil/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:76:
UserWarning: The `Merge` Fold 0 prediction cv gini: 0.28934
Fold 1 prediction cv gini: 0.25602
Fold 2 prediction cv gini: 0.28685
Fold 3 prediction cv gini: 0.27863
Fold 4 prediction cv gini: 0.26646
Fold 5 prediction cv gini: 0.28944
Fold 6 prediction cv gini: 0.27869
Fold 7 prediction cv gini: 0.27919
Mean out of fold gini: 0.27808
Full validation gini: 0.27713
```

We can see that
mean out of fold gini : 0.27808
and
full validation gini : 0.27713
which are really good score.

Conclusion

The model was successfully build by using embedding concept and so that we can predict the probability that an auto insurance policy holder files a claim. It allows the insurance company to further tailor their prices, and hopefully make auto insurance coverage more accessible to more drivers. If we train the model with more efficient data, there is more change to increase the efficiency of the model and we can get better results.

For detailed information about neural networks, you can read this online book[4] by Michael Nielsen.

For online lecture videos, you can watch Cs231n tutorial[5] from stanford university in youtube.

For more information, You can see research paper "A systematic comparison of context-counting vs. context-predicting semantic vectors"[6]

[4] http://neuralnetworksanddeeplearning.com/

[5] https://www.youtube.com/watch?v=NfnWJUyUJYU&list=PLkt2uSq6rBVctENoVBg1TpCC7OQi31AlC

[6] http://www.aclweb.org/anthology/P14-1023

The Nature Conservancy Fisheries Monitoring

Introduction

Nearly half of the world depends on seafood for their main source of protein. In the Western and Central Pacific, where 60% of the world's tuna is caught, illegal, unreported, and unregulated fishing practices are threatening marine ecosystems, global seafood supplies and local livelihoods. The Nature Conservancy is working with local, regional and global partners to preserve this fishery for the future.

Currently, the Conservancy is looking to the future by using cameras to dramatically scale the monitoring of fishing activities to fill critical science and compliance monitoring data gaps. Although these electronic monitoring systems work well and are ready for wider deployment, the amount of raw data produced is cumbersome and expensive to process manually.

In this project, we are going to develop algorithm to automatically detect and classify species of tunas, sharks and more that fishing boats catch, which will accelerate the video review process. Faster review and more reliable data will enable countries to reallocate human capital to management and enforcement activities which will have a positive impact on conservation and our planet.

Approach

In this project, we are going to design the model to automatically detect and classify species of tunas, sharks and more that finding boats catch, which will accelerate the video review. Input data consists of 8 different columns (different fishes). We implemented data preprocessing by resizing the images to rows :90 and column :160. In order to have better understanding about the data, we wrote a code which says, how many fishes are there of each types.

We observed that the majority of the images belong to ALB(type of fish) type. Later, we applied data preprocessing to our training data. We applied One Hot Encoding to represent the categorical variables as binary vectors. A one hot encoding is a representation of categorical variables as binary vectors. This first requires that the categorical values be mapped to integer values. Then, each integer value is represented as a binary vector that is all zero values except the index of the integer, which is marked with a 1. Then, we trained the model and predicted the probabilities of a image with respect to all 8 types on test images.

Code Implementation

Import Required Libraries

we are going to import required libraries, models, layers required for this project.

```
import os, cv2, random
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import log_loss
from sklearn.preprocessing import LabelEncoder
import matplotlib.pyplot as plt
from matplotlib import ticker
import seaborn as sns
%matplotlib inline
from keras.models import Sequential
from keras.layers import Dropout, Flatten, Convolution2D, MaxPooling2D,
ZeroPadding2D, Dense, from keras.optimizers import RMSprop, Adam
from keras.callbacks import EarlyStopping
from keras.utils import np_utils
from keras import backend as K
```

we are going to resize the images for training the model**.**

```
TRAIN_DIR = '../input/train/'
TEST_DIR = '../input/test_stg1/'
FISH_CLASSES = ['ALB', 'BET', 'DOL', 'LAG', 'NoF', 'OTHER', 'SHARK', 'YFT']
ROWS = 90  #720
COLS = 160 #1280
CHANNELS = 3
```

Loading and Preprocessing

Resize the images to 90 rows and 160 columns. We also keeping track of the labels as we loop through each image folder.

```
def get_images(fish):
    """Load files from train folder"""
    fish_dir = TRAIN_DIR+'{}'.format(fish)
    images = [fish+'/'+im for im in os.listdir(fish_dir)]
    return images
def read_image(src):
    """Read and resize individual images"""
    im = cv2.imread(src, cv2.IMREAD_COLOR)
    im = cv2.resize(im, (COLS, ROWS), interpolation=cv2.INTER_CUBIC)
    return im
files = []
y_all = []
for fish in FISH_CLASSES:
    fish_files = get_images(fish)
    files.extend(fish_files)
    y_fish = np.tile(fish, len(fish_files))
    y_all.extend(y_fish
  print("{0} photos of {1}".format(len(fish_files), fish))
  y_all = np.array(y_all)
```

The Output for the above code is

```
1719 photos of ALB
200 photos of BET
117 photos of DOL
67 photos of LAG
465 photos of NoF
299 photos of OTHER
176 photos of SHARK
```

734 photos of YFT

```
X_all = np.ndarray((len(files), ROWS, COLS, CHANNELS), dtype=np.uint8)
for i, im in enumerate(files):
X_all[i] = read_image(TRAIN_DIR+im)
if i%1000 == 0: print('Processed {} of {}'.format(i, len(files)))
print(X_all.shape)
```

The output for the above code is

```
Processed 0 of 3777
Processed 1000 of 3777
Processed 2000 of 3777
Processed 3000 of 3777
(3777, 90, 160, 3)
```

Display the Preprocessed Image

We are going to display the preprocessed image of each type.

```
uniq = np.unique(y_all, return_index=True)
for f, i in zip(uniq[0], uniq[1]):
plt.imshow(X_all[i])
plt.title(f)
plt.show()
```

Splitting the training Data

One-Hot-Encode the labels, then create a stratified train/validation split.

```
# One Hot Encoding Labels
y_all = LabelEncoder().fit_transform(y_all)
y_all = np_utils.to_categorical(y_all)
X_train, X_valid, y_train, y_valid = train_test_split(X_all, y_all, test_size=0.2, random_state=23, stratify=y_all)
```

The Model

```
optimizer = RMSprop(lr=1e-4)
objective = 'categorical_crossentropy'
def center_normalize(x):
    return (x - K.mean(x)) / K.std(x)
model = Sequential()
model.add(Activation(activation=center_normalize, input_shape=(ROWS,
COLS, CHANNELS)))
model.add(Convolution2D(32, 5, 5, border_mode='same', activation='relu',
dim_ordering='tf'))
model.add(Convolution2D(32, 5, 5, border_mode='same', activation='relu',
dim_ordering='tf'))
model.add(MaxPooling2D(pool_size=(2, 2), dim_ordering='tf'))
model.add(Convolution2D(64, 3, 3, border_mode='same', activation='relu',
dim_ordering='tf'))
model.add(Convolution2D(64, 3, 3, border_mode='same', activation='relu',
dim_ordering='tf'))
model.add(MaxPooling2D(pool_size=(2, 2), dim_ordering='tf'))
model.add(Convolution2D(128, 3, 3, border_mode='same',
activation='relu', dim_ordering='tf'))
model.add(Convolution2D(128, 3, 3, border_mode='same',
activation='relu', dim_ordering='tf'))
model.add(MaxPooling2D(pool_size=(2, 2), dim_ordering='tf'))
model.add(Convolution2D(256, 3, 3, border_mode='same',
activation='relu', dim_ordering='tf'))
model.add(Convolution2D(256, 3, 3, border_mode='same',
activation='relu', dim_ordering='tf'))
model.add(MaxPooling2D(pool_size=(2, 2), dim_ordering='tf'))
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
```

```
model.add(Dropout(0.5))
model.add(Dense(len(FISH_CLASSES)))
model.add(Activation('sigmoid'))
model.compile(loss=objective, optimizer=optimizer
```

Fit Model

```
early_stopping = EarlyStopping(monitor='val_loss', patience=4, verbose=1,
mode='auto')

model.fit(X_train, y_train, batch_size=64,
nb_epoch=10,validation_split=0.2, verbose=1, shuffle=True,
callbacks=[early_stopping])
```

we got the loss : 0.21 and val_loss : 0.29

```
preds = model.predict(X_valid, verbose=1)
print("Validation Log Loss: {}".format(log_loss(y_valid, preds)))
```

Now, Validation Log Loss : 0. 32

Predicting the Test Set

we are going to predict the probability on the test set images.

```
test_files = [im for im in os.listdir(TEST_DIR)]
test = np.ndarray((len(test_files), ROWS, COLS, CHANNELS),
dtype=np.uint8)
for i, im in enumerate(test_files):
   test[i] = read_image(TEST_DIR+im)
   test_preds = model.predict(test, verbose=1)
```

Now we are going to display the results for the top five test images

submission = pd.DataFrame(test_preds, columns=FISH_CLASSES)
submission.insert(0, 'image', test_files)
submission.head()

	Image	ALB	BET	DOL	LAG	NoF	OTHER	SHARK	YFT
0	img_05139.jpg	0.916198	0.155555	0.423464	0.091762	0.612399	0.269948	0.189734	0.766048
1	img_05657.jpg	0.916966	0.200932	0.354845	0.119914	0.566853	0.362618	0.207011	0.672743
2	img_02974.jpg	0.887490	0.213137	0.360376	0.144954	0.591034	0.346267	0.269630	0.692588
3	img_02837.jpg	0.885008	0.220824	0.360276	0.135349	0.594016	0.326693	0.261343	0.704006
4	img_00432.jpg	0.825517	0.252544	0.433282	0.207467	0.589070	0.389228	0.320735	0.651753

Carvana Image Masking Challenge

As with any big purchase, full information and transparency are key. While most everyone describes buying a used car as frustrating, it's just as annoying to sell one, especially online. Shoppers want to know everything about the car but they must rely on often blurry pictures and little information, keeping used car sales a largely inefficient, local industry. Carvana, a successful online used car startup, has seen opportunity to build long term trust with consumers and streamline the online buying process. An interesting part of their innovation is a custom rotating photo studio that automatically captures and processes 16 standard images of each vehicle in their inventory. While Carvana takes high quality photos, bright reflections and cars with similar colors as the background cause automation errors, which requires a skilled photo editor to change.

source: https://www.kaggle.com/c/carvana-image-masking-challenge

In this project, we are going to develop an algorithm that automatically removes the photo studio background. This will allow Carvana to superimpose cars on a variety of backgrounds. You'll be analyzing a dataset of photos, covering different vehicles with a wide variety of year, make, and model combinations.

Data Description

This dataset contains a large number of car images (as .jpg files). Each car has exactly 16 images, each one taken at different angles. Each car has a unique id and images are named according to id_01.jpg, id_02.jpg ... id_16.jpg. In addition to the images, you are also provided some basic metadata about the car make, model, year, and trim.

For the training set, you are provided a .gif file that contains the manually cutout mask for each image. The competition task is to automatically segment

the cars in the images in the test set folder. To deter hand labeling, we have supplemented the test set with car images that are ignored in scoring.

The metric used to score this competition requires that your submissions are in run-length encoded

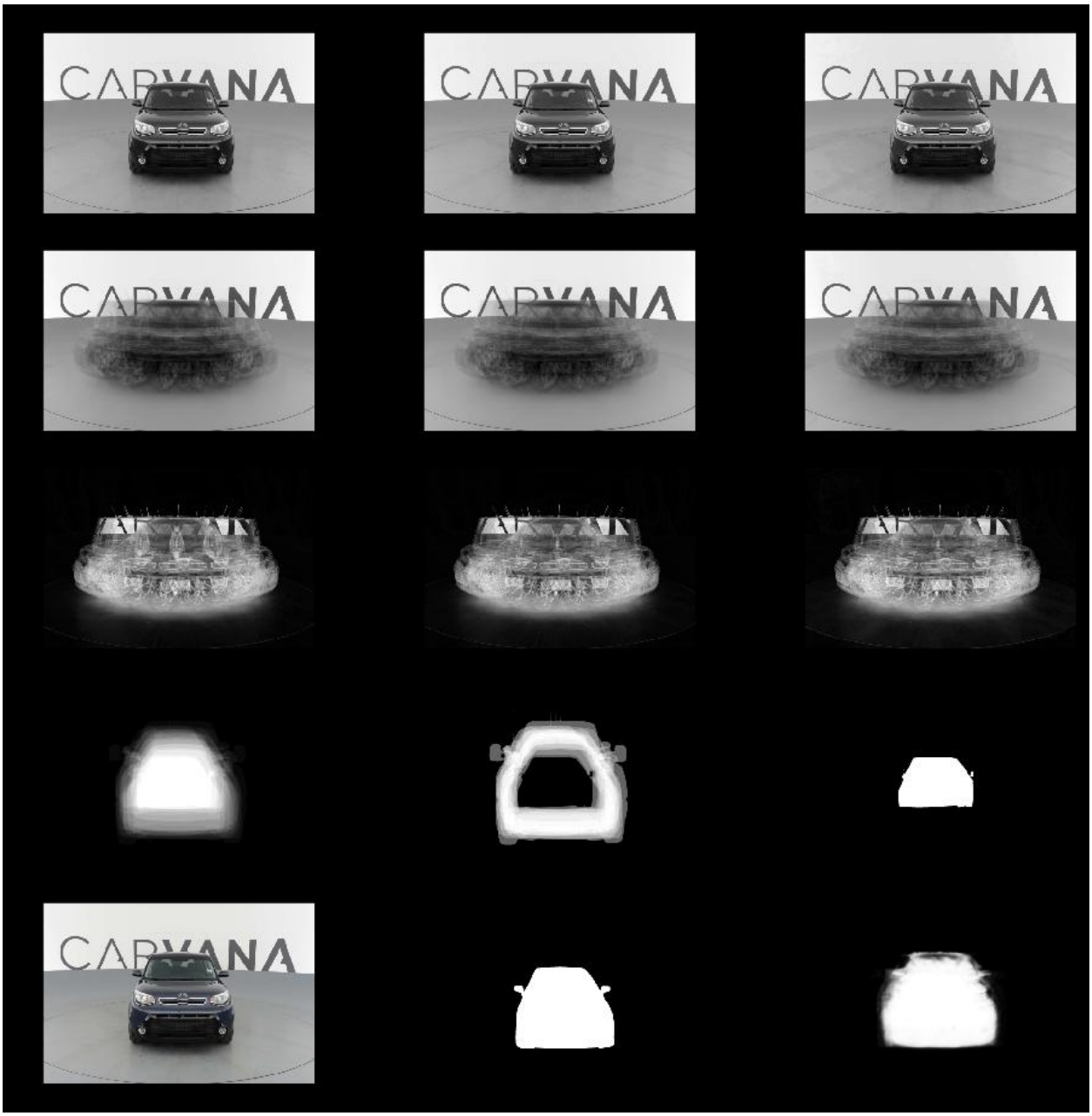
Implementation

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from skimage.io import imread
from skimage.transform import downscale_local_mean
from os.path import join
from tqdm import tqdm_notebook
import cv2
from sklearn.model_selection import train_test_split
input_folder = join('..', 'input')
df_mask = pd.read_csv(join(input_folder, 'train_masks.csv'), usecols=['img'])
ids_train = df_mask['img'].map(lambda s: s.split('_')[0]).unique()
imgs_idx = list(range(1, 17))
load_img = lambda im, idx: imread(join(input_folder, 'train', '{}_{:02d}.jpg'.format(im, idx)))
load_mask = lambda im, idx: imread(join(input_folder, 'train_masks', '{}_{:02d}_mask.gif'.format(im, idx)))
resize = lambda im: downscale_local_mean(im, (4,4) if im.ndim==2 else (4,4,1))
mask_image = lambda im, mask: (im * np.expand_dims(mask, 2))
num_train = 32  # len(ids_train)
```

```
# Load data for position id=1
X = np.empty((num_train, 320, 480, 12), dtype=np.float32)
y = np.empty((num_train, 320, 480, 1), dtype=np.float32)
 with tqdm_notebook(total=num_train) as bar:
   idx = 1 # Rotation index
   for i, img_id in enumerate(ids_train[:num_train]):
      imgs_id = [resize(load_img(img_id, j)) for j in imgs_idx]
      # Input is image + mean image per channel + std image per channel
      X[i, ..., :9] = np.concatenate([imgs_id[idx-1], np.mean(imgs_id, axis=0),     np.std(imgs_id, axis=0)], axis=2)
      y[i] = resize(np.expand_dims(load_mask(img_id, idx), 2)) / 255.
      del imgs_id # Free memory
      bar.update()
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.2, random_state=42)
# Concat overall y info to X
# This is important as the kernels of CNN used below has no information of its location
y_train_mean = y_train.mean(axis=0)
y_train_std = y_train.std(axis=0)
y_train_min = y_train.min(axis=0)
y_features = np.concatenate([y_train_mean, y_train_std, y_train_min], axis=2)
X_train[:, ..., -3:] = y_features
X_val[:, ..., -3:] = y_features
# Normalize input and output
X_mean = X_train.mean(axis=(0,1,2), keepdims=True)
```

```
X_std = X_train.std(axis=(0,1,2), keepdims=True)
X_train -= X_mean
X_train /= X_std
X_val -= X_mean
X_val /= X_std
# Create simple model
from keras.layers import Conv2D
from keras.models import Sequential
import keras.backend as K
model = Sequential()
model.add( Conv2D(16, 3, activation='relu', padding='same', input_shape=(320, 480, 12) ) )
model.add( Conv2D(32, 3, activation='relu', padding='same') )
model.add( Conv2D(1, 5, activation='sigmoid', padding='same') )
from keras.optimizers import Adam
from keras.losses import binary_crossentropy
smooth = 1.
  # From here: https://github.com/jocicmarko/ultrasound-nerve-segmentation/blob/master/train.py
def dice_coef(y_true, y_pred):
  y_true_f = K.flatten(y_true)
  y_pred_f = K.flatten(y_pred)
  intersection = K.sum(y_true_f * y_pred_f)
  return (2. * intersection + smooth) / (K.sum(y_true_f) + K.sum(y_pred_f) + smooth)
def bce_dice_loss(y_true, y_pred):
  return 0.5 * binary_crossentropy(y_true, y_pred) - dice_coef(y_true,
```

```
  y_pred)
  #Compile the Model
   model.compile(Adam(lr=1e-3), bce_dice_loss, metrics=['accuracy',
dice_coef])
  #Fit the Model
  history = model.fit(X_train, y_train, epochs=15, validation_data=(X_val,
  y_val), batch_size=5, verbose=2)
  pd.DataFrame(history.history)[['dice_coef', 'val_dice_coef']].plot()
  idx = 0
  x = X_val[idx]
  fig, ax = plt.subplots(5,3, figsize=(16, 16))
  ax = ax.ravel()
  cmaps = ['Reds', 'Greens', 'Blues']
  for i in range(x.shape[-1]):
  ax[i].imshow(x[...,i], cmap='gray') #cmaps[i%3])
  ax[i].set_title('channel {}'.format(i))
  ax[-3].imshow((x[...,:3] * X_std[0,...,:3] + X_mean[0,...,:3]) / 255.)
  ax[-3].set_title('X')
  ax[-2].imshow(y_train[idx,...,0], cmap='gray')
  ax[-2].set_title('y')
  y_pred = model.predict(x[None]).squeeze()
  ax[-1].imshow(y_pred, cmap='gray')
  ax[-1].set_title('y_pred')
```


CARVANA
CARVANA
CARVANA
CARVANA
CARVANA
CARVANA
CARVANA

```
plt.imshow(y_pred > 0.5, cmap='gray')
```

Time Series Prediction with LSTM Recurrent Neural Networks in Python with Keras

ABSTRACT

Technology is developing at a rapid pace and the luxury of the human being is increasing. Now a days, we can travel to almost all famous and important cities by plane. Due to more demand and profits in airline services, many airline services have been established to provide service to the customers and to gain profits. As an airline service, it is very important to know the expected number of travellers for the next months in order to change the frequency of flights between respective routes in order to increase the profits and decrease the loses and can provide excellent service to the customers by increasing the number of flights during more demand and decrease the frequency of flights during less demand. Now, we are going to build a model which can predict the expected number of passengers for the next month using LSTM Recurrent Neural Network in keras Deep Learning library having Tensor flow as back-end. Using the past 12 years international airline passengers data we are going to build a model, which can predict the expected number of passengers for next month. So that the airline services can use this model and plan their flights according to need based on expected number of passengers.

INTRODUCTION

It is very important to decide which neural network to use while building a model. we are going to work on prediction problem. Time series prediction problems are a difficult type of predictive modeling problem. Unlike regression predictive modeling, time series also adds the complexity of a sequence dependence among the input variables. A powerful type of neural network designed to handle sequence dependence is called recurrent neural networks. The Long Short-Term Memory network or LSTM network is a type of recurrent neural network used in deep learning because very large architectures can be successfully trained. Another important thing we need to consider to while building a model training data. The performance and accuracy of the model is mostly based on the data we used to train the network. So, we should use correct data to train the model. In this model we are going to use past 12 years international airline passenger data.

In this project, we are going to predict the expected number of passengers for next month using the LSTM Recurrent Neural Network by training the network using the past 12 years international airline passengers data. So, it makes easier for the airline services to plan the frequency of flights for the next month.

PROBLEM STATEMENT

The problem we are going to look at in this project is the International Airline Passengers prediction problem. This is a problem where, given a year and a month, the task is to predict the number of international airline passengers in units of 1,000. The data ranges from January 1949 to December 1960, or 12 years, with 144 observations. Irrespective of what our objective is, while working on any project, we should be able to provide a fully legitimate final "soft copy" of a project. This particular project provides a complexity of unlike regression predictive modeling, time series also adds the complexity of a sequence dependence among the input variables. Building from bottom up, there are a number of things to consider and get right.

1) In the beginning, it is very important to look at the data
2) to make it suitable for training the neural network,
3) which is called data cleaning
4) we need to choose the neural network which works
5) best for our problem
6) we need to create the model
7) Fit the model
8) Evaluate the model
9) Make predictions

Software Requirements

Anaconda Cloud

Anaconda is an open-source package manager, environment manager, and distribution of the Python and R programming languages. It is commonly used for largescale data processing, scientific computing, and predictive analytics, serving data scientists, developers, business analysts, and those working in DevOps. Anaconda offers a collection of over 720 open-source packages, and is available in both free and paid versions. The Anaconda distribution ships with the conda command-line utility. You can learn more about Anaconda cloud by reading the Anaconda Documentation pages.

Installing Anaconda cloud

The best way to install Anaconda is to download the latest Anaconda installer bash script, verify it, and then run it and you can see this page[7] for installation steps

a. Python

Python is a widely used high-level, general-purpose, interpreted, dynamic programming language. Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than would be possible in languages such as C++ or Java. The language provides constructs intended to enable clear programs on both a small and large scale.

To install python using command line

$ conda install -c anaconda python

b. Pandas

pandas is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language. To install pandas using command line

$ conda install -c conda-forge pandas

[7] https://www.pugetsystems.com/labs/hpc/How-to-Install-Anaconda-Python-and-First-Steps-for-Linux-and-Windows-917/

c. Jupyter

The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text. Uses include: data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more.

You can find jupyter by launching the anaconda navigator.

d. Numpy

NumPy is the fundamental package for scientific computing with Python. It contains among other things:a powerful N-dimensional array object sophisticated (broadcasting) functions tools for integrating C/C++ and Fortran code.

To install Numpy using command line

$ conda install -c anaconda numpy

e. Matplotlib

Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. Matplotlib can be used in Python scripts, the Python and Ipython shells, the Jupyter notebook, web application servers, and four graphical user interface toolkits.

To install matplotlib using command line

$ conda install -c anaconda matplotlib

f. Scikit - learn

Scikit - Learn It is used for machine learning in python.Simple and efficient tools for data mining and data analysis. Accessible to everybody, and reusable in various contexts.Built on NumPy, SciPy, and matplotlib.

To install scikit – learn using command line

$ conda install -c anaconda scikit-learn

g. Tensor flow

Tensor Flow is an open source software library for high performance numerical computation. Its flexible architecture allows easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices.Originally developed by researchers and engineers from the Google Brain team within Google's AI organization, it comes with strong support for machine learning and deep learning and the flexible numerical computation core is used across many other scientific domains. We are not using tensor flow directly. But, we are using it as a backendfor keras

To install tensor flow using command line

$ conda install -c anaconda tensorflowgpu

h. Keras

Keras is a high-level neural networks API, written in Python and capable of running on top of Tensor Flow, CNTN or Theano. It was developed with a focus on enabling fast experimentation. Being able to go from idea to result with the least possible delay is key to doing good research.

Keras allows us : For easy and fast prototyping (through user friendliness, modularity, and extensibility). Supports both convolutional networks and recurrent networks, as well as combinations of the two to Runs seamlessly on CPU and GPU.

To install keras using command line

$ conda install -c anaconda keras-gpu

CONCEPTUAL MODEL

Long Short-Term Memory Network

The Long Short-Term Memory network, or LSTM network, is a recurrent neural network that is trained using Back propagation Through Time and overcomes the vanishing gradient problem. As such, it can be used to create large recurrent networks that in turn can be used to address difficult sequence problems in machine learning and achieve state-of-the-art results. Instead of neurons, LSTM networks have memory blocks that are connected through layers. A block has components that make it smarter than a classical neuron and a memory for recent sequences. A block contains gates that manage the block's state and output. A block operates upon an input sequence and each gate within a block uses the sigmoid activation units to control whether they are triggered or not, making the change of state and addition of information flowing through the block conditional.

There are three types of gates within a unit:

- Forget Gate: conditionally decides what information to throw away from the block.
- Input Gate: conditionally decides which values from the input to update the memory state.
- Output Gate: conditionally decides what to output based on input and the memory of the block.

Each unit is like a mini-state machine where the gates of the units have weights that are learned during the training procedure. You can see how you may achieve sophisticated learning and memory from a layer of LSTMs, and it is not hard to imagine how higher-order abstractions may be layered with multiple such layers.

For advanced information, you can see this research paper[8]

[8] https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/43905.pdf

METHODOLOGY AND APPROACH

Outlined below is the methodology adopted to achieve the end resultThe problem we are going to look at in this project is the International Airline Passengers prediction problem. This is a problem where, given a year and a month, the task is to predict the number of international airline passengers in units of 1,000. The data ranges from January 1949 to December 1960, or 12 years, with 144 observations.

The dataset is available for free from the DataMarket webpage as a CSV download[9] with the filename "*international-airline-passengers.csv*".

Below is a sample of the first few lines of the file.

"Month","International airline passengers: monthly totals in

thousands. Jan 49 ? Dec 60"

"1949-01",112

"1949-02",118

"1949-03",132

"1949-04",129

"1949-05",121

We can load this dataset easily using the Pandas library. Weare not interested in the date, given that each observation is separated by the same interval of one month. Therefore, when we load the dataset we can exclude the first column. The downloaded dataset also has footer information that we can exclude with the skipfooter argument to pandas.read_csv() set to 3 for the 3 footer lines. Once loaded we can easily plot the whole dataset. The code to load and plot the dataset is listed below.

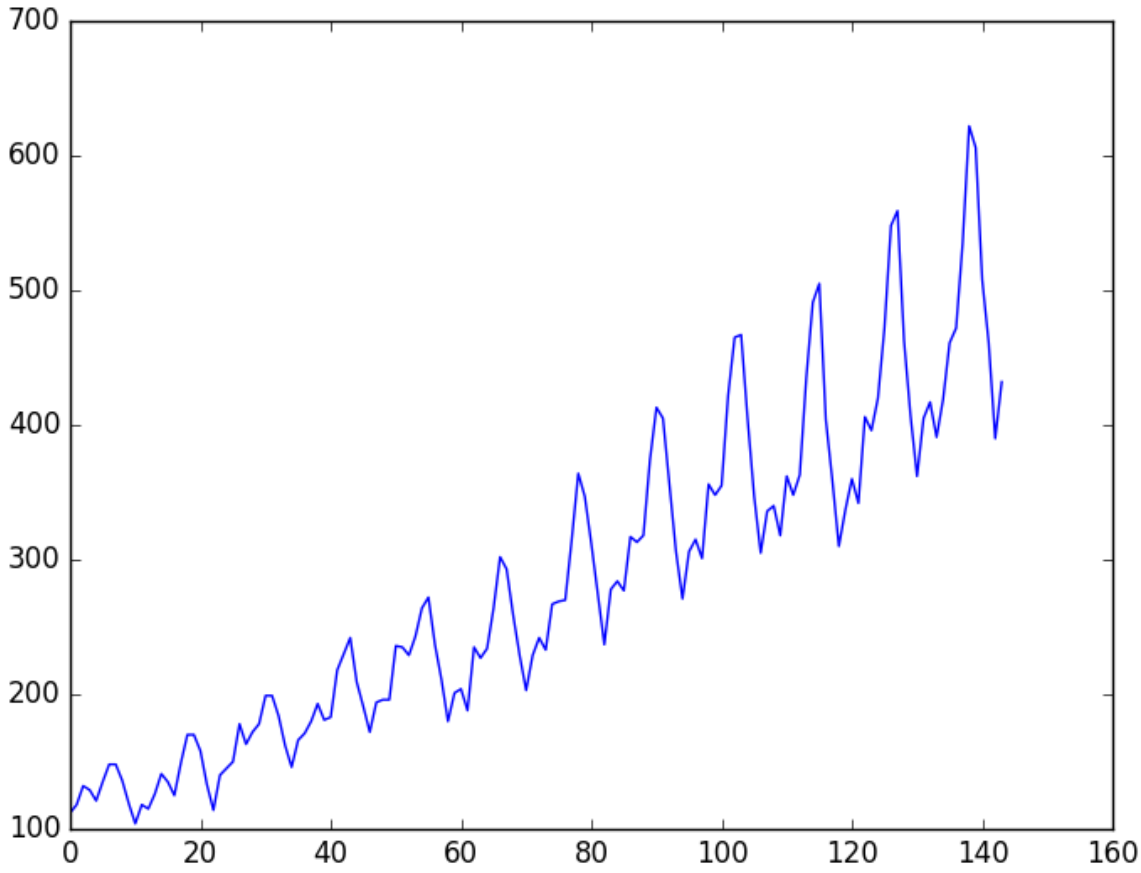
```
import pandas
import matplotlib.pyplot as plt
dataset = pandas.read_csv('international-airlinepassengers.
csv', usecols=[1], engine='python', skipfooter=3)
```

[9] https://datamarket.com/data/set/22u3/international-airline-passengers-monthly-totals-in-thousands-jan-49-dec-60#!ds=22u3&display=line

plt.plot(dataset)

plt.show()

You can see an upward trend in the dataset over time. You can also see some periodicity to the dataset that probably corresponds to the Northern Hemisphere vacation period



LSTM Network for Regression

We can phrase the problem as a regression problem. That is, given the number of passengers (in units of thousands) this month, what is the number of passengers next month ?

We can write a simple function to convert our single column of data into a two-column dataset: the first column containing this month's (t) passenger count and the second column containing next month's (t+1)

passenger count, to be predicted.

Before we get started, let's first import all of the functions and classes we intend to use.

```
import numpy
import matplotlib.pyplot as plt
import pandas
import math
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
Before we do anything, it is a good idea to fix the random
number seed to ensure our results are reproducible.
# fix random seed for reproducibility
numpy.random.seed(7)
```

We can also use the code from the previous section to load the dataset as a Pandas data frame. We can then extract the NumPy array from the data frame and convert the integer values to floating point values, which are more suitable for modeling with a neural network.

```
# load the dataset
dataframe = pandas.read_csv('international-airlinepassengers.csv',
usecols=[1], engine='python', skipfooter=3)
dataset = dataframe.values
dataset = dataset.astype('float32')
```

LSTMs are sensitive to the scale of the input data, specifically when the sigmoid (default) or tanh activation functions are used. It can be a good practice to rescale the data to the range of 0-to-1, also called normalizing. We

can easily normalize the dataset using the MinMaxScaler3 preprocessing class from the scikit-learn library.

```
# normalize the dataset
scaler = MinMaxScaler(feature_range=(0, 1))
dataset = scaler.fit_transform(dataset)
```

After we model our data and estimate the skill of our model on the training dataset, we need to get an idea of the skill of the model on new unseen data. For a normal classification or regression problem, we would do this using cross validation.

With time series data, the sequence of values is important. A simple method that we can use is to split the ordered dataset into train and test datasets. The code below calculates the index of the split point and separates the data into the training datasets with 67% of the observations that

we can use to train our model, leaving the remaining 33% for testing the model.

```
# split into train and test sets
train_size = int(len(dataset) * 0.67)
test_size = len(dataset) - train_size
train, test = dataset[0:train_size,:],
dataset[train_size:len(dataset),:]
print(len(train), len(test))
```

Now we can define a function to create a new dataset, as described above.

The function takes two arguments: the dataset, which is a NumPy array that we want to convert into a dataset, and the look_back, which is the number of previous time steps to use as input variables to predict the next time period — in this case defaulted to 1. This default will create a dataset where X is the number of passengers at a given time (t) and Y is the number of passengers at the next time (t + 1). It can be configured, and we will by constructing a differently shaped dataset in the next section.

```
# convert an array of values into a dataset matrix
```

```
def create_dataset(dataset, look_back=1):
dataX, dataY = [], []
for i in range(len(dataset)-look_back-1):
a = dataset[i:(i+look_back), 0]
dataX.append(a)
dataY.append(dataset[i + look_back, 0])
return numpy.array(dataX), numpy.array(dataY)
```

Let's take a look at the effect of this function on the first rows of the dataset (shown in the unnormalized form for clarity).

```
X Y
112 118
118 132
132 129
129 121
121 135
```

If you compare these first 5 rows to the original dataset sample listed in the previous section, you can see the X=t and Y=t+1 pattern in the numbers.

Let's use this function to prepare the train and test datasets for modeling.

```
# reshape into X=t and Y=t+1
look_back = 1
trainX, trainY = create_dataset(train, look_back)
testX, testY = create_dataset(test, look_back)
```

The LSTM network expects the input data (X) to be provided with a specific array structure in the form of: *[samples, time steps, features]*.

Currently, our data is in the form: [*samples, features*] and we are framing the problem as one time step for each sample. We can transform the prepared train and test input data into the expected structure using numpy.reshape() as follows:

```
# reshape input to be [samples, time steps, features]
trainX = numpy.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))
testX = numpy.reshape(testX, (testX.shape[0], 1, testX.shape[1]))
```

We are now ready to design and fit our LSTM network for this problem.

The network has a visible layer with 1 input, a hidden layer with 4 LSTM blocks or neurons, and an output layer that makes a single value prediction. The default sigmoid activation function is used for the LSTM blocks. The network is trained for 100 epochs and a batch size of 1 is used.

```
# create and fit the LSTM network
model = Sequential()
model.add(LSTM(4, input_shape=(1, look_back)))
model.add(Dense(1))
model.compile(loss='mean_squared_error',
optimizer='adam')
model.fit(trainX, trainY, epochs=100, batch_size=1,
verbose=2)
```

Once the model is fit, we can estimate the performance of the model on the train and test datasets. This will give us a point of comparison for new models. Note that we invert the predictions before calculating error scores to ensure that performance is reported in the same units as the original data (thousands of passengers per month).

```
# make predictions
trainPredict = model.predict(trainX)
testPredict = model.predict(testX)
# invert predictions
trainPredict = scaler.inverse_transform(trainPredict)
```

```
trainY = scaler.inverse_transform([trainY])
testPredict = scaler.inverse_transform(testPredict)
testY = scaler.inverse_transform([testY])
# calculate root mean squared error
trainScore = math.sqrt(mean_squared_error(trainY[0], trainPredict[:,0]))
print('Train Score: %.2f RMSE' % (trainScore))
testScore = math.sqrt(mean_squared_error(testY[0],
testPredict[:,0]))
print('Test Score: %.2f RMSE' % (testScore))
```

Finally, we can generate predictions using the model for both the train and test dataset to get a visual indication of the skill of the model. Because of how the dataset was prepared, we must shift the predictions so that they align on the x-axis with the original dataset. Once prepared, the data is plotted, showing the original dataset in blue, the predictions for the training dataset in green, and the predictions on the unseen test dataset in red

```
# shift train predictions for plotting
trainPredictPlot = numpy.empty_like(dataset)
trainPredictPlot[:, :] = numpy.nan
trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict
# shift test predictions for plotting
testPredictPlot = numpy.empty_like(dataset)
testPredictPlot[:, :] = numpy.nan
testPredictPlot[len(trainPredict)+
(look_back*2)+1:len(dataset)-1, :] = testPredict
# plot baseline and predictions
plt.plot(scaler.inverse_transform(dataset))
plt.plot(trainPredictPlot)
plt.plot(testPredictPlot)
```

```
plt.show()
```

OUTPUT AND RESULTS

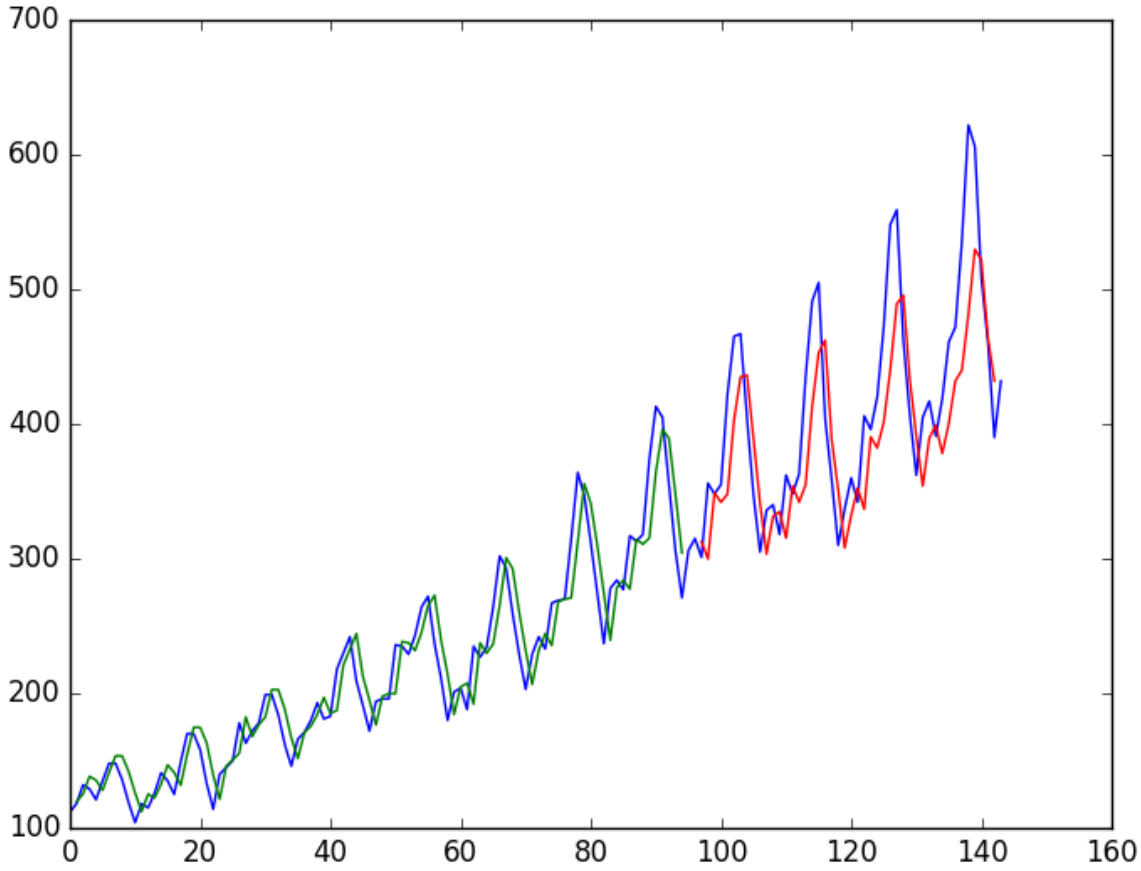
We can see that model did an excellent job of fitting both the training and test datasets

Epoch 1/1
- 9s - loss: 0.0057
Epoch 1/1
- 2s - loss: 0.0141
Epoch 1/1
- 2s - loss: 0.0099
Epoch 1/1
- 2s - loss: 0.0072
Epoch 1/1
- 2s - loss: 0.0060
Epoch 1/1
- 2s - loss: 0.0056
Epoch 1/1
- 2s - loss: 0.0056
Epoch 1/1
- 2s - loss: 0.0056
Epoch 1/1
- 2s - loss: 0.0056
Epoch 1/1
- 2s - loss: 0.0056
Epoch 1/1
- 2s - loss: 0.0056
Epoch 1/1
- 2s - loss: 0.0056
Epoch 1/1
- 2s - loss: 0.0056
Epoch 1/1
- 2s - loss: 0.0056
Epoch 1/1
- 2s - loss: 0.0056
Epoch 1/1
- 2s - loss: 0.0056
Epoch 1/1
- 2s - loss: 0.0056
Epoch 1/1
- 2s - loss: 0.0056
Epoch 1/1
- 2s - loss: 0.0056
Epoch 1/1
- 2s - loss: 0.0056
Epoch 1/1
- 2s - loss: 0.0056
Epoch 1/1
- 2s - loss: 0.0056
Epoch 1/1
- 2s - loss: 0.0056
Epoch 1/1
- 2s - loss: 0.0056
Epoch 1/1

- 2s - loss: 0.0055
Epoch 1/1
- 2s - loss: 0.0055
Epoch 1/1
- 2s - loss: 0.0055
Epoch 1/1
- 2s - loss: 0.0055
Epoch 1/1
- 2s - loss: 0.0055
Epoch 1/1
- 2s - loss: 0.0055
Epoch 1/1
- 2s - loss: 0.0055
Epoch 1/1
- 2s - loss: 0.0054
Epoch 1/1
- 2s - loss: 0.0054
Epoch 1/1
- 2s - loss: 0.0054
Epoch 1/1
- 2s - loss: 0.0054
Epoch 1/1
- 2s - loss: 0.0054
Epoch 1/1
- 2s - loss: 0.0054
Epoch 1/1
- 2s - loss: 0.0054
Epoch 1/1
- 2s - loss: 0.0054
Epoch 1/1
- 2s - loss: 0.0054
Epoch 1/1
- 2s - loss: 0.0053
Epoch 1/1
- 2s - loss: 0.0053
Epoch 1/1
- 2s - loss: 0.0053
Epoch 1/1
- 2s - loss: 0.0053
Epoch 1/1
- 2s - loss: 0.0053
Epoch 1/1
- 2s - loss: 0.0053
Epoch 1/1
- 2s - loss: 0.0053
Epoch 1/1
- 2s - loss: 0.0052
Epoch 1/1
- 2s - loss: 0.0052
Epoch 1/1
- 2s - loss: 0.0052
Epoch 1/1
- 2s - loss: 0.0052
Epoch 1/1
- 2s - loss: 0.0052
Epoch 1/1

- 2s - loss: 0.0052
Epoch 1/1
- 2s - loss: 0.0052
Epoch 1/1
- 2s - loss: 0.0051
Epoch 1/1
- 2s - loss: 0.0051
Epoch 1/1
- 2s - loss: 0.0051
Epoch 1/1
- 2s - loss: 0.0051
Epoch 1/1
- 2s - loss: 0.0051
Epoch 1/1
- 2s - loss: 0.0051
Epoch 1/1
- 2s - loss: 0.0050
Epoch 1/1
- 2s - loss: 0.0050
Epoch 1/1
- 2s - loss: 0.0050
Epoch 1/1
- 2s - loss: 0.0050
Epoch 1/1
- 2s - loss: 0.0050
Epoch 1/1
- 2s - loss: 0.0049
Epoch 1/1
- 2s - loss: 0.0049
Epoch 1/1
- 2s - loss: 0.0049
Epoch 1/1
- 2s - loss: 0.0049
Epoch 1/1
- 2s - loss: 0.0048
Epoch 1/1
- 2s - loss: 0.0048
Epoch 1/1
- 2s - loss: 0.0048
Epoch 1/1
- 2s - loss: 0.0048
Epoch 1/1
- 2s - loss: 0.0047
Epoch 1/1
- 2s - loss: 0.0047
Epoch 1/1
- 2s - loss: 0.0047
Epoch 1/1
- 2s - loss: 0.0046
Epoch 1/1
- 2s - loss: 0.0046
Epoch 1/1
- 2s - loss: 0.0046
Epoch 1/1
- 2s - loss: 0.0045
Epoch 1/1

- 2s - loss: 0.0045
Epoch 1/1
- 2s - loss: 0.0045
Epoch 1/1
- 2s - loss: 0.0044
Epoch 1/1
- 2s - loss: 0.0044
Epoch 1/1
- 2s - loss: 0.0044
Epoch 1/1
- 2s - loss: 0.0043
Epoch 1/1
- 2s - loss: 0.0043
Epoch 1/1
- 2s - loss: 0.0042
Epoch 1/1
- 2s - loss: 0.0042
Epoch 1/1
- 2s - loss: 0.0042
Epoch 1/1
- 2s - loss: 0.0041
Epoch 1/1
- 2s - loss: 0.0041
Epoch 1/1
- 2s - loss: 0.0040
Epoch 1/1
- 2s - loss: 0.0040
Epoch 1/1
- 2s - loss: 0.0040
Epoch 1/1
- 2s - loss: 0.0039
Epoch 1/1
- 2s - loss: 0.0038
Epoch 1/1
- 2s - loss: 0.0038
Epoch 1/1
- 2s - loss: 0.0037
Epoch 1/1
- 2s - loss: 0.0036
Train Score: 29.79 RMSE
Test Score: 79.47 RMSE



We can see that the model has an average error of about 29 passengers (in thousands) on the training dataset, and about 79 passengers (in thousands) on the test dataset, which is really good.

CONCLUSION

The model was successfully created by using LSTM Recurrent Neural Network and trained and it produced excellent results, We can see that the model has an average error of about 29 passengers (in thousands) on the training dataset, and about 79 passengers (in thousands) on the test dataset, which is really good. If we train the model with more efficient data, there is more change to increase the efficiency of the model and we can get better results.

Multi Label Classification

In this project, we are going to design a keras deep neural network to return multiple predictions i.e Multi label classification with keras. We are going to use a simplified version of VGGNet model. The VGGNet model was first introduced by Simonyan and Zisserman in their 2014 paper.

Our multi-label classification dataset

The dataset we will be using in our keras multi label classification project consists of 2167 images across six categories, including :

- **Black jeans (344 images)**
- **Blue dress (386 images)**
- **Blue jeans (356 images)**
- **Blue shirt (369 images)**
- **Red dress (380 images)**
- **Red shirt (332 images)**

The entire process of downloading the images and manually removing the irrelevant images for each of the six classes took approximately 1 to 2 hours.

Keras network architecture for multi-label classification

We import the relevant keras modules ,

```
# import the necessary packages
from keras.models import Sequential
from keras.layers.normalization import BatchNormalization
from keras.layers.convolutional import Conv2D
from keras.layers.convolutional import MaxPooling2D
from keras.layers.core import Activation
from keras.layers.core import Flatten
from keras.layers.core import Dropout
from keras.layers.core import Dense
from keras import backend as K
```

We create our smallerVGGNet class:

```
class SmallerVGGNet:
    @staticmethod
    def build(width, height, depth, classes, finalAct="softmax"):
        # initialize the model along with the input shape to be
        # "channels last" and the channels dimension itself
        model = Sequential()
        inputShape = (height, width, depth)
        chanDim = -1
        # if we are using "channels first", update the input shape
        # and channels dimension
        if K.image_data_format() == "channels_first":
            inputShape = (depth, height, width)
            chanDim = 1
```

The build method requires four parameters — width , height , depth , and classes . The depth specifies the number of channels in an input image, and classes is the number (integer) of categories/classes (not the class labels themselves). We'll use these parameters in our training script to instantiate the model with a 96 x 96 x 3 input volume.

The optional argument, finalAct (with a default value of "softmax") will be utilized at the end of the network architecture. Changing this value from softmax to sigmoid will enable us to perform multi-label classification with Keras.

From there, we enter the body of build , initializing the model and defaulting to"channels_last" architecture (with a convenient switch for backends that support "channels_first" architecture).

Let's build the first CONV => RELU => POOL block:

```
# CONV => RELU => POOL
model.add(Conv2D(32, (3, 3), padding="same",
    input_shape=inputShape))
model.add(Activation("relu"))
model.add(BatchNormalization(axis=chanDim))
model.add(MaxPooling2D(pool_size=(3, 3)))
model.add(Dropout(0.25))
```

Our CONV layer has 32 filters with a 3 x 3 kernel and RELU activation (Rectified Linear Unit). We apply batch normalization, max pooling, and 25% dropout.

Dropout is the process of randomly disconnecting nodes from the *current* layer to the *next* layer. This process of random disconnects naturally helps the network to reduce overfitting as no one single node in the layer will be responsible for predicting a certain class, object, edge, or corner.

From there we have two sets of **(CONV => RELU) * 2 => POOL** blocks

```
# (CONV => RELU) * 2 => POOL
```

```
model.add(Conv2D(64, (3, 3), padding="same"))
model.add(Activation("relu"))
model.add(BatchNormalization(axis=chanDim))
model.add(Conv2D(64, (3, 3), padding="same"))
model.add(Activation("relu"))
model.add(BatchNormalization(axis=chanDim))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

# (CONV => RELU) * 2 => POOL
model.add(Conv2D(128, (3, 3), padding="same"))
model.add(Activation("relu"))
model.add(BatchNormalization(axis=chanDim))
model.add(Conv2D(128, (3, 3), padding="same"))
model.add(Activation("relu"))
model.add(BatchNormalization(axis=chanDim))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
```

Notice the changes in filters, kernels, and pool sizes in this code block which work together to progressively reduce the spatial size but increase depth.

These blocks are followed by our only set of **FC => RELU** layers:

```
# first (and only) set of FC => RELU layers
model.add(Flatten())
model.add(Dense(1024))
model.add(Activation("relu"))
model.add(BatchNormalization())
model.add(Dropout(0.5))
```

```
        # use a *softmax* activation for single-label classification
        # and *sigmoid* activation for multi-label classification
        model.add(Dense(classes))
        model.add(Activation(finalAct))

        # return the constructed network architecture
        return model
```

Fully connected layers are placed at the end of the network.

Implementing our keras model for multi-label classification

Now that we have implemented smallerVGGNet, let's create train.py, the script we will use to train our keras network for multi-label classification.

```
# set the matplotlib backend so figures can be saved in the background
import matplotlib
matplotlib.use("Agg")
# import the necessary packages
from keras.preprocessing.image import ImageDataGenerator
from keras.optimizers import Adam
from keras.preprocessing.image import img_to_array
from sklearn.preprocessing import MultiLabelBinarizer
from sklearn.model_selection import train_test_split
from pyimagesearch.smallervggnet import SmallerVGGNet
import matplotlib.pyplot as plt
from imutils import paths
import numpy as np
import argparse
```

```
import random
import pickle
import cv2
import os
```

We import the packages and modules required for this script. Our code specifies a matplotlib backend so that we can save our plot figure in the background.

lets parse command line arguments:

```
# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-d", "--dataset", required=True,
    help="path to input dataset (i.e., directory of images)")
ap.add_argument("-m", "--model", required=True,
    help="path to output model")
ap.add_argument("-l", "--labelbin", required=True,
    help="path to output label binarizer")
ap.add_argument("-p", "--plot", type=str, default="plot.png",
    help="path to output accuracy/loss plot")
args = vars(ap.parse_args())
```

we are working with four command line arguments:

1. **--dataset : The path to our dataset.**
2. **--model : The path to our output serialized Keras model.**
3. **--labelbin : The path to our output multi-label binarizer object.**
4. **--plot : The path to our output plot of training loss and accuracy.**

lets move on to initializing some important variables that play critical roles in our training process :

```
# initialize the number of epochs to train for, initial learning rate,
# batch size, and image dimensions
EPOCHS = 75
INIT_LR = 1e-3
BS = 32
IMAGE_DIMS = (96, 96, 3)
```

Our network will train 75 epochs in order to learn patterns by incremental improvements via backprpagation.

We are establishing an intial learning rate of 1e-3

The batch size is 32.

Our images are 96 * 96 and contain 3 channels.

```
# grab the image paths and randomly shuffle them
print("[INFO] loading images...")
imagePaths = sorted(list(paths.list_images(args["dataset"])))
random.seed(42)
random.shuffle(imagePaths)


# initialize the data and labels
data = []
labels = []
```

Here we are grabbing the imagepaths and shuffling them randomly, followed by initializing data and labels lists.

Next, we are to loop over the imagepaths, preprocess the image data, and extract multi-class-labels.

```
# loop over the input images
for imagePath in imagePaths:
    # load the image, pre-process it, and store it in the data list
```

```
image = cv2.imread(imagePath)
image = cv2.resize(image, (IMAGE_DIMS[1], IMAGE_DIMS[0]))
image = img_to_array(image)
data.append(image)

# extract set of class labels from the image path and update the
# labels list
l = label = imagePath.split(os.path.sep)[-2].split("_")
labels.append(l)
```

First, we load each image into memory. Then, we perform preprocessing (an important step of the deep learning pipeline). We append the image to data.

Above code handle splitting the image path into multiple labels for our multi-label classification task. A 2-element list is created and is then appended to the labels list. Here's an example broken down in the terminal so you can see what's going on during the multi-label parsing:

```
$ python
>>> import os
>>> labels = []
>>> imagePath = "dataset/red_dress/long_dress_from_macys_red.png"
>>> l = label = imagePath.split(os.path.sep)[-2].split("_")
>>> l
['red', 'dress']
>>> labels.append(l)
>>>
>>> imagePath = "dataset/blue_jeans/stylish_blue_jeans_from_your_favorite_store.png"
>>> l = label = imagePath.split(os.path.sep)[-2].split("_")
>>> labels.append(l)
```

```
>>>
>>> imagePath = "dataset/red_shirt/red_shirt_from_target.png"
>>> l = label = imagePath.split(os.path.sep)[-2].split("_")
>>> labels.append(l)
>>>
>>> labels
[['red', 'dress'], ['blue', 'jeans'], ['red', 'shirt']]
```

As you can see, the labels list is a "list of lists" — each element of labels is a 2-element list. The two labels for each list is constructed based on the file path of the input image.

We're not quite done with preprocessing:

```
# scale the raw pixel intensities to the range [0, 1]
data = np.array(data, dtype="float") / 255.0
labels = np.array(labels)
print("[INFO] data matrix: {} images ({:.2f}MB)".format(
    len(imagePaths), data.nbytes / (1024 * 1000.0)))
```

Our data list contains images stored as Numpy arrays. In a single line of code, we convert the list to a Numpy array and scale the pixel intensities to range [0,1].

We also convert labels labels to Numpy array as well.

From there, lets binarize the labels – the below block is important for this project.

```
# binarize the labels using scikit-learn's special multi-label
# binarizer implementation
print("[INFO] class labels:")
mlb = MultiLabelBinarizer()
labels = mlb.fit_transform(labels)
```

```
# loop over each of the possible class labels and show them
for (i, label) in enumerate(mlb.classes_):
    print("{}. {}".format(i + 1, label))
```

In order to binarize our labels for multi-class classification, we need to utilize the scikit-learn library's MultiLabelBinarizer class. You *cannot* use the standard LabelBinarizer class for multi-class classification. Transform our human-readable labels into a vector that encodes which class(es) are present in the image.

Here's an example showing how MultiLabelBinarizer transforms a tuple of ("red", "dress") to a vector with six total categories:

```
$ python
>>> from sklearn.preprocessing import MultiLabelBinarizer
>>> labels = [
...     ("blue", "jeans"),
...     ("blue", "dress"),
...     ("red", "dress"),
...     ("red", "shirt"),
...     ("blue", "shirt"),
...     ("black", "jeans")
... ]
>>> mlb = MultiLabelBinarizer()
>>> mlb.fit(labels)
MultiLabelBinarizer(classes=None, sparse_output=False)
>>> mlb.classes_
array(['black', 'blue', 'dress', 'jeans', 'red', 'shirt'], dtype=object)
>>> mlb.transform([("red", "dress")])
array([[0, 0, 1, 0, 1, 0]])
```

One-hot encoding transforms categorical labels from a single integer to a vector. The Python shell (not to be confused with the code blocks fortrain.py) two categorical labels are "hot" (represented by a *"1"* in the array), indicating the presence of each label. In this case "dress" and "red" are hot in the array. All other labels have a value of *"0"*.

Let's construct the training and testing splits as well as initialize the data augmenter:

```
# partition the data into training and testing splits using 80% of
# the data for training and the remaining 20% for testing
(trainX, testX, trainY, testY) = train_test_split(data,
    labels, test_size=0.2, random_state=42)


# construct the image generator for data augmentation
aug = ImageDataGenerator(rotation_range=25, width_shift_range=0.1,
    height_shift_range=0.1, shear_range=0.2, zoom_range=0.2,
    horizontal_flip=True, fill_mode="nearest")
```

splitting the data for training and testing is common in machine learning practice – I have allocated 80% of the images for training data and 20% for testing data. This is handled by scikit – learn.

Our data augmenter object is initialized. Data augmentation is a best practice and a most-likely a "must" if you are working with less than 1,000 images per class.

Next, let's build the model and initialize the Adam optimizer:

```
# initialize the model using a sigmoid activation as the final layer
# in the network so we can perform multi-label classification
print("[INFO] compiling model...")
model = SmallerVGGNet.build(
    width=IMAGE_DIMS[1], height=IMAGE_DIMS[0],
    depth=IMAGE_DIMS[2], classes=len(mlb.classes_),
```

```
    finalAct="sigmoid")

# initialize the optimizer
opt = Adam(lr=INIT_LR, decay=INIT_LR / EPOCHS)
```

we build our SmallerVGGNet model, noting the finalAct="sigmoid" parameter indicating that we'll be performing multi-label classification.

From there, we'll compile the model and kick off training.

```
# compile the model using binary cross-entropy rather than
# categorical cross-entropy -- this may seem counterintuitive for
# multi-label classification, but keep in mind that the goal here
# is to treat each output label as an independent Bernoulli
# distribution
model.compile(loss="binary_crossentropy", optimizer=opt,
    metrics=["accuracy"])

# train the network
print("[INFO] training network...")
H = model.fit_generator(
    aug.flow(trainX, trainY, batch_size=BS),
    validation_data=(testX, testY),
    steps_per_epoch=len(trainX) // BS,
    epochs=EPOCHS, verbose=1)
```

we compile the model using binary cross-entropy rather than categorical cross-entropy. This may seem counterintuitive for multi-label classification; however, the goal is to treat each output label as an *independent Bernoulli*

distribution and *we want to penalize each output node independently*. From there we launch the training process with our data augmentation generator . After training is complete we can save our model and label binarizer to disk:

```
# save the model to disk
print("[INFO] serializing network...")
model.save(args["model"])

# save the multi-label binarizer to disk
print("[INFO] serializing label binarizer...")
f = open(args["labelbin"], "wb")
f.write(pickle.dumps(mlb))
f.close()
```

From there, we plot accuracy and loss.

```
# plot the training loss and accuracy
plt.style.use("ggplot")
plt.figure()
N = EPOCHS
plt.plot(np.arange(0, N), H.history["loss"], label="train_loss")
plt.plot(np.arange(0, N), H.history["val_loss"], label="val_loss")
plt.plot(np.arange(0, N), H.history["acc"], label="train_acc")
plt.plot(np.arange(0, N), H.history["val_acc"], label="val_acc")
plt.title("Training Loss and Accuracy")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend(loc="upper left")
plt.savefig(args["plot"])
```

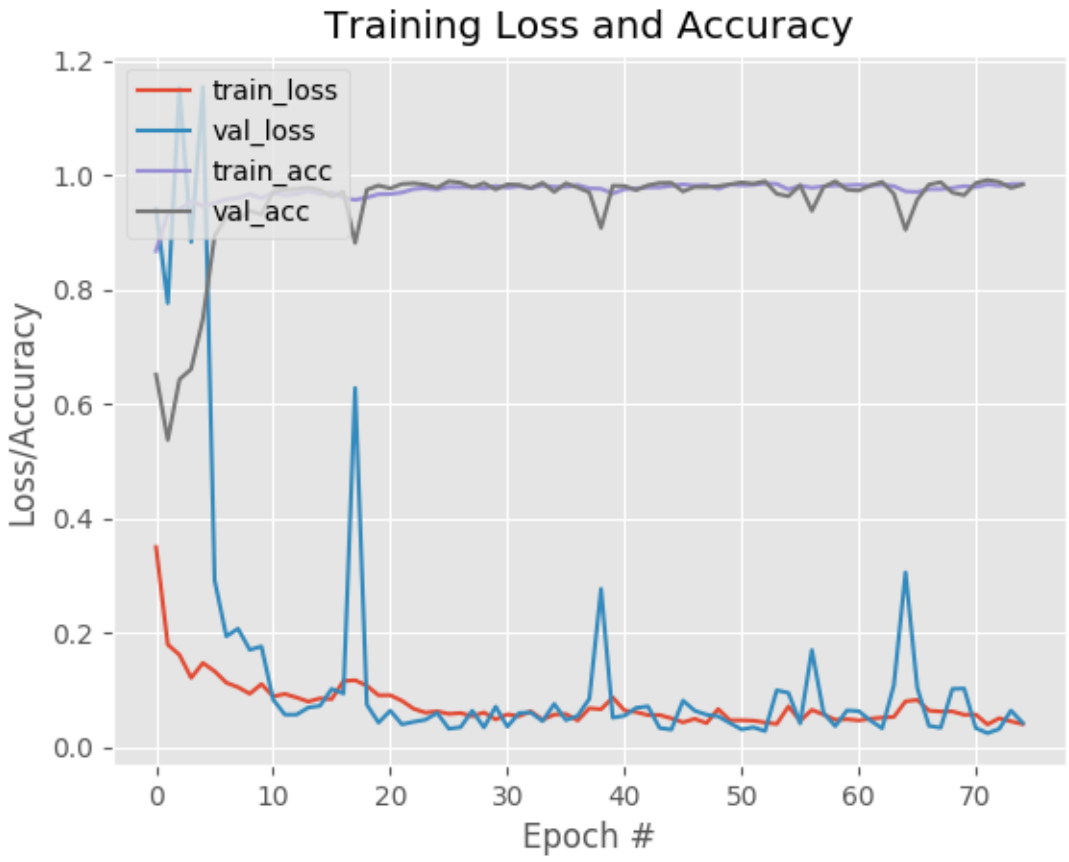
Accuracy + loss for training and validation is plotted . The plot is saved as an image file.

Training a Keras network for multi-label classification

We are going to train the model using command line arguments.

```
$ python train.py --dataset dataset --model fashion.model \
    --labelbin mlb.pickle
```

Training Loss and Accuracy Plot



Applying Keras multi-label classification to new images

Now that our multi-label classification keras model is trained, let's apply it to images outside of our testing set.

When you're ready, open create a new file in the project directory named classify.py and insert the following code.

```
# import the necessary packages
from keras.preprocessing.image import img_to_array
from keras.models import load_model
import numpy as np
import argparse
import imutils
import pickle
import cv2
import os
# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-m", "--model", required=True,
    help="path to trained model model")
ap.add_argument("-l", "--labelbin", required=True,
    help="path to label binarizer")
ap.add_argument("-i", "--image", required=True,
    help="path to input image")
args = vars(ap.parse_args())
```

We import the necessary packages for this script. Notably, we will be using keras and OpenCVin this script.

From there, we load and preprocess the input image:

```
# load the image
image = cv2.imread(args["image"])
output = imutils.resize(image, width=400)

# pre-process the image for classification
image = cv2.resize(image, (96, 96))
image = image.astype("float") / 255.0
image = img_to_array(image)
image = np.expand_dims(image, axis=0)
```

We take care to process the image in the same manner as we preprocessed our training data.

Next, let's load the model + multi – label binarizer and classify the image :

```
# load the trained convolutional neural network and the multi-label
# binarizer
print("[INFO] loading network...")
model = load_model(args["model"])
mlb = pickle.loads(open(args["labelbin"], "rb").read())

# classify the input image then find the indexes of the two class
# labels with the *largest* probability
print("[INFO] classifying image...")
proba = model.predict(image)[0]
idxs = np.argsort(proba)[::-1][:2]
```

We load the model and multi label binarizer from disk into memory.

From there we classify the (preprocessed) input image (Line 40) and extract the top two class labels indices (Line 41) by:

- Sorting the array indexes by their associated probability in descending

order

- Grabbing the first two class label indices which are thus the top-2 predictions from our network

You can modify this code to return more class labels if you wish. I would also suggest thresholding the probabilities and only returning labels with > *N%* confidence.

From there, we'll prepare the class labels + associated confidence values for overlay on the output image:

```
# loop over the indexes of the high confidence class labels
for (i, j) in enumerate(idxs):
    # build the label and draw the label on the image
    label = "{}: {:.2f}%".format(mlb.classes_[j], proba[j] * 100)
    cv2.putText(output, label, (10, (i * 30) + 25),
        cv2.FONT_HERSHEY_SIMPLEX, 0.7, (0, 255, 0), 2)

# show the probabilities for each of the individual labels
for (label, p) in zip(mlb.classes_, proba):
    print("{}: {:.2f}%".format(label, p * 100))

# show the output image
cv2.imshow("Output", output)
cv2.waitKey(0)
```

The loop draws the top two multi-label predictions and corresponding confidence values on the output image.

Similarly, the code prints the all the predictions in the terminal. This is useful for debugging purposes.

Finally, we show the output image on the screen .

Keras multi – label classification results

Let's put classify.py to work using command line arguments. You do not need to modify the code discussed above in order to pass new images through the CNN.

Let's try an image of a red dress — notice the three command line arguments that are processed at runtime :

```
$ python classify.py --model fashion.model --labelbin mlb.pickle \
    --image examples/example_01.jpg
```

Output and Results

lets try an image of an image of a red shirt

```
$ python classify.py --model fashion.model --labelbin mlb.pickle \
    --image examples/example_03.jpg
```

Output

Lets try black jeans :

```
$ python classify.py --model fashion.model --labelbin mlb.pickle \
    --image examples/example_06.jpg
```

OUTPUT

Conclusion

In this project, we learned how to perform multi-label classification with keras.

Performing multi-label classification with keras is straight forward and includes two primary steps :

1. Replace the softmax activation at the end of your network with a sigmaoid activation

2. Swap out categorical cross-entropy for binary cross-entropy for your loss function.

The end result of applying the process above is amulti-class classifier.

You can use our keras multi-class classifier to predict multiple labels with just a single forward pass.

However, there is a difficulty you need to consider:

You need training data for *each combination* of categories you would like to predict.

Just like a neural network cannot predict classes it was never trained on, your neural network cannot predict multiple class labels for combinations it has never seen. The reason for this behavior is due to activations of neurons inside the network.

If your network is trained on examples of both (1) black pants and (2) red shirts and now you want to predict "red pants" (where there are no *"red pants"* images in your dataset), the neurons responsible for detecting *"red"* and *"pants"* will fire, but since the network has never seen this combination of data/activations before once they reach the fully-connected layers, your output predictions will very likely be incorrect (i.e., you may encounter *"red"* or *"pants"* but very unlikely both).

Again, your network cannot correctly make predictions on data it was never trained on (and you shouldn't expect it to either). Keep this caveat in mind when training your own Keras networks for multi-label classification.

Problem Statement

Irrespective of what our objective is, while working on any project, we should be able to provide a fully legitimate final "soft copy" of a project.This ensures that every single part of the project is thought through to completion and no gaps are left logistically. This particular project provides a complexity in input data and integration between our trained model and simulator.

We collected data on a bright beautiful day, what if it's dark or raining when our autonomous car runs on a street, that wouldn't be good because our car can be confused.

While designing our model, we should make sure that our trained model is suitable to integrate with Udacity self driving car simulator.

LITERATURE OVERVIEW

The Literature Review is split into the six components:

They are

(1) Deep Learning
(2) Computer Vision
(3) Python
(4) keras
(5) End-to-End Deep Learning for Self-Driving Cars
(6) Udacity Self Driving Car Simulator

Deep Learning

Deep learning is a subset of a subset of the way that Machine's think. If we want to know about Deep learning, we need to have a basic idea on Artificial Intelligence and Machine Learning. So, what is AI and Machine Learning.

Artificial intelligence

Artificial intelligence is the broadest term we use for technology that allows machines to mimic human behavior. AI can include logic, if-then rules, and more.

Machine Learning

One subset of AI technology is machine learning. Machine learning allows computers to use statistics and use the experience to improve at tasks.

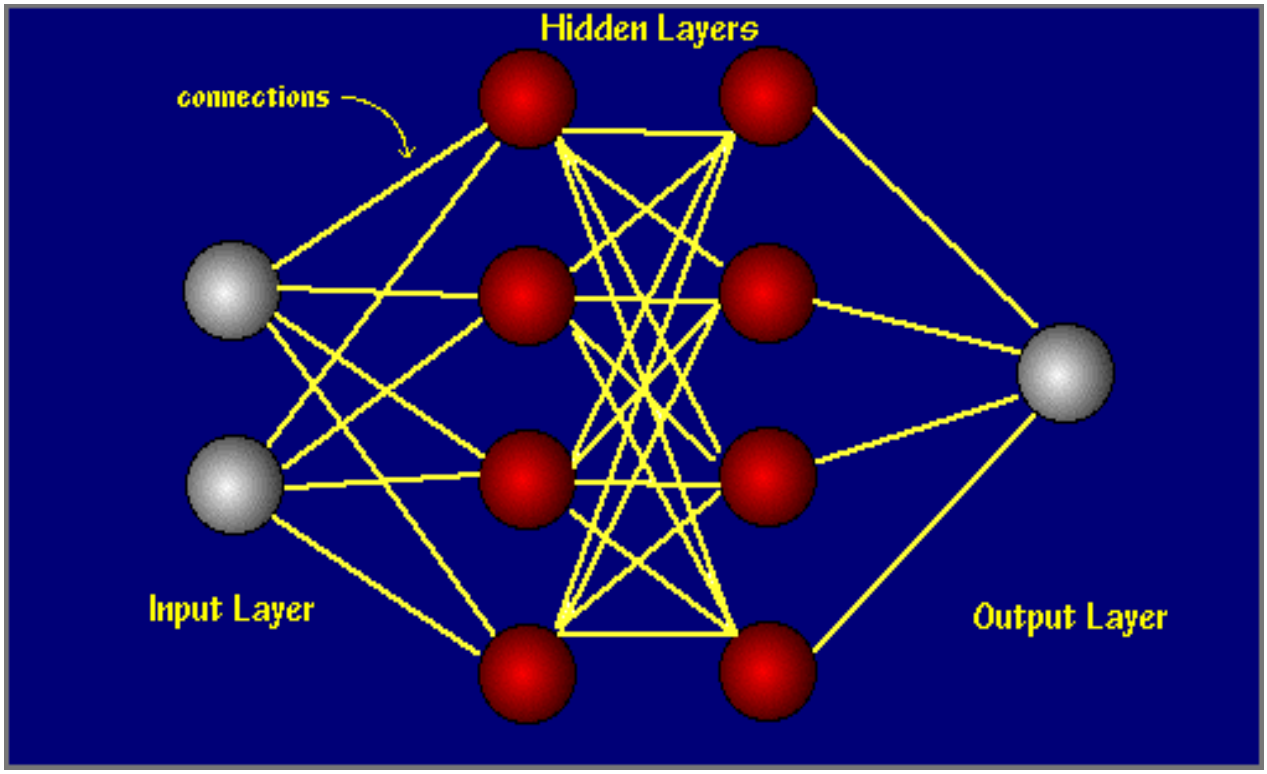
Deep Learning

A subset of machine learning is deep learning, which uses algorithms to train computers to perform tasks by exposing neural nets to huge amounts of data, allowing them to predict responses without needing to actually complete every task.

Neural Networks

Neural networks are typically organized in layers. Layers are made up of a number of interconnected 'nodes' which contain an 'activation function'. Patterns are presented to the network via the 'input layer',

which communicates to one or more 'hidden layers' where the actual processing is done via a system of weighted 'connections'. The hidden layers then link to an 'output layer' where the answer is output as shown in the graphic below.

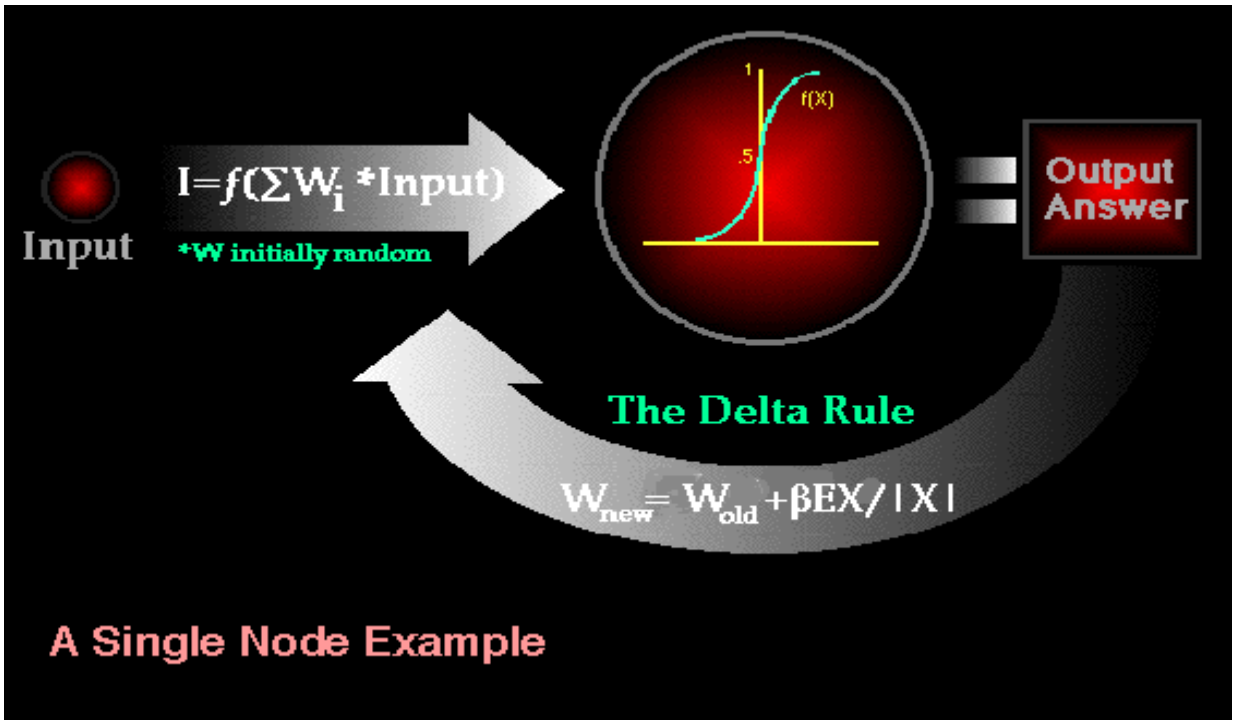


source : http://pages.cs.wisc.edu/~bolo/shipyard/neural/local.html

Most ANNs[1] contain some form of 'learning rule' which modifies the weights of the connections according to the input patterns that it is presented with. In a sense, ANNs learn by example as do their biological counterparts; a child learns to recognize dogs from examples of dogs. Although there are many different kinds of learning rules used by neural networks, this demonstration is concerned only with one; the delta rule. The delta rule is often utilized by the most common class of ANNs called BPNNs[2]. Back propagation is an abbreviation for the backwards propagation of error. With the delta rule, as with other types of back propagation, 'learning' is a supervised process that occurs with each

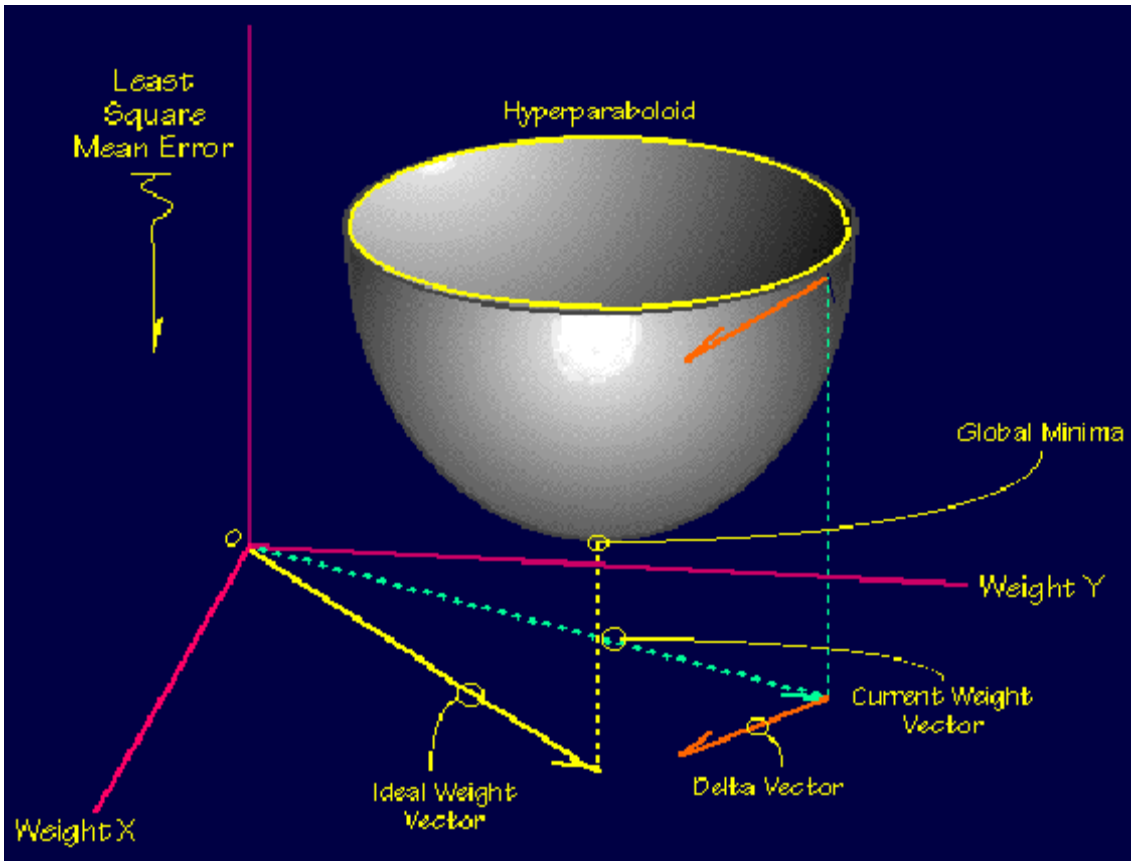
1 ANNs – Artificial Neural Network
2 BPNNs-Backpropagation Neural Network

cycle or 'epoch' (i.e. each time the network is presented with a new input pattern) through a forward activation flow of outputs, and the backwards error propagation of weight adjustments. More simply, when a neural network is initially presented with a pattern it makes a random 'guess' as to what it might be. It then sees how far its answer was from the actual one and makes an appropriate adjustment to its connection weights. More graphically, the process looks something like figure below:



source :http://pages.cs.wisc.edu/~bolo/shipyard/neural/local.html

Note also, that within each hidden layer node is a sigmoidal activation function which polarizes network activity and helps it to stablize. Back propagation performs a gradient descent within the solution's vector space towards a 'global minimum' along the steepest vector of the error surface. The global minimum is that theoretical solution with the lowest possible error. The error surface itself is a hyper paraboloid but is seldom 'smooth' as is depicted in the graphic below. Indeed, in most problems, the solution space is quite irregular with numerous 'pits' and 'hills' which may cause the network to settle down in a 'local minum' which is not the the best overall solution, which is in the below figure.



source : http://pages.cs.wisc.edu/~bolo/shipyard/neural/local.html

Since the nature of the error space can not be known a prioi, neural network analysis often requires a large number of individual runs to determine the best solution. Most learning rules have built-in mathematical terms to assist in this process which control the 'speed' (Beta-coefficient) and the 'momentum' of the learning. The speed of learning is actually the rate of convergence between the current solution and the global minimum. Momentum helps the network to overcome obstacles (local minima) in the error surface and settle down at or near the global minimum.

Once a neural network is 'trained' to a satisfactory level it may be used as an analytical tool on other data. To do this, the user no longer specifies any training runs and instead allows the network to work in forward propagation mode only. New inputs are presented to the input pattern where they filter into and are processed by the middle layers as though training were taking place, however, at this point the output is

retained and no back propagation occurs. The output of a forward propagation run is the predicted model for the data which can then be used for further analysis and interpretation. It is also possible to over- train a neural network, which means that the network has been trained exactly to respond to only one type of input; which is much like rote memorization. If this should happen then learning can no longer occur and the network is referred to as having been "grand mothered" in neural network jargon. In real world applications this situation is not very useful since one would need a separate grand mothered network for each new kind of input. We can find more information in this page [4].

Computer Vision

The computer vision part of the project is accomplished using OpenCV. This is a library which contains functions that are used to achieve real- time computer vision. It was released by Intel in 1999 to simplify computations that were too intensive for a computer. It started off with the simple applications of ray tracing and 3D display walls.
Documentation for OpenCV3, which is what we
will be using in this project is available in the respective page. The main focus and use of computer vision in this project to rotate, translate, flip, shift, transform to darker or to brighter. To achieve this, inspiration was taken from this post[10] by Vivek Yadav.

Python

Installation of Python is required for this project. Python is a programming language that is dynamic and object oriented. It emphasises on readability of the code and thereby, features indentation to delimit blocks of code. Python can be downloaded from official python website [5]. Version 3.5.2 has been used in our project.

[10] augmentation -https://chatbotslife.com/using-augmentation-to-mimic-human-driving-496b569760a9

Keras

In this project, we are going to use keras deep learning library to create a model. Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTN or Theano. It was developed with a focus on enabling fast experimentation. Being able to go from idea to result with the least possible delay is key to doing good research. We can find more information in keras documentation

End-to-End Deep Learning for Self-Driving Car

In a new automotive application, NVIDIA have used convolutional neural networks (CNNs) to map the raw pixels from a front-facing camera to the steering commands for a self-driving car. This powerful end-to-end approach means that with minimum training data from humans, the system learns to steer, with or without lane markings, on both local roads and highways. The system can also operate in areas with unclear visual guidance such as parking lots or unpaved roads.

Figure 1 : NVIDIA Self Driving Car in action

3 keras Documentation - https://keras.io/

They designed the end-to-end learning system using an NVIDIA DevBox running Torch 7 for training. An NVIDIA DRIVE^{TM} PX self-driving car computer, also with Torch 7, was used to determine where to drive— while operating at 30 frames per second (FPS). The system is trained to automatically learn the internal representations of necessary processing steps, such as detecting useful road features, with only the human steering angle as the training signal. They never explicitly trained it to detect, for example, the outline of roads. In contrast to methods using explicit decomposition of the problem, such as lane marking detection, path planning, and control, our end-to-end system optimizes all processing steps simultaneously.

They believe that end-to-end learning leads to better performance and smaller systems. Better performance results because the internal components self-optimize to maximize overall system performance, instead of optimizing human-selected intermediate criteria, e. g., lane detection. Such criteria understandably are selected for ease of human interpretation which doesn't automatically guarantee maximum system performance. Smaller networks are possible because the system learns to solve the problem with the minimal number of processing steps.

Convolutional Neural Networks to Process Visual Data

CNNs have revolutionized the computational pattern recognition process [6]. Prior to the widespread adoption of CNNs, most pattern recognition tasks were performed using an initial stage of hand-crafted feature extraction followed by a classifier. The important breakthrough of CNNs is that features are now learned automatically from training examples. The CNN approach is especially powerful when applied to image recognition tasks because the convolution operation captures the 2D nature of images. By using the convolution kernels to scan an entire image, relatively few parameters need to be learned compared to the total number of operations.

4 CNN – Convolutional Neural Network

While CNNs with learned features have been used commercially for over twenty years [7],their adoption has exploded in recent years because of two important developments. First, large, labeled data sets such as the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [8] are now widely available for training and validation. Second, CNN learning algorithms are now implemented on massively parallel graphics processing units (GPUs), tremendously accelerating learning and inference ability.

The CNNs that we describe here go beyond basic pattern recognition. We developed a system that learns the entire processing pipeline needed to steer an automobile. The groundwork for this project was actually done over 10 years ago in a Defense Advanced Research Projects Agency (DARPA) seedling project known as DARPA Autonomous Vehicle (DAVE) [9], in which a sub-scale radio control (RC) car drove through a junk-filled alley way. DAVE[6] was trained on hours of human driving in similar, but not identical, environments. The training data included video from two cameras and the steering commands sent by a human operator.

In many ways, DAVE was inspired by the pioneering work of Pomerleau[10], who in 1989 built the Autonomous Land Vehicle in a Neural Network (ALVINN) system. ALVINN[7] is a precursor to DAVE, and it provided the initial proof of concept that an end-to-end trained neural network might one day be capable of steering a car on public roads. DAVE demonstrated the potential of end-to-end learning, and indeed was used to justify starting the DARPA Learning Applied to Ground Robots (LAGR) program[11], but DAVE's performance was not sufficiently reliable to provide a full alternative to the more modular approaches to off-road driving. (DAVE's mean distance between crashes was about 20 meters in complex environments.)

About a year ago we started a new effort to improve on the original DAVE, and create a robust system for driving on public roads.

---

5 DAVE – DAPRA Autonomous Vehicle

6 ALVINN – Autonomous Land Vehicle in a Neural Network

The primary motivation for this work is to avoid the need to recognize specific human-designated features, such as lane markings, guard rails, or other cars, and to avoid having to create a collection of "if, then, else" rules, based on observation of these features. We are excited to share the preliminary results of this new effort, which is aptly named: DAVE–2.

The DAVE-2 System

Figure 2 shows a simplified block diagram of the collection system for training data of DAVE-2. Three cameras are mounted behind the windshield of the data-acquisition car, and time stamped video from the cameras is captured simultaneously with the steering angle applied by the human driver. The steering command is obtained by tapping into the vehicle's Controller Area Network (CAN) bus. In order to make our system independent of the car geometry, we represent the steering command as $1/r$, where r is the turning radius in meters. We use $1/r$ instead of r to prevent a singularity when driving straight (the turning radius for driving straight is infinity). $1/r$ smoothly transitions through zero from left turns (negative values) to right turns (positive values).

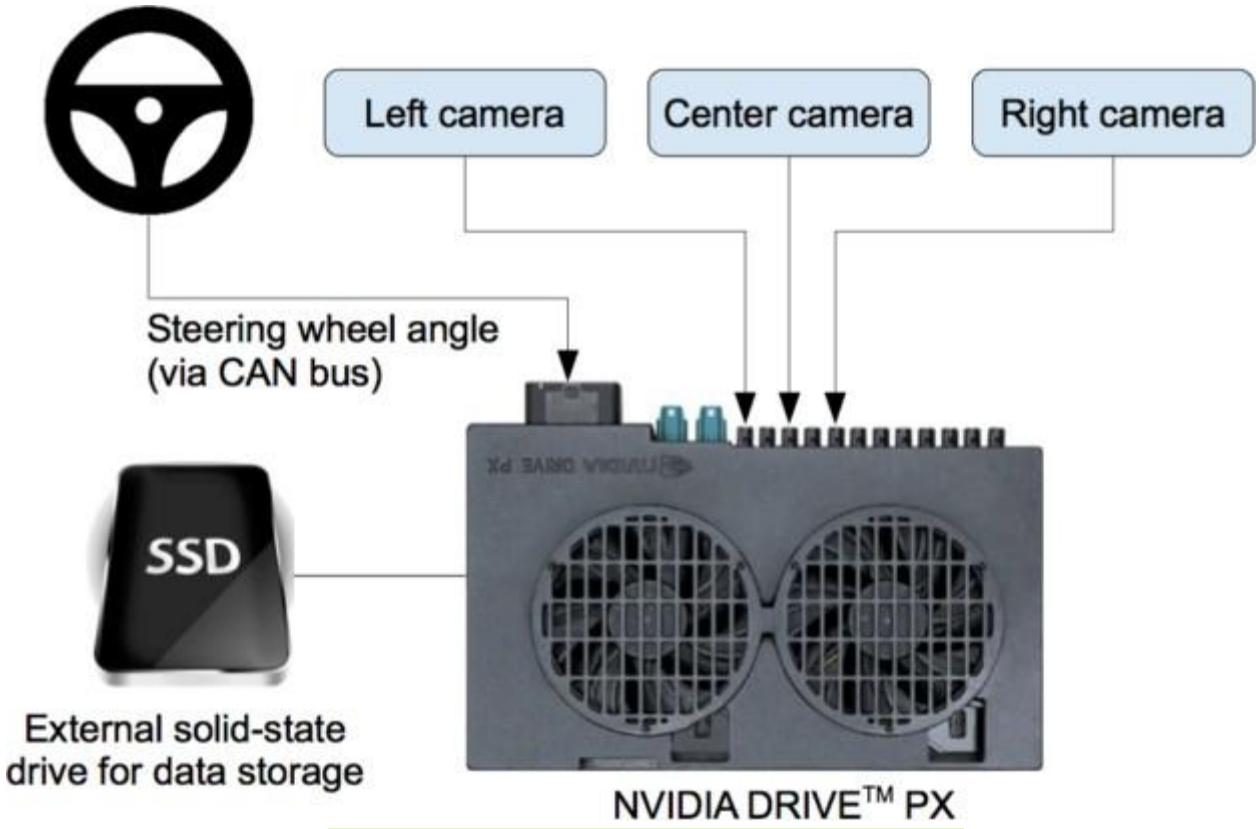


Figure 2: High-level view of the data collection system

Training data contains single images sampled from the video, paired with the corresponding steering command (1/r). Training with data from only the human driver is not sufficient; the network must also learn how to recover from any mistakes, or the car will slowly drift off the road. The training data is therefore augmented with additional images that show the car in different shifts from the center of the lane and rotations from the direction of the road. The images for two specific off-center shifts can be obtained from the left and the right cameras. Additional shifts between the cameras and all rotations are simulated through viewpoint transformation of the image from the nearest camera. Precise viewpoint transformation requires 3D scene knowledge which we don't have, so we approximate the transformation by assuming all points below the horizon are on flat ground, and all points above the horizon are infinitely far away.

This works fine for flat for terrain, but for more complete rendering to introduces distortions for objects that stick above the ground, such as cars, poles, trees, and buildings. Fortunately these distortions don't pose a significant problem for network training. The steering label for the transformed images is quickly adjusted to one that correctly steers the vehicle back to the desired location and orientation in two seconds. Figure 3 shows a block diagram of our training system. Images are fed into a CNN that then computes a proposed steering command. The proposed command is compared to the desired command for that image, and the weights of the CNN are adjusted to bring the CNN output closer to the desired output. The weight adjustment is accomplished using back propagation as implemented in the Torch 7 machine learning package.

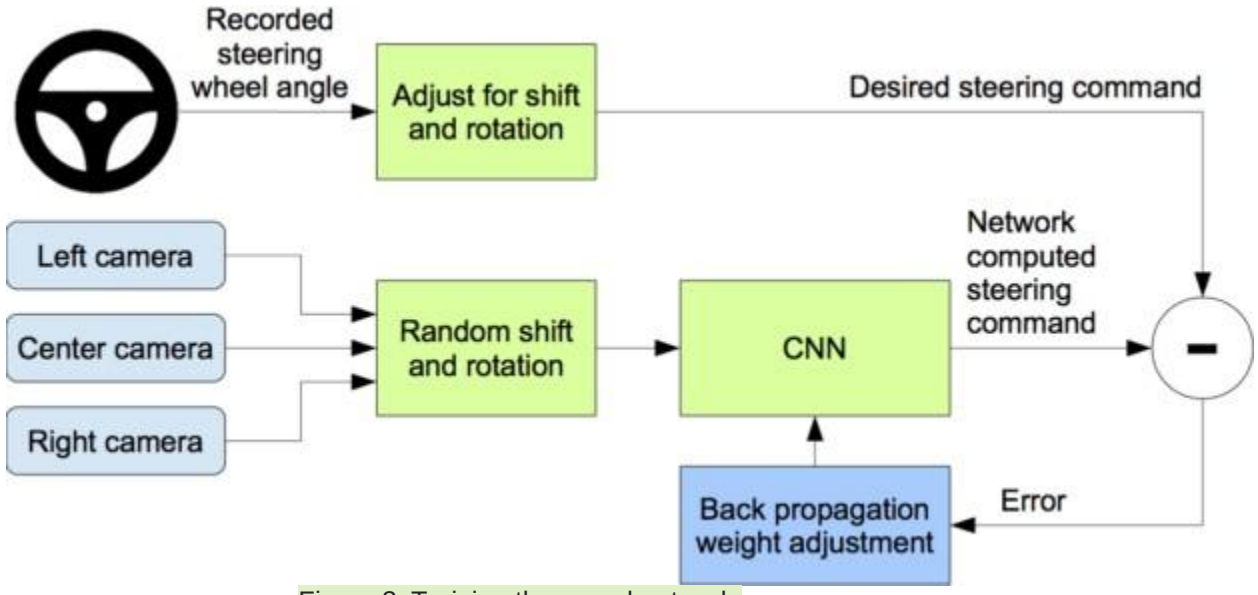


Figure 3: Training the neural network.

Once trained, the network is able to generate steering commands from the video images of a single center camera. Figure 4 shows this configuration.

Figure 4: The trained network is used to generate steering commands from a single front-facing center camera

Data Collection

Training data was collected by driving on a wide variety of roads and in a diverse set of lighting and weather conditions. They gathered surface street data in central New Jersey and highway data from Illinois, Michigan, Pennsylvania, and New York. Other road types include two- lane roads (with and without lane markings), residential roads with parked cars, tunnels, and unpaved roads. Data was collected in clear, cloudy, foggy, snowy, and rainy weather, both day and night. In some instances, the sun was low in the sky, resulting in glare reflecting from the road surface and scattering from the windshield.

The data was acquired using either our drive-by-wire test vehicle, which is a 2016 Lincoln MKZ, or using a 2013 Ford Focus with cameras placed in similar positions to those in the Lincoln. System has no dependencies on any particular vehicle make or model. Drivers were encouraged to maintain full attentiveness, but otherwise drive as they usually do. As of March 28, 2016, about 72 hours of driving data was collected.

Network Architecture

They train the weights of our network to minimize the mean-squared error between the steering command output by the network, and either the command of the human driver or the adjusted steering command for off-center and rotated images (see “Augmentation”, later). Figure 5 shows the network architecture, which consists of 9 layers, including a normalization layer, 5 convolutional layers, and 3 fully connected layers.

The input image is split into YUV planes and passed to the network.

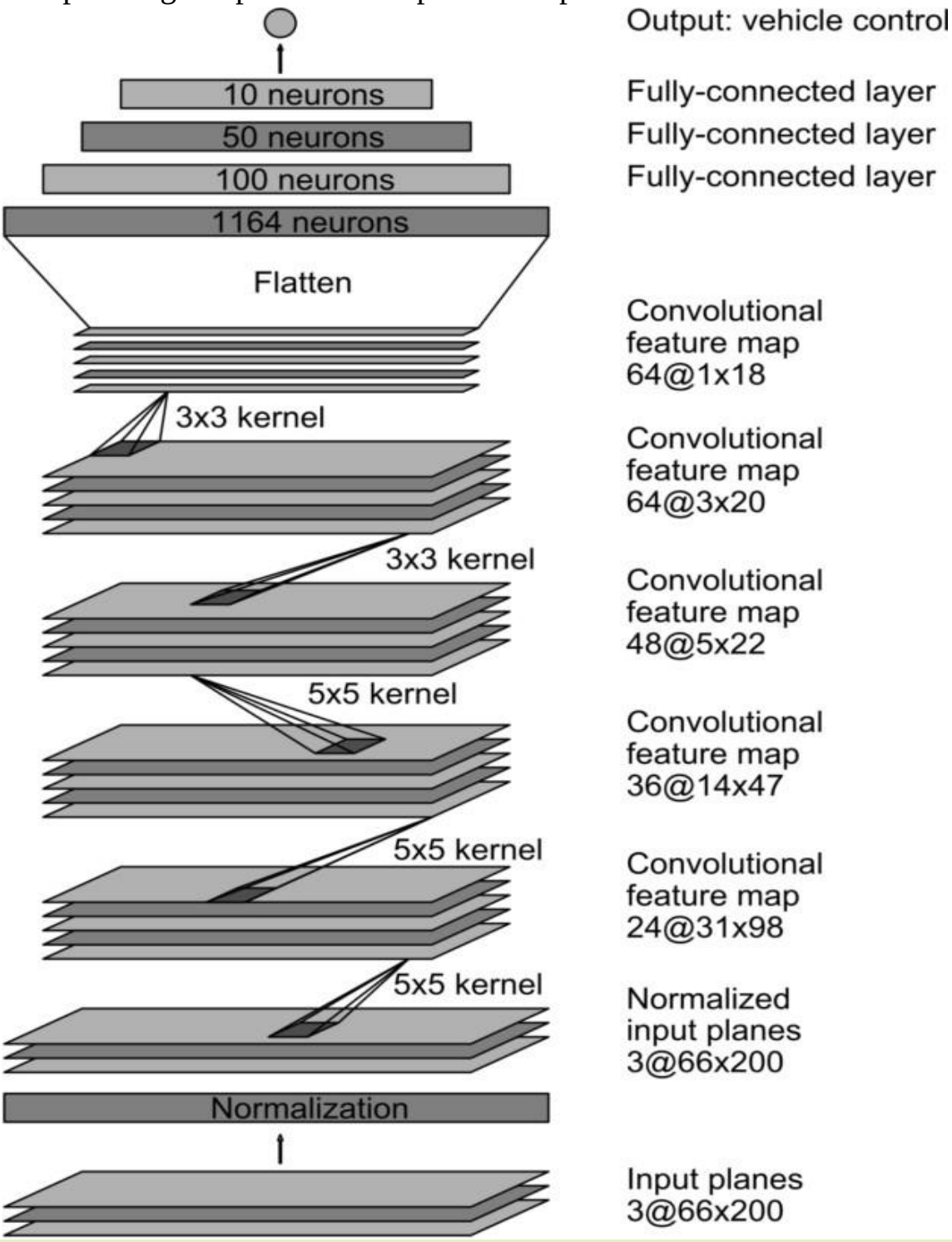


Figure 5: CNN architecture. The network has about 27 million connections and 250 thousand parameters

The first layer of the network performs image normalization. The normalizer is hard-coded and is not adjusted in the learning process.

Performing normalization in the network allows the normalization scheme to be altered with the network architecture, and to be accelerated via GPU processing.
The convolutional layers are designed to perform feature extraction, and are chosen empirically through a series of experiments that vary layer configurations. They then use strided convolutions in the first three convolutional layers with a 2×2 stride and a 5×5 kernel, and a non-strided convolution with a 3×3 kernel size in the final two convolutional layers.
They follow the five convolutional layers with three fully connected layers, leading to a final output control value which is the inverse- turning-radius. The fully connected layers are designed to function as a controller for steering, but we noted that by training the system end-to- end, it is not possible to make a clean break between which parts of the network function primarily as feature extractor, and which serve as controller.

Data Selection

The first step to training a neural network is selecting the frames to use. Collected data is labeled with road type, weather condition, and the driver's activity (staying in a lane, switching lanes, turning, and so forth). To train a CNN to do lane following, simply select data where the driver is staying in a lane, and discard the rest. They then sample that video at 10 FPS because a higher sampling rate would include images that are highly similar, and thus not provide much additional useful information. To remove a bias towards driving straight the training data includes a higher proportion of frames that represent road curves.

Augmentation

After selecting the final set of frames, They augment the data by adding artificial shifts and rotations to teach the network how to recover from

a poor position or orientation. The magnitude of these perturbations is chosen randomly from a normal distribution. The distribution has zero mean, and the standard deviation is twice the standard deviation that we measured with human drivers. Artificially augmenting the data does add undesirable artifacts as the magnitude increases (as mentioned previously).

Simulation

Before road-testing a trained CNN, They first evaluate the network's performance in simulation. Figure 6 shows a simplified block diagram of the simulation system, and Figure 7 shows a screenshot of the simulator in interactive mode.

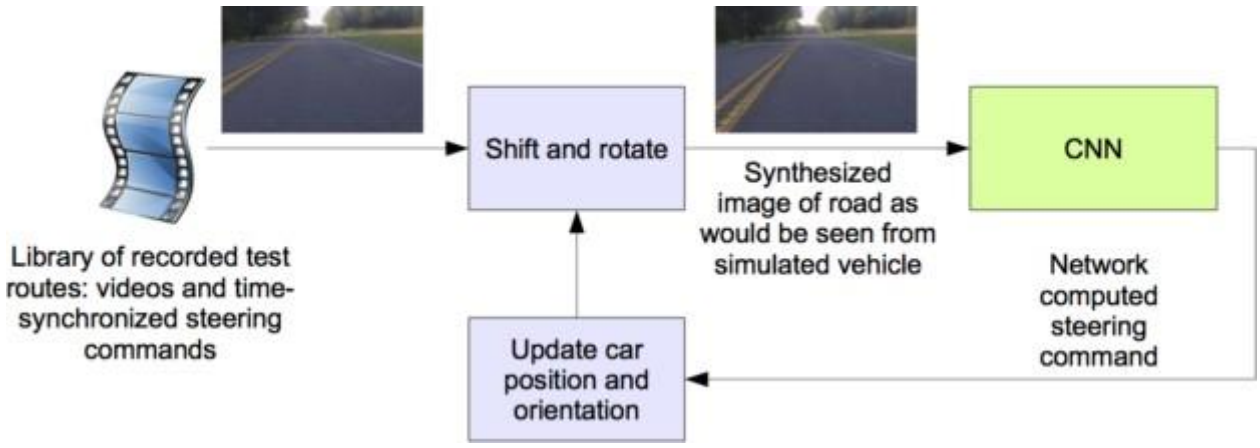


figure 6 : Block Diagram

The simulator takes prerecorded videos from a forward-facing on-board camera connected to a human-driven data-collection vehicle, and generates images that approximate what would appear if the CNN were instead steering the vehicle. These test videos are time-synchronized with the recorded steering commands generated by the human driver.

Since human drivers don't drive in the center of the lane all the time, they must manually calibrate the lane's center as it is associated with each frame in the video used by the simulator. They call this position the "ground truth".

The simulator transforms the original images to account for departures from the ground truth. Note that this transformation also includes any discrepancy between the human driven path and the ground truth. The transformation is accomplished by the same methods as described previously.

The simulator accesses the recorded test video along with the synchronized steering commands that occurred when the video was captured. The simulator sends the first frame of the chosen test video, adjusted for any departures from the ground truth, to the input of the trained CNN, which then returns a steering command for that frame. The CNN steering commands as well as the recorded human-driver commands are fed into the dynamic model [12] of the vehicle to update the position and orientation of the simulated vehicle.

Figure 7: Screenshot of the simulator in interactive mode. See text for explanation of the performance metrics. The green area on the left is unknown because of the viewpoint transformation. The highlighted wide rectangle below the horizon is the area which is sent to the CNN.

The simulator then modifies the next frame in the test video so that the image appears as if the vehicle were at the position that resulted by following steering commands from the CNN. This new image is then fed to the CNN and the process repeats.

The simulator records the off-center distance (distance from the car to the lane center), the yaw, and the distance traveled by the virtual car. When the off-center distance exceeds one meter, a virtual human intervention is triggered, and the virtual vehicle position and orientation is reset to match the ground truth of the corresponding frame of the original test video.

They evaluated the networks in two steps: first in simulation, and then in on-road tests.
In simulation they have the networks provide steering commands in our simulator to an ensemble of prerecorded test routes that correspond to about a total of three hours and 100 miles of driving in Monmouth County, NJ. The test data was taken in diverse lighting and weather conditions and includes highways, local roads, and residential streets.
We estimate what percentage of the time the network could drive the car (autonomy) by counting the simulated human interventions that occur when the simulated vehicle departs from the center line by more than one meter. They assume that in real life an actual intervention would require a total of six seconds: this is the time required for a human to retake control of the vehicle, re-center it, and then restart the self-steering mode. We calculate the percentage autonomy by counting the number of interventions, multiplying by 6 seconds, dividing by the elapsed time of the simulated test, and then subtracting the result from 1:

$$\text{autonomy} = (1 - \frac{\#\text{ of interventions} \cdot 6[\text{seconds}]}{\text{elapsed time}[\text{seconds}]}) \cdot 100$$

Thus, if we had 10 interventions in 600 seconds, we would have an autonomy value of

$$(1 - \frac{10 \cdot 6}{600}) \cdot 100 = 90\%$$

On-road Tests

After a trained network has demonstrated good performance in the simulator, the network is loaded on the DRIVE PX in our test car and taken out for a road test. For these tests they measure performance as the fraction of time during which the car performs autonomous steering. This time excludes lane changes and turns from one road to another. For a typical drive in Monmouth County NJ from our office in Holmdel to Atlantic Highlands, we are autonomous approximately 98% of the time. They also drove 10 miles on the Garden State Parkway (a multi-lane divided highway with on and off ramps) with zero intercepts. You can see video here [13]

Udacity Self Driving Car Simulator

Udacity recently made its self-driving car simulator source code available on their GitHub which was originally build to teach their self- Driving Car Engineer Nanodegree students. Now, anybody can take advantage of the useful tool to train your machine learning models to clone driving behavior.

We can manually drive a car to generate training data, or machine learning model can autonomously drive for testing.
In the main screen of the simulator, we can choose a scene and mode. Figure below is Simulator main screen.

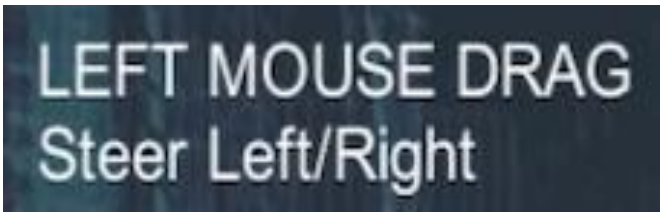
First, you choose a scene by clicking one of the scene pictures. In above, the lake side scene (the left) is selected. Next, we choose a mode: Training Mode or Autonomous Mode. As soon as you click one of the mode buttons, a car appears at the start position.

simulator train mode

Training Mode

In the training mode, you drive the car manually to record the driving behavior. You can use the recorded images to train your machine learning model. To drive the car, use the following keys:

If you like using mouse to direct the car, you can do so by dragging:



To start a recording your driving behavior, press R on your keyboard.

You can press R again to stop the recording.

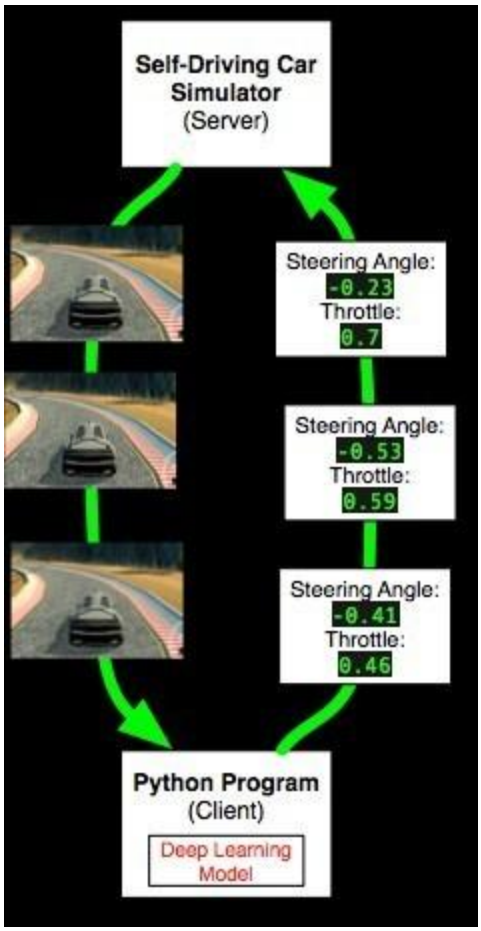
Finally, use ESC to exit the training mode.

You can see the driving instructions any time by clicking **CONTROLS** button in the top right of the main screen.

Autonomous Mode

In the autonomous mode, you are testing your machine learning model to see how well your model can drive the car without dropping off the road / falling into the lake.

Technically, the simulator is acting as a server where your program can connect to and receive a stream of image frames from.



Simulator (Server) <-> Model (Client)

For example, your Python program can use a machine learning model to process the road images to predict the best driving instructions, and send them back to the server.

Each driving instruction contains a steering angle and an acceleration throttle, which changes the car's direction and the speed (via acceleration). As this happens, your program will receive new image frames at real time.

SOFTWARE REQUIREMENTS

Python

Python is a widely used high-level, general-purpose,interpreted, dynamic programming language. Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than would be possible in languages such as C++ or Java. The language provides constructs intended to enable clear programs on both a small and large scale.

To install python using command line. In this project we should use python version 3.5.2 only. In order to change the version of python using command line,use

$ alias python3 = python 3.5.2

For more information, you can see this documentation [14].

b. Pandas

pandas is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language.

To install pandas using command line

$ pip3 install pandas

for more information, you can see this documentation [15].

d. Numpy

NumPy is the fundamental package for scientific computing with Python. It contains among other things:

• a powerful N-dimensional array object
• sophisticated (broadcasting) functions
• tools for integrating C/C++ and Fortran code

To install Numpy using command line

$ pip3 install numpy

for more information, you can see this documentation [16].

e. Matplotlib

Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. Matplotlib can be used in Python scripts, the Python and Ipython shells, the Jupyter notebook, web application servers, and four graphical user interface toolkits.

To install matplotlib using command line, use

```
$ pip3 install matplotlib
```

For more information, you can see this documentation [17].

f. Scikit - learn

Scikit - Learn It is used for machine learning in python. It is simple and efficient tools for data mining and data analysis. It is accessible to everybody, and reusable in various contexts. It can be built on NumPy, SciPy, and matplotlib

To install scikit – learn using command line, use

```
$ pip3 install scikit-learn
```

for more information, you can see this documentation [18].

g. OpenCV

OpenCV (Open Source Computer Vision Library) is released under a BSD license and hence it's free for both academic and commercial use. It has C++, Python and Java interfaces and supports Windows, Linux, Mac OS, iOS and Android. OpenCV was designed for computational efficiency and with a strong focus on real-time applications. Written in optimized C/C++, the library can take advantage of multi-core processing. Enabled with OpenCL, it can take advantage of the hardware acceleration of the underlying heterogeneous compute platform.

Adopted all around the world, OpenCV has more than 47 thousand people of user community and estimated number of downloads exceeding 14 million. Usage ranges from interactive art, to mines inspection, stitching maps on the web or through advanced robotics.

To install opencv using command line, use

```
$ sudo apt-get install python-opencv
```

For detailed installation steps, see this guide lines [19]
For more information, you can see this documentation [20].

h. Pillow

The **Python Imaging Library** adds image processing capabilities to your Python interpreter. This library provides extensive file format support, an efficient internal representation, and fairly powerful image processing capabilities. The core image library is designed for fast access to data stored in a few basic pixel formats. It should provide a solid foundation for a general image processing tool.

To install pillow using command line argument, use

```
$ pip3 install pillow
```

For more information, you can see this documentation [21].

f. Scikit-image

Scikit-image is a collection of algorithms for Image processing. It is available free of charge and free of restriction. It is high quality, per reviewed code, written by an active community of volunteers.
To install scikit-image using command line arguments, use

```
$ sudo apt-get install python-skimage
```

For more information, you can see this documentation [22].

g. Scipy

SciPy (pronounced "Sigh Pie") is a Python-based ecosystem of open source software for mathematics, science, and engineering. In particular, some of the core packages are Numpy, SciPy library, Matplotlib, IPython, Sympy, pandas.
To install scipy using command line arguments, use

```
$ sudo apt-get install python-numpy python-scipy python-matplotlib
ipython ipython-notebook python-pandas python-sympy python-nose
```

For more information, you can see this documentation [23].

h. h5py

The h5py package is a Pythonic interface to the HDF5 binary data format. It lets you store huge amounts of numerical data, and easily manipulate that data from NumPy. For example, you can slice into multi-terabyte datasets stored on disk, as if they were real NumPy arrays. Thousands of datasets can be stored in a single file, categorized and tagged however you want.

To install h5py using command line argument, use

```
$ pip3 install h5py
```

For more information, you can see this documentation [24].

Eventlet

Eventlet is a concurrent networking library for Python that allows you to change how you run your code, not how you write it. It uses epoll or kqueue or libevent for highly scalable non-blocking I/O. Coroutines ensure that the developer uses a blocking style of programming that is similar to threading, but provide the benefits of non-blocking I/O. The event dispatch is implicit, which means you can easily use Eventlet from the Python interpreter, or as a small part of a larger application.

To install Eventlet using command line arguments, use

```
$ pip3 install eventlet
```

For more information, you can see this documentation [25].

Flask-socketio

Flask-SocketIO gives Flask applications access to low latency bi-directional communications between the clients and the server. The client-side application can use any of the SocketIO official clients libraries in Javascript, C++, Java and Swift, or any compatible client to establish a permanent connection to the server.

To install it using command line arguments, use

```
$ pip3 install flask-socketio
```

For more information, you can see this documentation [26].

g. Seaborn

Seaborn is a Python visualization library based on matplotlib. It provides a high-level interface for drawing attractive statistical graphic.

To install it using command line arguments, use

```
$ pip3 install seaborn
```

For more information, you can see this documentation [27].

g. Imageio

Imageio is a Python library that provides an easy interface to read and write a wide range of image data, including animated images, volumetric data, and scientific formats. It is cross-platform, runs on Python 2.7 and 3.4+, and is easy to install.

To install it using command line arguments, use

```
$ pip3 install imageio
```

For more information, you can see this documentation [28].

f. Moviepy

MoviePy is a Python library for video editing: cutting, concatenations, title insertions, video compositing (a.k.a. non-linear editing), video processing, and creation of custom effects. MoviePy can read and write all the most common audio and video formats, including GIF, and runs on Windows/Mac/Linux, with Python 2.7+ and 3.

To install it using command line arguments, use

```
$ pip3 install moviepy
```

For more information, you can see this documentation [29].

Tensorflow

TensorFlow is an open source software library for high performance numerical computation. Its flexible architecture allows easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices. Originally developed by researchers and engineers from the Google Brain team within Google's AI organization[30], it comes with strong support for machine learning and deep learning and the flexible numerical computation core is used across many other scientific domains.

To install it using command line arguments, use

```
$ pip3 install tensorflow-gpu
```

For complete installation guide, follow the steps in this website[8]. For more information, you can see this documentation[31]

Keras

Keras is a high-level neural networks API, written in Python and capable of running on top of Tensor Flow, CNTN or Theano. It was developed with a focus on enabling fast experimentation. Being able to go from idea to result with the least possible delay is key to doing good research. Keras allows us : For easy and fast prototyping (through user friendliness, modularity, and extensibility).

7 Tensor flow installation - https://www.tensorflow.org/tutorials/

Supports both convolutional networks and recurrent networks, as well as combinations of the two. To Runs seamlessly on CPU and GPU.

To install it using command line arguments, use

```
$ sudo pip3 install keras
```

For more information, you can see this documentation [32].

Udacity Self Driving Car Simulator

This simulator was built for Udacity's Self-Driving Car Nanodegree, to teach students how to train cars how to navigate road courses using deep learning. It is very easy to install. Please go to this page, follow the instructions and download these files.

8 Udacity Simulator Installation - https://github.com/udacity/self-driving-car-sim
9 Download Files for simulator - https://github.com/udacity/self-driving-car-sim.git

Methodology and Approach

Data Generation

As an input, we used Udacity Self Driving Car Simulator to generate input data. We use keyboard to drive around the circuit. The simulator records screen shots of the images and the decisions we made: steering angles, throttle and brake, while driving the car in simulator.

The data comprises of images as seen by the car at each point of time while navigating around the track, along with a CSV file, holding the entire data of the car (speed, break, throttle, images) for the entire navigation period. So the columns in the CSV are:

Centre camera image path	Left camera image path	Right camera image path	Steering angle	Speed	Throttle	Break

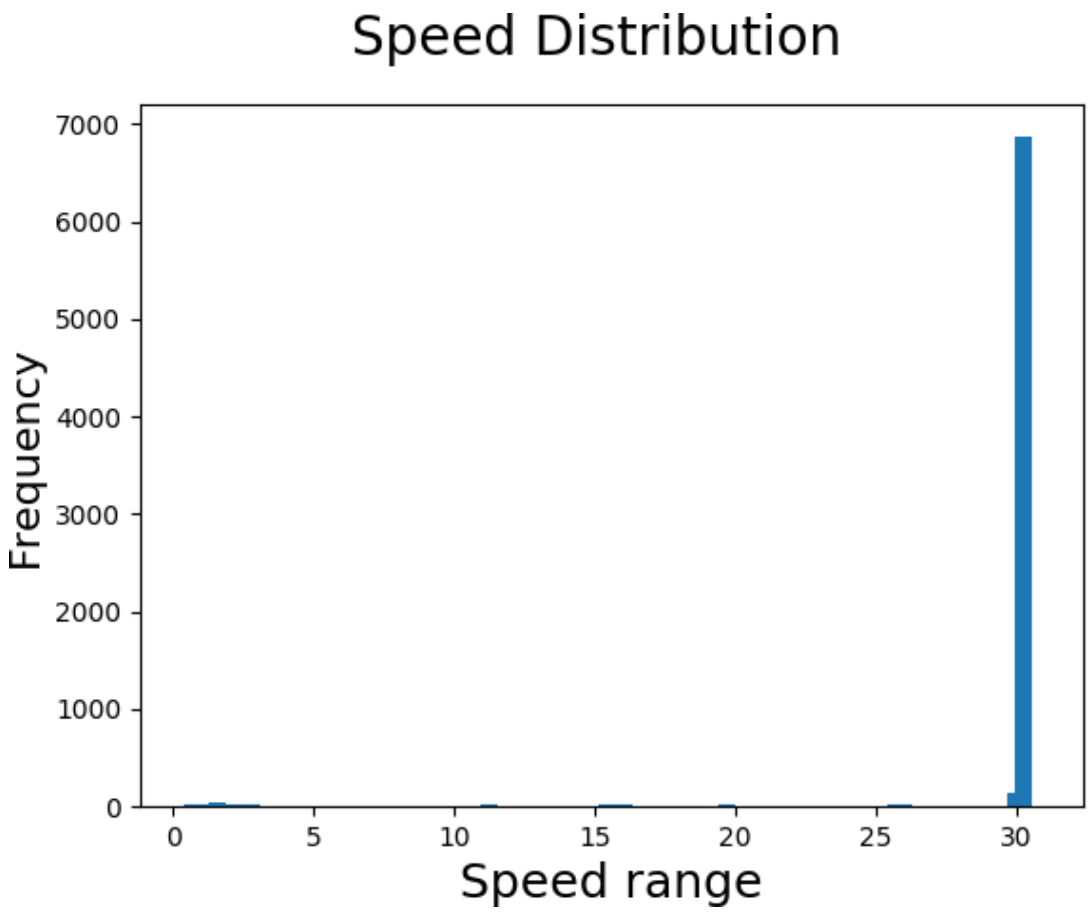
First three columns are just image paths to the images as seen by the car during navigation from three cameras.

Steering angle is the values of steering wheel, lies between -1 and 1.
Speed the speed of the car for that moment.

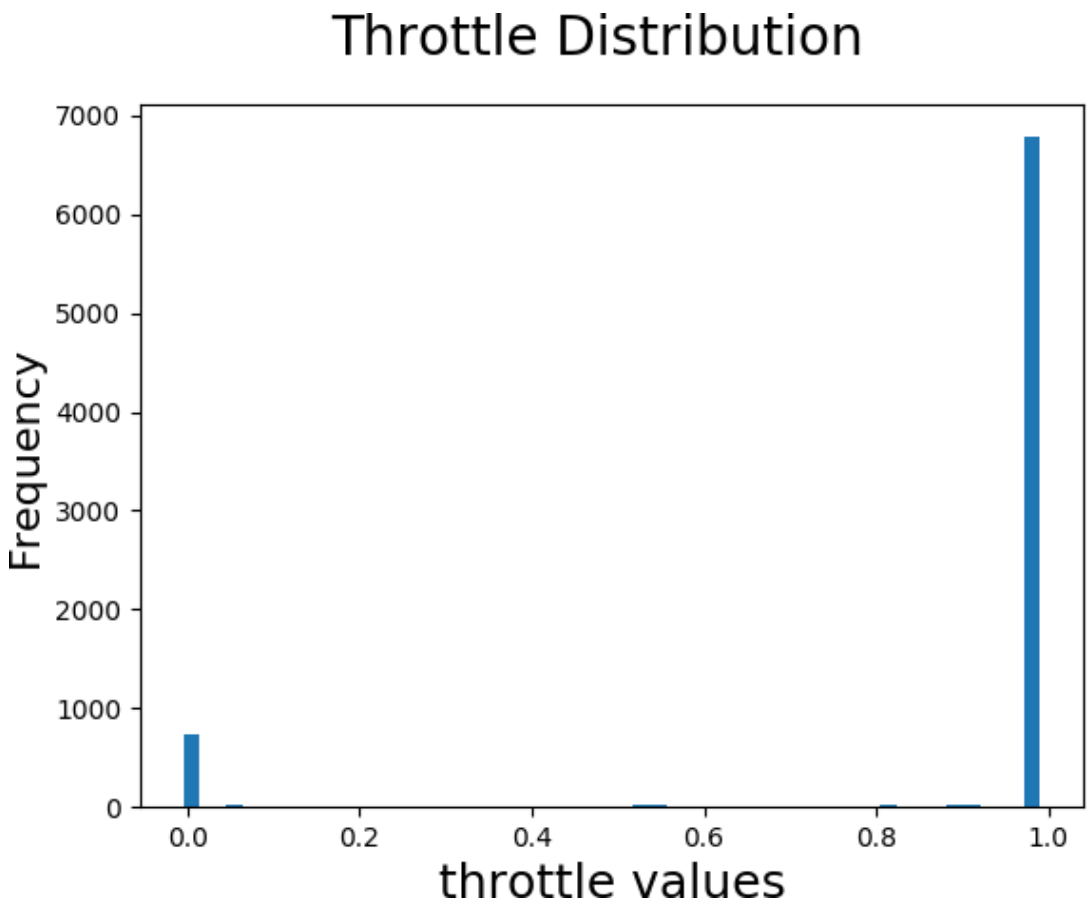
Data Distribution

By now, we are all too familiar with the tricks data can play. So, let's peek inside a bit more as to how our data looks like.

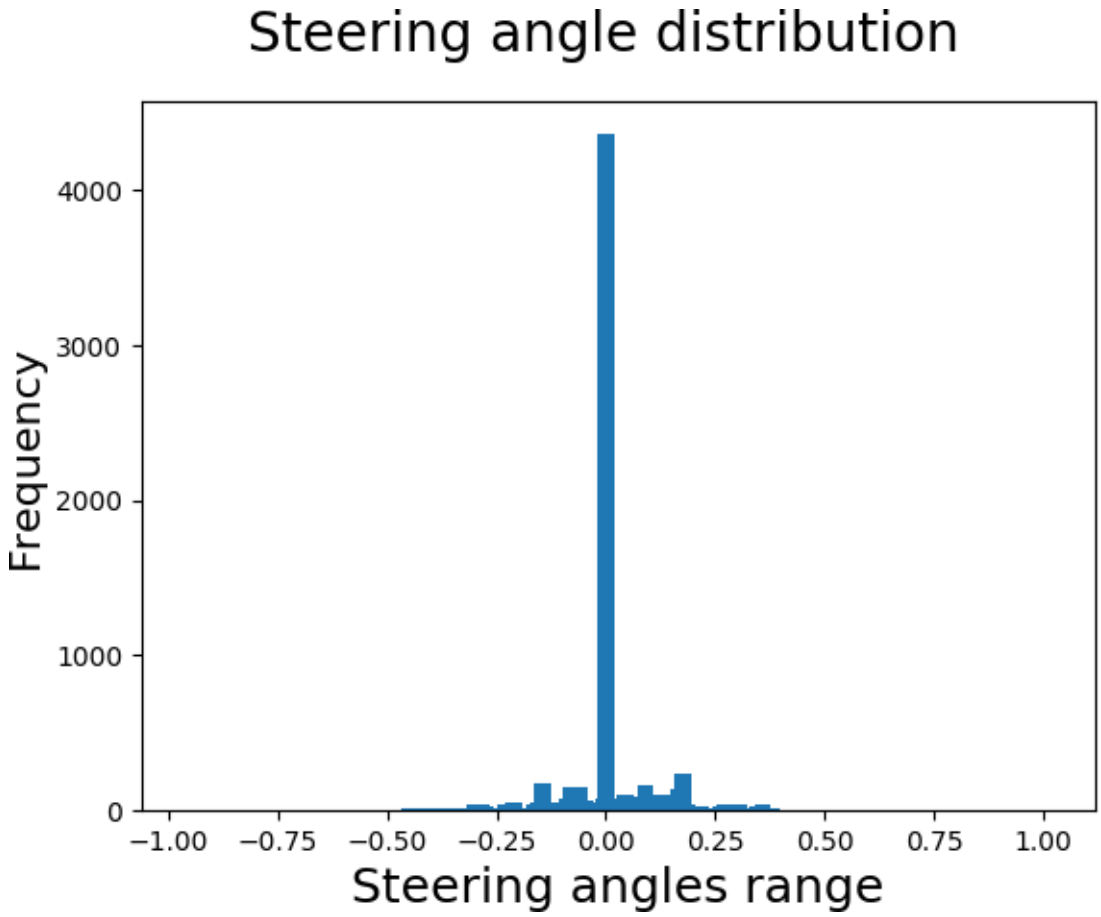
Let's look at the distribution of speed in our input data.


Speed Distribution
Frequency
7000
6000
5000
4000
3000
2000
1000
0
0
5
10
15
20
25
30
Speed range

Let’s look at the distribution of throttle



We are primarily concern with the steering angle and images. Let's look at the distribution of the steering angle



The data is highly skewed and unbalanced.

The main task is to drive a car around in a simulator on a race track, and then use deep learning to mimic the behavior of human. This is a very interesting problem because it is not possible to drive under all possible scenarios on the track, so the deep learning algorithm will have to learn general rules for driving. We must be very careful while using deep learning models, because they have a tendency to overfit the data. Overfitting refers to the condition where the model is very sensitive to the training data itself, and the model's behavior does not generalize to new/unseen data. One way to avoid overfitting is to collect a lot of data.

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. For our car example, this will require us to drive the car under different weather, lighting, traffic and road conditions. One way to avoid overfitting is to use augmentation.

Augmentation

Augmentation refers to the process of generating new training data from a smaller data set such that the new data set represents the real world data one may see in practice. As we are generating thousands of new training instances from each image, it is not possible to generate and store all these data on the disk. We will therefore utilize keras generators to read data from the file, augment on the fly and use it to train the model. We will utilize images from the left and right cameras so we can generate additional training data to simulate recovery. 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. After setting up the image augmentation pipeline, we can proceed to train the model. The training was performed using simple adam learning algorithm with learning rate of 0.0001. After this training, the model was able to drive the car by itself on the first track for hours and generalized to the second track.

All the training was based on driving data of about 4 laps using key board on track 1 in one direction alone. The model never saw track 2 in training, but with image augmentation (flipping, darkening, shifting, etc) and using data from all the cameras (left, right and center) the model

was able to learn general rules of driving that helped translate this learning to a different track.

Augmentation helps us extract as much information from data as possible. We will generate additional data using the following data augmentation techniques. Augmentation is a technique of manipulating the incoming training data to generate more instances of training data. This technique has been used to develop powerful classifiers with little data. However, augmentation is very specific to the objective of the neural network.

Brightness augmentation

Changing brightness to simulate day and night conditions. We will generate images with different brightness by first converting images to HSV, scaling up or down the V channel and converting back to the RGB channel

You can observe the images below for brightness augmentation.

Above Image is after applying random brightness

Above Image is after applying random brightness

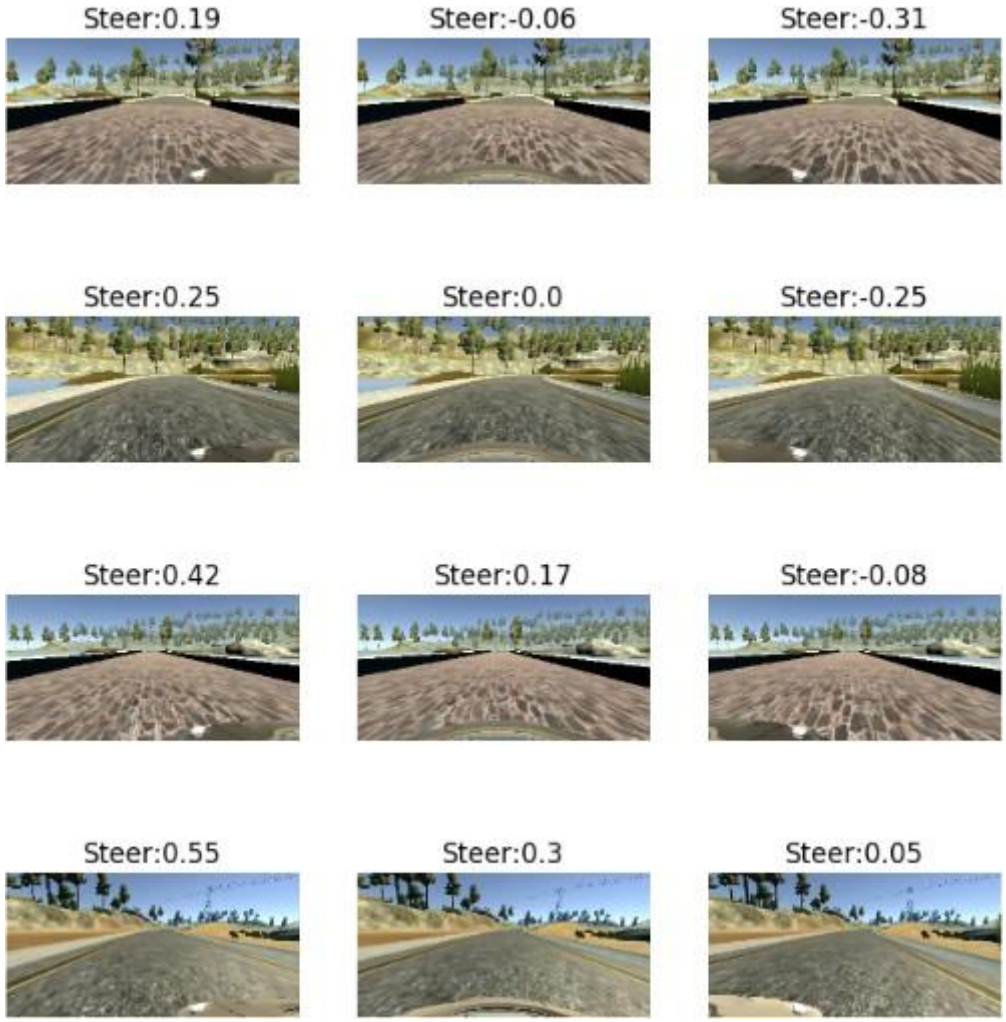
Above Image is after applying random brightness

Above Image is after applying random brightness

Using left and right camera images

Using left and right camera images to simulate the effect of car wandering off to the side, and recovering. We will add a small angle .25 to the left camera and subtract a small angle of 0.25 from the right camera.

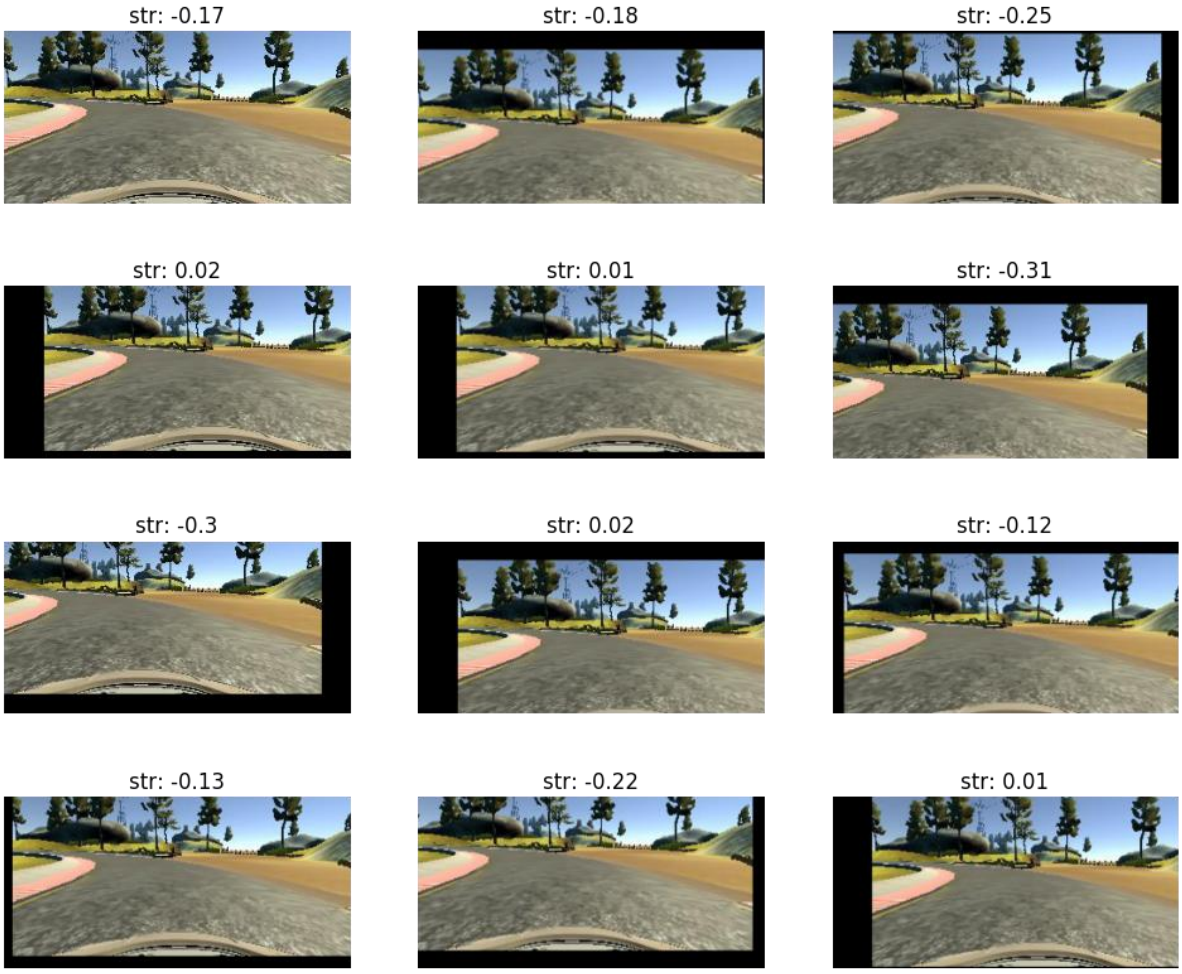
The main idea being the left camera has to move right to get to center, and right camera has to move left. See the images below for better understanding.



Horizontal and vertical shifts

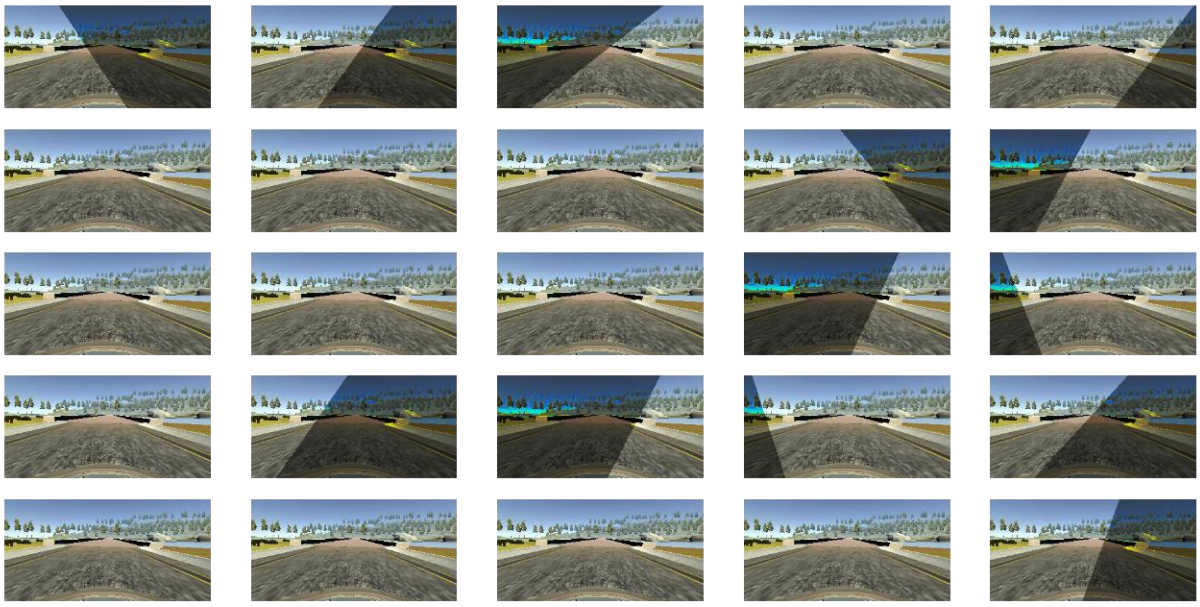
We will shift the camera images horizontally to simulate the effect of car being at different positions on the road, and add an offset corresponding to the shift to the steering angle. We added 0.004 steering angle units per pixel shift to the right, and subtracted 0.004 steering angle units per pixel shift to the left. We will also shift the images vertically by a random

number to simulate the effect of driving up or down the slop. See the images below for better understanding.



Shadow augmentation

The next augmentation we will add is shadow augmentation where random shadows are cast across the image. This is implemented by choosing random points and shading all points on one side (chosen randomly) of the image. The results for this augmentation is presented below.

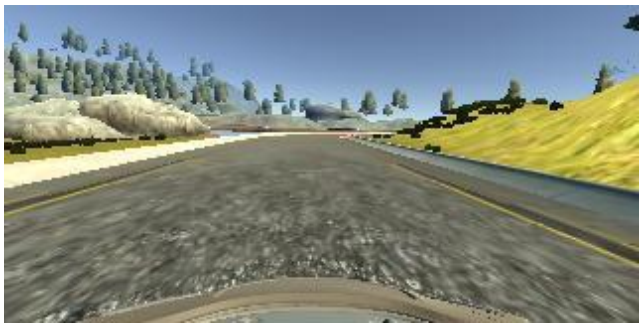


Flipping

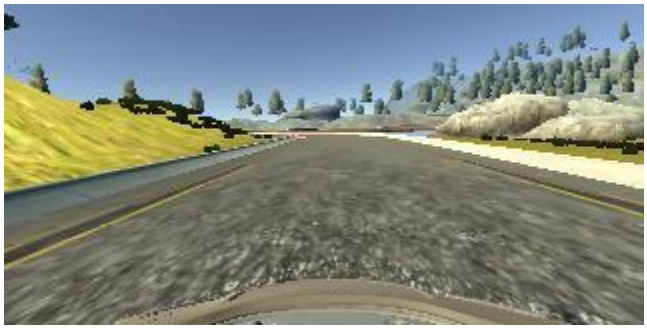
In addition to the transformations above, we will also flip images at random and change the sign of the predicted angle to simulate driving in the opposite direction.

We can understand the flipping operation by observing the original image and flipped image.

Original image



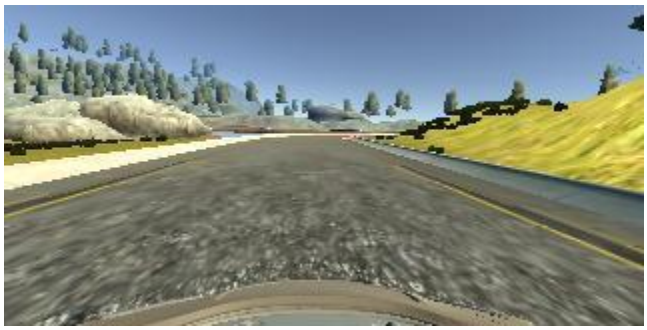
Flipped image



Preprocessing

After augmenting the image as above, we will crop the top 1/5 of the image to remove the horizon and the bottom 25 pixels to remove the car's hood. Originally 1/3 of the top of car image was removed, but later it was changed to 1/5 to include images for cases when the car may be driving up or down a slope. We will next rescale the image to a 64X64 square image. After augmentation, the augmented images looks as follows. These images are generated using keras generator, and unlimited number of images can be generated from one image. I used Lambda layer in keras to normalize intensities between -.5 and .5.

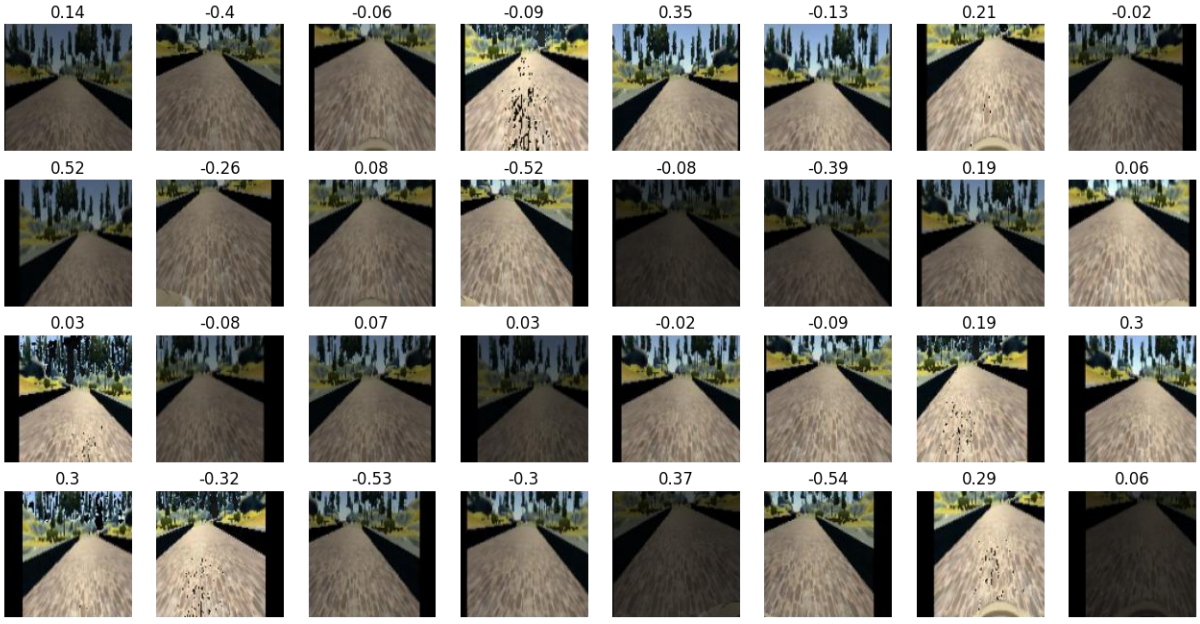
Orginal image



Cropped image

Keras generator forsubsampling

As there was limited data and we are generating thousands of training examples from the same image, it is not possible to store all the images apriori into memory. We will utilize keras generator function to sample images such that images with lower angles have lower probability of getting represented in the data set. This alleviates any problems we may ecounter due to model having a bias towards driving straight. Panel below shows multiple training samples generated from one image.



The Model

The model architecture has been inspired from this post and from End to End Deep Learning for Self Driving Car paper by NVIDIA.

There are two interesting bits about this architecture:

1. Its a simpleified version of the architcture mentioned in the paper published by NVIDIA.

2. The 1x1 convolutions at the beginning of this architecture.

To quote the author of this post,

> *"The first layer is 3 1X1 filters, this has the effect of transforming the color space of the images. Research has shown that different color spaces are better suited for different applications. As we do not know the best color space apriori, using 3 1X1 filters allows the model to choose its best color space"*

The 1x1 convolutions are followed by 3 sets of 3x3 convolution layers, of filter sizes 16, 32 and 64. Each set is accompanied with an max pooling layer and a drop out layer. These sets are followed by 3 fully connected layers. ELU has been chosen as the activation function.

---

10 Architecture post - https://github.com/vxy10/P3-BehaviorCloning

11 Author Architecture post - https://chatbotslife.com/using-augmentation-to-mimic-human-driving-496b569760a9

Model parameters:

1. Training/Test split = 0.8
2. Batch Size = 32
3. Number of epochs = 10
4. Learning rate = 1.0e-4
5. Samples per epoch = 20000
6. Optimizer : Adam

And now it's time to train.

Training

Having divided the dataset into train and test sets, generator is used to generate data on the fly for each epoch of training. The generator creates a batch of 32, where for each batch:

An image is chosen from amongst the left, center and right camera images.

The steering correction is set to be 0.25.

The image is then augmented by using aforementioned techniques.

The batches are then fed to the model.

Test

Using the Udacity provided simulator and drive.py[11] file, the car can be driven autonomously around the track by executing

From the results , we observed that the steering angles are somewhat correct, but not completely accurate. Hence, the car turns, but not quite. And on sharp turns, the *not quite* factor plays a big role in taking the car out of the track. We can improve that in 2 ways:

Retrain the model by tweaking parts of training configurations and settings such as data, model architecture etc.

Put a leash on the car and control it using that leash.

Resolve the Data distribution

A quick look into the data reveals that there are 6148 number of rows in our data that have steering values less than 0.01. Which justifies our model's tendency to move straight and not being able to make sharp turns. If we removed around 70% of those, we would be left with 3732 data points in total, which would be **comparatively** balanced.

Since we know that when the car turns, the steering angles do not go up to the required values, lets inflate through steering values received from the model and give our car a help around the turns. This would require modifying the drive.py file.

[11] https://github.com/udacity/CarND-Behavioral-Cloning-P3/blob/master/drive.py

```
if  np.absolute(steering_angle)>0.07:
    print('super inflating angles')
    steering_angle*=2.5
    throttle=0.001
elif np.absolute(steering_angle)>0.04:
    print('inflating angles')
    steering_angle*=1.5
    throttle=0.1
```

Now Test Again

It works. we observed , after removing the data points for straight path navigation, there's a swivel in the movement of the car, which means, the learning could be improved a bit here. Also, at the sharp turns, it no longer turns at the last moment, relying on the control code to get it through, signifying improved learning for turns.

CODE IMPLEMENTATION

```
import pandas as pf
from keras.optimizers import Adam
from keras.callbacks import TensorBoard
from keras.callbacks import ModelCheckpoint, Callback, EarlyStopping
from keras.layers.convolutional import Conv2D, Cropping2D
from keras.layers.pooling import MaxPooling2D
from keras.layers import Flatten, Dense, Lambda, ELU, Dropout
from keras.layers.advanced_activations import LeakyReLU, PReLU
from keras.models import Sequential
import numpy as np
from keras.preprocessing.image import img_to_array, load_img
from keras.layers.normalization import BatchNormalization
import cv2
DATA_DIRECTORY = 'data/'
TRAINING_SPLIT = 0.8
BATCH_SIZE = 32
LEARNING_RATE = 1.0e-4
EPOCHS = 15
MODEL_NAME =
        'model_hsv_70_elu_data_proper_enhancement_extreme_bn.h5'
MODEL_ROW_SIZE = 64
MODEL_COL_SIZE = 64
NO_OF_CHANNELS = 3
MODEL_INPUT_SHAPE =
        (MODEL_ROW_SIZE,
        MODEL_COL_SIZE,
NO_OF_CHANNELS)
TARGET_SIZE = (MODEL_ROW_SIZE, MODEL_COL_SIZE)
STEERING_CORRECTION = 0.25
STEERING_THRESHOLD = 0.15
STEERING_KEEP_PROBABILITY_THRESHOLD = 1
IMAGE_WIDTH = 320
IMAGE_HEIGHT = 160
#ACTIVATION = 'LeakyReLU'
ACTIVATION = 'elu'
```

```
def convert_to_YUV(image):
"converts the image from RGB space to YUV space"'
image = cv2.cvtColor(image, cv2.COLOR_RGB2YUV)
return image
def convert_to_HSV(image):
"'converts the image from RGB space to YUV space"'
image = cv2.cvtColor(image, cv2.COLOR_RGB2HSV)
return image
def resize_image(image):
"'resize the image to 64 by 64 size"
'return cv2.resize(image, TARGET_SIZE)

def crop_image(image):

'''
:param image: The input image of dimensions 160x320x3
:crops out the sky and bumper portion of the car form the image
'''
cropped_image = image[55:135, :, :]
return cropped_image
def translate_image(image, steering_angle, range_x=100, range_y=10):
"""
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 shift_horizon(img):
h, w, _ = img.shape
horizon = 2 * h / 5
v_shift = np.random.randint(-h/8,h/8)
pts1 = np.float32([[0,horizon],[w,horizon],[0,h],[w,h]])pts2 =
np.float32([[0,horizon+v_shift],[w,horizon+v_shift],[0,h],[w,h]])
M = cv2.getPerspectiveTransform(pts1,pts2)
```

```
return cv2.warpPerspective(img,M,(w,h),
borderMode=cv2.BORDER_REPLICATE)'''
def preprocess_image(image):
'''does the crop, resize and conversion to YUV space of the image'''
image = crop_image(image)
image = resize_image(image)
#image = convert_to_YUV(image)
image = convert_to_HSV(image)
image = np.array(image)
return image
def choose_image(row_data):
'''Selects an image from amongst the center camera image, left camra image
and the rigt camera image '''
toss = np.random.randint(3)
img_path = ''
steering = 0.0
if toss == 0:
#choose center image
img_path = row_data.iloc[0]
steering = row_data.iloc[3]
elif toss==1:
#choose left image
img_path = row_data.iloc[1]
steering = row_data.iloc[3] + STEERING_CORRECTION
elif toss==2:
#choose right image
img_path = row_data.iloc[2]
steering = row_data.iloc[3] - STEERING_CORRECTION
return img_path, steering
def load_image(img_path):
'''loads an image by reading from a file path'''
img_path = img_path.strip()
#print('image path-->', DATA_DIRECTORY + img_path)
image = cv2.imread(DATA_DIRECTORY + img_path)
image = cv2.cvtColor(image,cv2.COLOR_BGR2RGB)
return image
def bright_augment_image(img):
'''Adds rando mbrightness to the image'''
img1 = cv2.cvtColor(img,cv2.COLOR_RGB2HSV)
random_bright = .25 + np.random.uniform()
```

```
img1[:,:,2] = img1[:,:,2]*random_brightimg1 =
cv2.cvtColor(img1,cv2.COLOR_HSV2RGB)
return img1
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)
return img1
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 augment_image(row_data):
'''auments the input image by translating, adding random shadow, augmenting
brightness and preprocessing (crop , resize and conversion to YUV space)
the image'''
img_path, steering = choose_image(row_data)
image = load_image(img_path)
#translate image
image, steering = translate_image(image, steering)
#augment brightness
image = bright_augment_image(image)
#image = shift_horizon(image)
#flip image
# This is done to reduce the bias for turning left that is present in the training data
flip_prob = np.random.random()
if flip_prob > 0.5:
# flip the image and reverse the steering angle
steering = -1*steering
image = cv2.flip(image, 1)
#random shadow
image = random_shadow(image)
#preprocess image
image = preprocess_image(image)
return image, steering
def get_data_generator(data_frame):
'''generator to generate data for the model on the fly.'''
batch_size = BATCH_SIZE
print('data generator called')
N = data_frame.shape[0]
print('N -->', N)
batches_per_epoch = N // batch_size
```

```
print('batches per epoch-->', batches_per_epoch)
i = 0
while(True):
start = i*batch_size
end = start+batch_size - 1
X_batch = np.zeros((batch_size, 64, 64, 3), dtype=np.float32)
y_batch = np.zeros((batch_size,), dtype=np.float32)
j = 0
# slice a `batch_size` sized chunk from the dataframe
# and generate augmented data for each row in the chunk on the fly
for index, row in data_frame.loc[start:end].iterrows():
image, steering = augment_image(row)
X_batch[j] = image
y_batch[j] = steering
j += 1
i += 1
if i == batches_per_epoch - 1:
# reset the index so that we can cycle over the data_frame again
i = 0
yield X_batch, y_batch
def main():
'''main function from where the code flow starts'''
print('Behavioral Cloning')
org_data_frame = load_data()
print('Data loaded')
“get_data_stats(org_data_frame)train_data, valid_data, new_data_frame =
filter_dataset(org_data_frame)”
print('Data filtered')
#get_data_stats(new_data_frame)
org_data_frame = None
print('calling generators')
training_generator = get_data_generator(train_data)
validation_data_generator = get_data_generator(valid_data)
model = get_model()
adam = Adam(lr=LEARNING_RATE, beta_1=0.9, beta_2=0.999, epsilon=1e-08,
decay=0.0)
model.compile(optimizer = 'adam', loss = 'mse')
early_stopping = EarlyStopping(monitor='val_loss', patience=8, verbose=1,
mode='auto')
```

```
save_weights = ModelCheckpoint('model.h5', monitor='val_loss',
save_best_only=True)
tensorboard = TensorBoard(log_dir='lossandvalloss/', histogram_freq=0,
                  write_graph=True, write_images=False)
model.fit_generator(training_generator,
validation_data=validation_data_generator, nb_epoch =
EPOCHS,
callbacks=[tensorboard], samples_per_epoch = 20000, nb_val_samples =
len(valid_data))
“model.fit(np.array([[1]]), np.array([[2]]), batch_size=BATCH_SIZE, shuffle =
True, nb_epoch =
EPOCHS, callbacks=[save_weights, early_stopping])”
#save the model
model.save(MODEL_NAME)
print("Model Saved!")
def get_model():
model = Sequential()
#model.add(Lambda(lambda x: x / 255., input_shape=MODEL_INPUT_SHAPE,
name='normalizer'))
'"model.add(Cropping2D(cropping=crop,
input_shape=(row, col, ch), name='cropping'))"'
model.add(Conv2D(3, 1, 1, subsample=(2, 2),
input_shape=MODEL_INPUT_SHAPE,activation=ACTIVATION, name='cv0'))
model.add(BatchNormalization())
model.add(Conv2D(16, 3, 3, activation=ACTIVATION, name='cv1'))
model.add(BatchNormalization())
model.add(MaxPooling2D(name='maxPool_cv1'))
model.add(Dropout(0.5, name='dropout_cv1'))
model.add(Conv2D(32, 3, 3, activation=ACTIVATION, name='cv2'))
model.add(BatchNormalization())
model.add(MaxPooling2D(name='maxPool_cv2'))
model.add(Dropout(0.5, name='dropout_cv2'))
model.add(Conv2D(64, 3, 3, activation=ACTIVATION, name='cv3'))
model.add(BatchNormalization())model.add(MaxPooling2D(name='maxPool_cv3'
))
model.add(Dropout(0.5, name='dropout_cv3'))
model.add(Flatten())
model.add(Dense(1000, activation=ACTIVATION, name='fc1'))
model.add(BatchNormalization())
model.add(Dropout(0.5, name='dropout_fc1'))
```

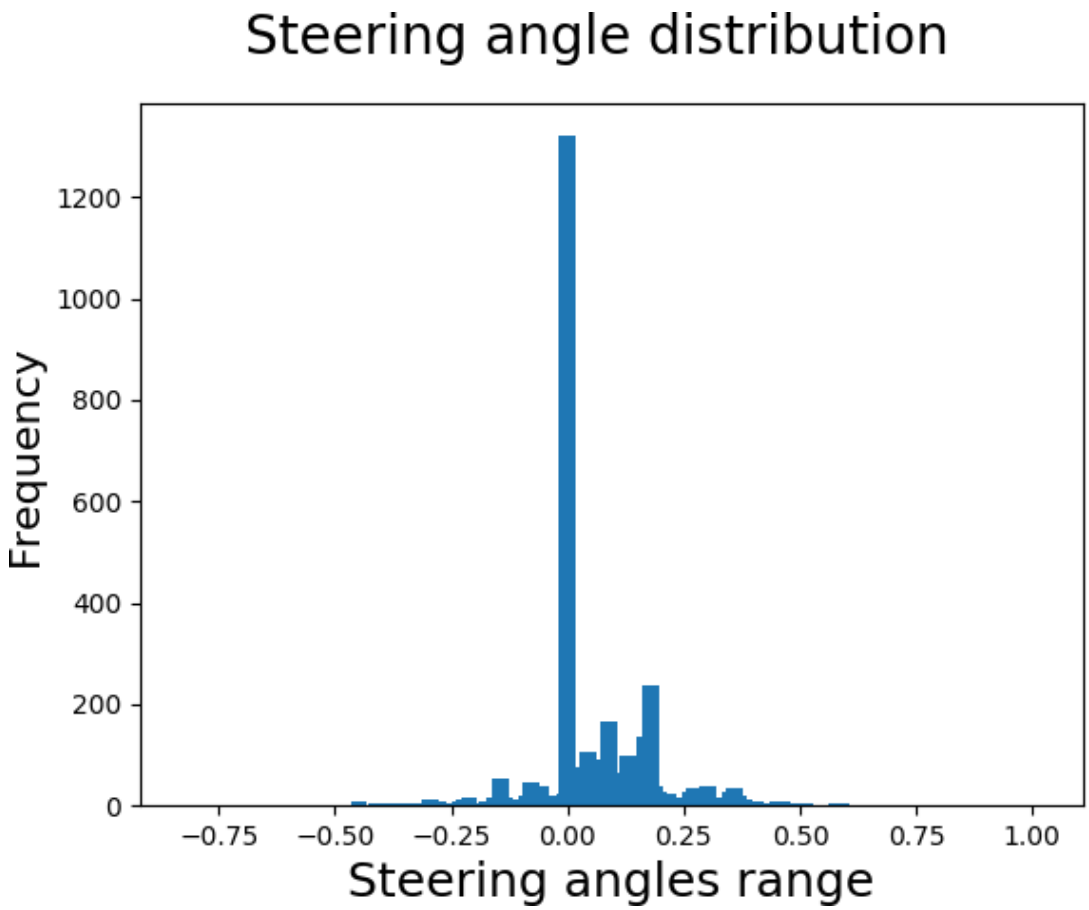
```
model.add(Dense(100, activation=ACTIVATION, name='fc2'))
model.add(Dense(100, activation=ACTIVATION, name='fc2'))
model.add(BatchNormalization())
model.add(Dropout(0.5, name='dropout_fc2'))
model.add(Dense(10, activation=ACTIVATION, name='fc3'))
model.add(BatchNormalization())
model.add(Dropout(0.5, name='dropout_fc3'))
model.add(Dense(1, name='output'))
return model
#even out the steering angle values in data
#filter out the 0 and lower speed values from data
def filter_dataset(data_frame):
data_frame = filter_steering(data_frame)
# replicate rows having extreme values of steering angles
#data_frame = replicate_steering(data_frame)
data_frame = replicate_proper_steering(data_frame)
num_rows_training = int(data_frame.shape[0]*TRAINING_SPLIT)
training_data = data_frame.loc[0:num_rows_training-1]
validation_data = data_frame.loc[num_rows_training:]
return training_data, validation_data, data_frame
def filter_steering(data_frame):
'''evens out the steering angle distribution in the data set by removing
70% of the rows with steering angle close to 0 (<0.01).'''
print('Steering filtering')
print ('number of rows with steering less than 0.01: ',
len(data_frame.loc[data_frame['steering']
<0.007]))
data_frame = data_frame.drop(data_frame[data_frame['steering']
<0.01].sample(frac=0.70).index)
print('new data frame length-->', len(data_frame))
return data_frame
def replicate_steering(data_frame):
print('Steering replicating')
new_data_frame = data_frame.loc[data_frame['steering'] >
0.20].append(data_frame.loc[data_frame['steering'] < - 0.20])
#print('new data frame-->', new_data_frame)
num_of_angles = len(data_frame.loc[data_frame['steering'] >
0.20].append(data_frame.loc[data_frame['steering'] < - 0.20]))
data_frame = data_frame.append([new_data_frame]*5,ignore_index=True)
#print()
```

```
#df_new = data_frame.loc[np.repeat(data_frame.loc[data_frame['steering'] >
#0.20].append(data_frame.loc[data_frame['steering'] < - 0.20]),
[5]*num_of_angles)]
return data_frame
def replicate_proper_steering(data_frame):
print('Steering proper replicating')
data_frame_0_10 = data_frame.loc[(data_frame['steering'] >= 0.008) &
(data_frame['steering'] <
0.10)]
data_frame_10_20 = data_frame.loc[(data_frame['steering'] >= 0.10) &
(data_frame['steering'] <
0.20)]
data_frame_20_30 = data_frame.loc[(data_frame['steering'] >= 0.20) &
(data_frame['steering'] <
0.30)]
data_frame_30_40 = data_frame.loc[(data_frame['steering'] >= 0.30) &
(data_frame['steering'] <
0.40)]
data_frame_40_50 = data_frame.loc[(data_frame['steering'] >= 0.40) &
(data_frame['steering'] <
0.50)]
data_frame_50 = data_frame.loc[data_frame['steering'] >= 0.50]
data_frame = data_frame.append([data_frame_0_10]*3,ignore_index=True)
data_frame = data_frame.append([data_frame_10_20]*2,ignore_index=True)
data_frame = data_frame.append([data_frame_20_30]*8,ignore_index=True)
data_frame = data_frame.append([data_frame_30_40]*6,ignore_index=True)
data_frame = data_frame.append([data_frame_40_50]*8,ignore_index=True)
data_frame = data_frame.append([data_frame_50]*60,ignore_index=True)
data_frame_neg_1_10 = data_frame.loc[(data_frame['steering'] <= -0.008) &
(data_frame['steering'] > -0.10)]
data_frame_neg_10_20 = data_frame.loc[(data_frame['steering'] <= -0.10) &
(data_frame['steering'] > -0.20)]
data_frame_neg_20_30 = data_frame.loc[(data_frame['steering'] <= -0.20) &
(data_frame['steering'] > -0.30)]
data_frame_neg_30_40 = data_frame.loc[(data_frame['steering'] <= -0.30) &
(data_frame['steering'] > -0.40)]
data_frame_neg_40_50 = data_frame.loc[(data_frame['steering'] <= -0.40) &
(data_frame['steering'] > -0.50)]
data_frame_neg_50 = data_frame.loc[data_frame['steering'] <= -0.50]
data_frame = data_frame.append([data_frame_neg_1_10]*8,ignore_index=True)
```

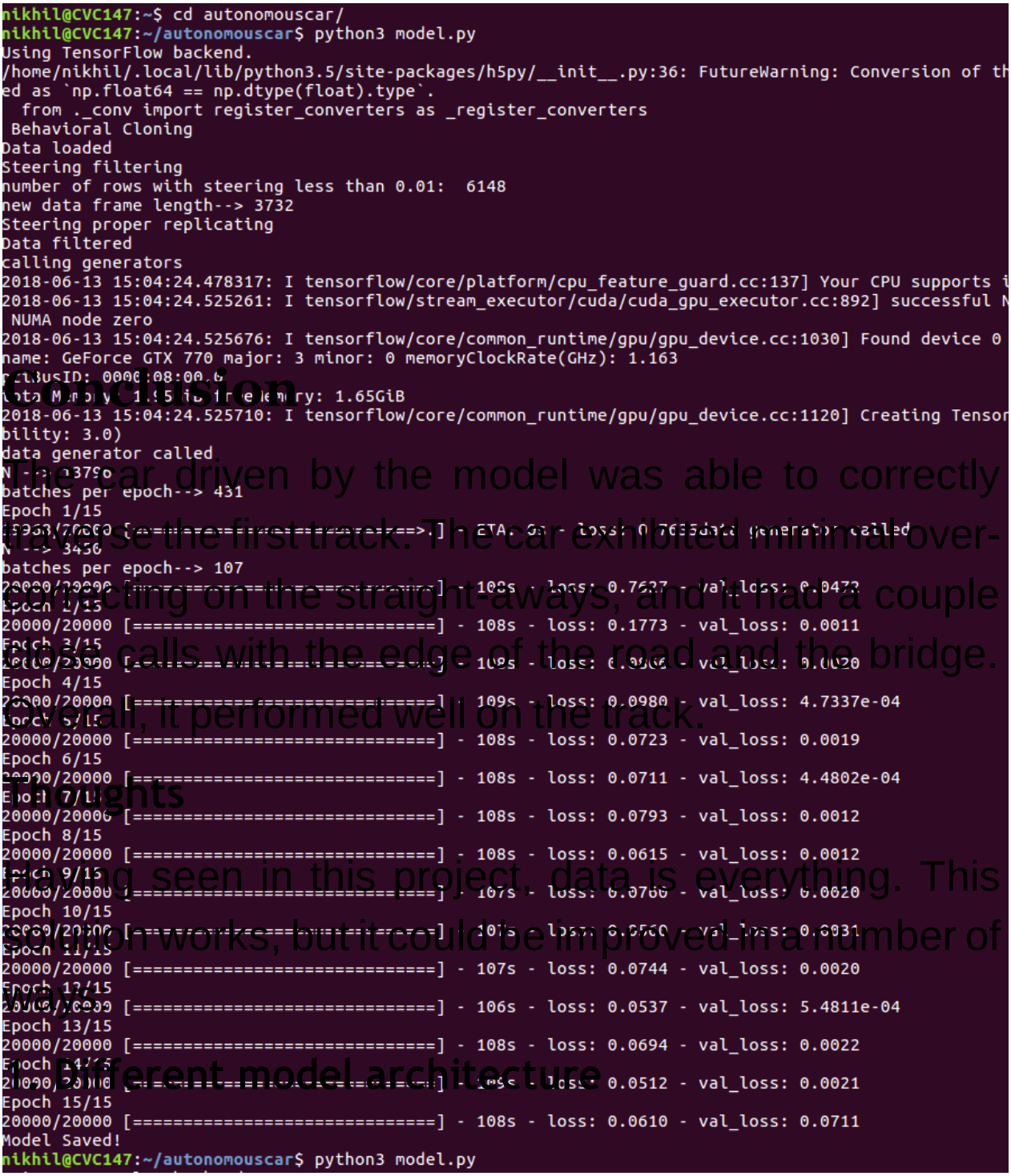
```
data_frame = data_frame.append([data_frame_neg_10_20]*8,ignore_index=True)
data_frame = data_frame.append([data_frame_neg_20_30]*20,ignore_index=True)
data_frame = data_frame.append([data_frame_neg_30_40]*20,ignore_index=True)
data_frame = data_frame.append([data_frame_neg_40_50]*20,ignore_index=True)
data_frame = data_frame.append([data_frame_neg_50]*80,ignore_index=True)
#print('new data frame-->', new_data_frame)
#num_of_angles = len(data_frame.loc[data_frame['steering'] >
#0.20].append(data_frame.loc[data_frame['steering'] < - 0.20]))
#print()
#df_new = data_frame.loc[np.repeat(data_frame.loc[data_frame['steering'] >
#0.20].append(data_frame.loc[data_frame['steering'] < - 0.20]),
[5]*num_of_angles)]
return data_frame
def filter_throttle():
'''filters out the throttle values less than 0.25'''
print('Throttle filtering')
#print (len(org_data_frame.loc[org_data_frame['throttle'] >=0.5]))
def load_data():
'''loads the data from the driving log .csv file into a dataframe'''
data_frame = pd.read_csv(DATA_DIRECTORY +'driving_log.csv')
# shuffle the data
data_frame = data_frame.sample(frac=1).reset_index(drop=True)
# 80-20 training validation split
'''num_rows_training = int(data_frame.shape[0]*TRAINING_SPLIT)
training_data = data_frame.loc[0:num_rows_training-1]
validation_data = data_frame.loc[num_rows_training:]'''
return data_frame
if__name__== '__main__':
main()
```

Outputs and Results

we can see that steering angle distribution is balanced.



From the screen shot below, We can observe the loss and val_loss of our model per each epoch while training the model.



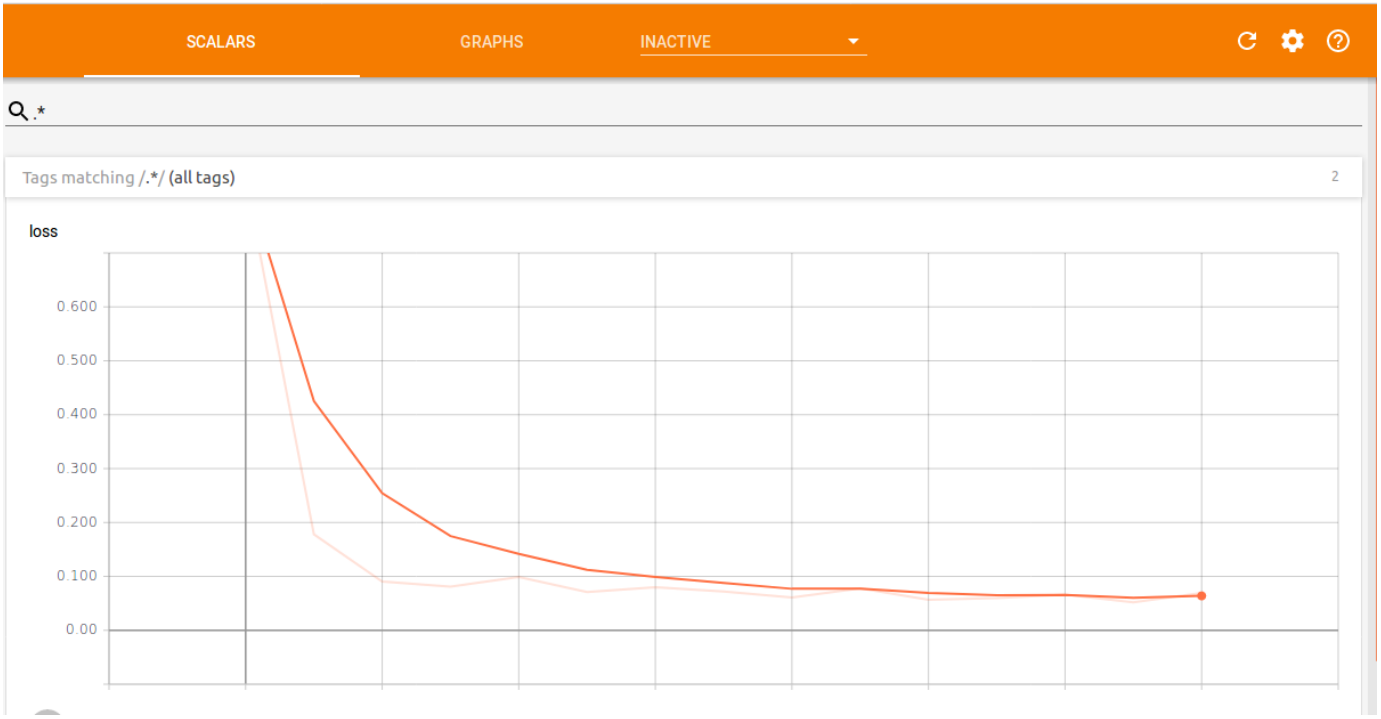
Conclusion

The car driven by the model was able to correctly traverse the first track. The car exhibited minimal over-correcting on the straight-aways, and it had a couple close calls with the edge of the road and the bridge. Overall, it performed well on the track.

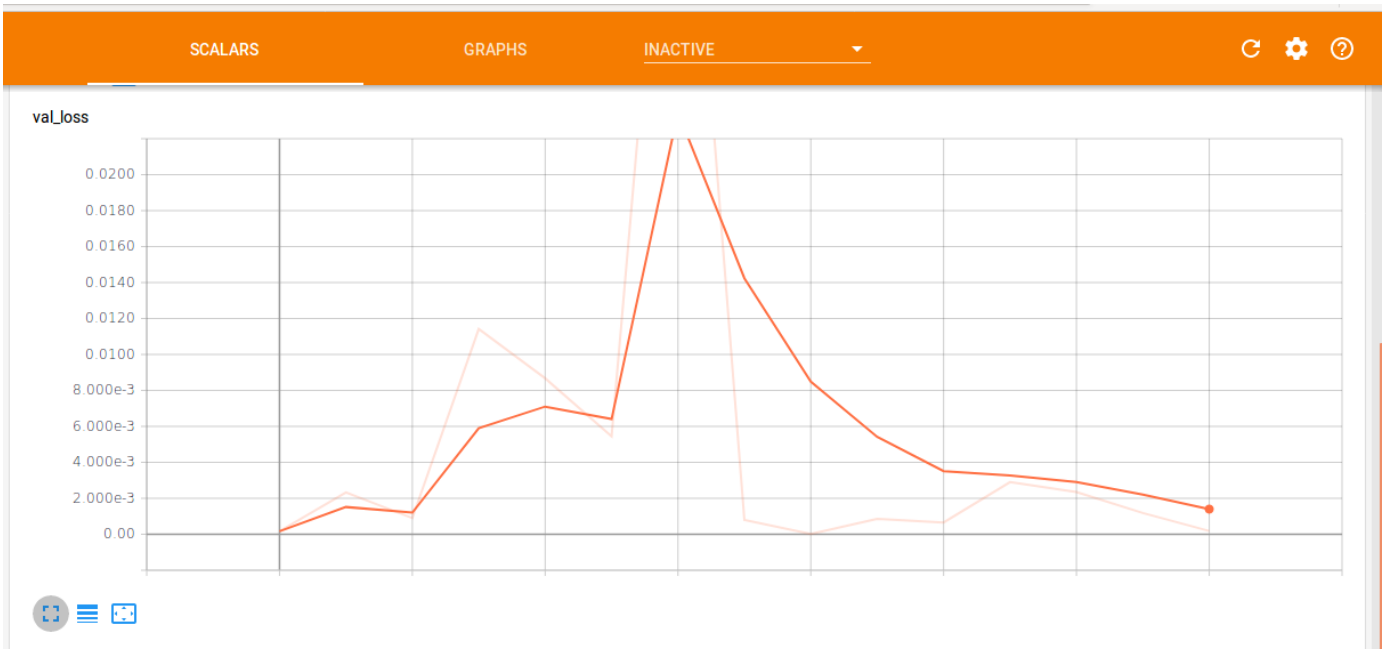
Thoughts

Having seen in this project, data is everything. This solution works, but it could be improved in a number of ways. Different model architecture

Loss



Validation_Loss



from the screen shot below, you can see that the car driving autonomously in the simulator.

for complete video of the car running autonomously in the simulator, see this video [33].

Conclusion

The car driven by the model was able to correctly traverse the track. The car exhibited minimum over-correcting on the straight – aways, and it had a couple close calls with the edge of the road and the bridge. Overall, it performed well on the track.

Thoughts

Having seen in this project, data is everything. This solution works, but it could be improved in a number of ways.

1. Different Model Architecture

Although the dataset is not that large or complex, still it could be insightful how some of the other architectures would perform for this project.

2. Data collection

We could actually try collecting data **ourselves**, the way it is suggested with recovery and regular laps. The point is, we could see what effect does more data have on this implementation and how that would change things.

3. Driving like a human

Right now, the car is able to drive around both tracks, but let's face it, if it were a real world driving, it would end up getting tickets even before reaching the first turn. We could try teaching the model lane driving, which would mean some added data augmentation techiniques and better data to work with.

4. Break and Throttle

Currently, our convolutional neural network only predicts the values of steering angles using images. It could be enhanced to predict the throttle and break values as well.

References

[1]. Udacity Self Driving Car Simulator - https://github.com/udacity/self-driving-car-sim

[2]. End to End Learning for Self Driving Cars Paper – https://arxiv.org/pdf/1604.07316v1.pdf

[3]. NVIDIA Website - http://www.nvidia.com/content/global/global.php

[4]. Neural Networks Basics - http://pages.cs.wisc.edu/~bolo/shipyard/neural/local.html

[5]. Python Website - https://www.python.org/

[6]. ImageNet Classification with Deep Convolutional Neural Networks - http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[7]. L. D. Jackel, D. Sharman, Stenard C. E., Strom B. I., , and D Zuckert. Optical character recognition for self-service banking. AT&T Technical Journal, 74(1):16–24, 1995

[8]. IMAGENET Large Scale Visual Recognition Challenge (ILSVRC) - http://www.image-net.org/challenges/LSVRC/

[9]. DAVE Final Technical Report – http://net-scale.com/doc/net-scale-dave-report.pdf

[10]. Autonomous Land Vehicle In a Neural Network paper by Dean A. Pamerleau, January 1989 - http://repository.cmu.edu/cgi/viewcontent.cgi?article=2874&context=compsci

[11]. DAPRA LAGR Program - https://en.wikipedia.org/wiki/DARPA_LAGR_Program

[12]. Danwei Wang and Feng Qi. Trajectory planning for a four-wheel- steering vehicle. In Proceedings of the 2001 IEEE International Conference on Robotics & Automation, May 21–26 2001. URL: http://www.ntu.edu.sg/home/edwwang/confpapers/wdwicar01.pdf.

[13]. DAVE -2, Neural Network drives a car video - https://www.youtube.com/watch?time_continue=2&v=NJU9ULQUwng

[14]. Python Documentation - https://docs.python.org/3/index.html

[15]. Pandas Documentation - https://pandas.pydata.org/pandas-docs/stable/

[16]. Numpy Documentation - https://docs.scipy.org/doc/

[17]. Matplotlib Documentation - https://matplotlib.org/contents.html

[18]. Scikit-learn Documentation -http://scikit- learn.org/stable/documentation.html

[19]. OpenCV Installation Linux - https://www.learnopencv.com/install-opencv3-on-ubuntu/

[20]. OpenCV Documentation - https://docs.opencv.org/2.4/index.html [21].

PillowDocumentation-https://pillow.readthedocs.io/en/5.1.x/ [22].

Scikit-image Documentation - http://scikit-image.org/docs/stable/ [23]. Scipy

Documentation - https://www.scipy.org/docs.html

[24]. h5py Documentation - http://docs.h5py.org/en/latest/

[25]. Eventlet Documentation - http://eventlet.net/doc/

[26]. Flask-Socketio Documentation - https://python-socketio.readthedocs.io/en/latest/

[27]. Seaborn Documentation - https://seaborn.pydata.org/

[28]. Imageio Documentation - https://imageio.readthedocs.io/en/stable/

[29]. Moviepy Documentation - https://zulko.github.io/moviepy/

[30]. Google AI Organisation was announced at Google I/O 2017 by CEO Sundar Pichai URL: https://ai.google/

[31]. Tensor Flow Documentation guide - https://www.tensorflow.org/programmers_guide/

[32]. Keras Documentation - https://keras.io/

[33]. Complete Result Video - https://drive.google.com/file/d/1oli3yQrNIskrSsjFzahlfXmLCShPTv_s/view?usp=sharing