Overview

1.Data Generation

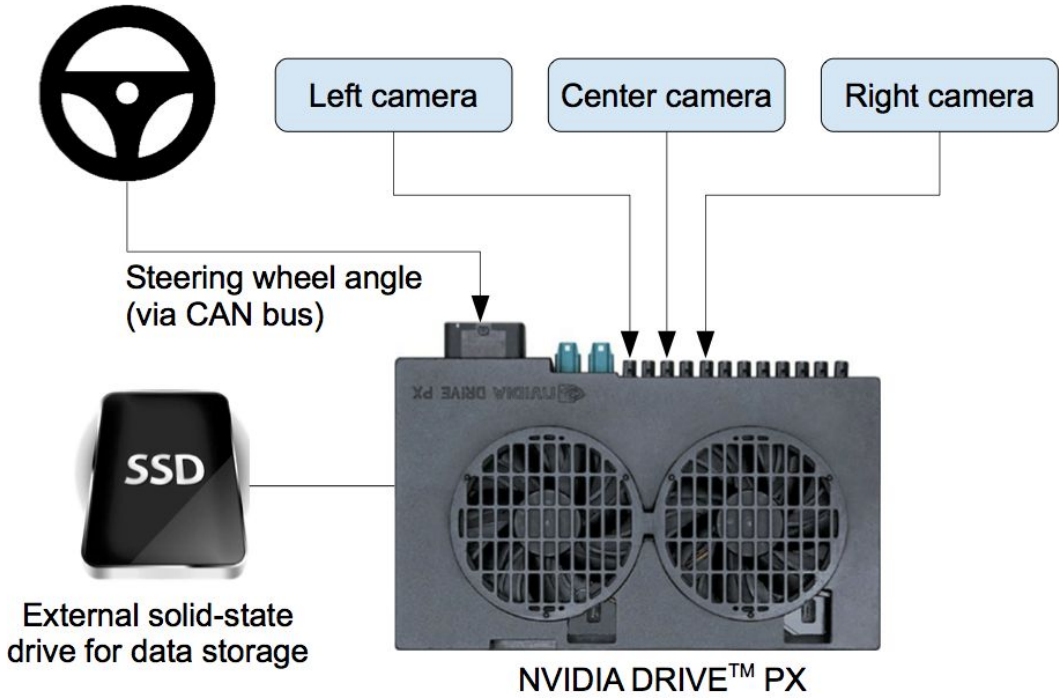
Udacity built a 9 layer convolutional network and fixed three sets of camera and four physics variables- speed, steering angle, throttle and brake and static images from three different angles- left, right and centre were recorded. We are going to do something similar here. We shall generate the data using the udacity self-driving-car simulator.

2. Training model

We shall do something called Behavioral cloning. Behavioral cloning is a method by which human subcognitive skills can be captured and reproduced in a computer program. As the human subject performs the skill, his or her actions are recorded along with the situation that gave rise to the action.
We shall build a 9 layer convolutional neural network and train it.

3. Hardware Design

Steering wheel angle recorded and fed to controller area network bus. Camera images fed frame by frame and an SSD to store the results.



In order to make the system independent of the car geometry, the steering command is 1/r, where r is the turning radius in meters. 1/r was used 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).

4. Supervised learning

We just need to teach our machines- Based on what you see, how should you move the car. Steering wheel and camera angle from the autonomous car and that from human generated data, are vectorized and compared and the error term is generated and backpropagation is used to update our initially chosen weights. Images are fed into the CNN and a steering command is predicted. We find the difference between the predicted and actual steering command from training data and then weights of CNN are adjusted to bring it closer to actual steering commands.

5. Testing mode

It can be thought of as a server(Self-Driving Car Simulator) – client(Python Program-Deep learning model) architecture. It's going to be a feedback loop.

Implementation

1. **Environment setup:**
   Install the dependencies from the environments file.
   //conda env create -f environments.yml//

2. **Activate the environment:**
   //source activate self-drive//

3. **Generate data:**
   Drive the car in training mode and record the data while doing so. You can do it by pressing the R button. Control the car using W-A-S-D keys. Record between 1 and 5 laps(ideally). Hit R again and it will capture the data and save it all to a csv file. The first three columns in your csv file will be images and you can have a look at them in the IMG folder. You will find Three sets of images- left, right, centre in first three columns and next four columns are those 4 physics variables.

4. **Training script.**

   It will consist of three functions basically:
   A. Loading the Data
   B. Building the model
   C. Training the model

i) Read the csv file, load the data and then split it into training and validation sets.

ii) We want to build a convolutional network.

Some important tools/methods/functionalities used:

- This is what CNN is: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

Essence of building a model :"Input times weights , add Bias and Activate"

- **Activation functions** are really important for an Artificial Neural Network to learn and make sense of something really complicated and Non-linear complex functional mappings between the inputs and response variable.They introduce non-linear properties to our Network.Their main purpose is to convert an input signal of a node in a A-NN to an output signal. That output signal now is used as an input in the next layer in the stack.

  Specifically in A-NN we do the sum of products of inputs(X) and their corresponding Weights(W) and apply an Activation function f(x) to it to get the output of that layer and feed it as an input to the next layer.

  We have used the ELU activation function as it prevents the problem of vanishing gradient. As more layers using certain activation functions are added to neural networks, the gradient of the loss function approaches zero, making the network hard to train.

  https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html
  https://sefiks.com/2018/01/02/elu-as-a-neural-networks-activation-function/

- **Dropout**: Simply put, dropout refers to ignoring units (i.e. neurons) during the training phase of a certain set of neurons which is chosen at random. By "ignoring", I mean these units are not considered during a particular forward or backward pass.
  More technically, At each training stage, individual nodes are either dropped out of the net with probability 1-p or kept with probability p, so

that a reduced network is left; incoming and outgoing edges to a dropped-out node are also removed.

This is done to prevent overfitting.

A fully connected layer occupies most of the parameters, and hence, neurons develop co-dependency amongst each other during training which curbs the individual power of each neuron leading to over-fitting of the training data.

https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5

The code will perform the following operations in order to build a model:

Image normalization to avoid saturation and make gradients work better.

1. **Convolution: 5x5, filter: 24, strides: 2x2, activation: ELU**
2. **Convolution: 5x5, filter: 36, strides: 2x2, activation: ELU**
3. **Convolution: 5x5, filter: 48, strides: 2x2, activation: ELU**
4. **Convolution: 3x3, filter: 64, strides: 1x1, activation: ELU**
5. **Convolution: 3x3, filter: 64, strides: 1x1, activation: ELU**

   Drop out (0.5)
6. **Fully connected: neurons: 100, activation: ELU**
7. **Fully connected: neurons: 50, activation: ELU**
8. **Fully connected: neurons: 10, activation: ELU**
9. **Fully connected: neurons: 1 (output)**

iii) Train the model:

a. Modelcheckpoint- saving model after every epoch.
b. Gradient descent to minimize the mean squared error using adam optimizer.
c. Fit generator to fit the model on the data generated batch by batch by a python generator: https://keras.io/models/sequential/

5. Testing Script

1. Setup the server using socket io.
2. Initialize your server and create a flask object- The flask object implements a WSGI application and acts as the central object.

3. Initialize our model and image array as empty.
4. Register the event handler for the server. Define a function- “telemetry” which as the name suggests, records and transmits the readings of the car. It will do the following operations: Get the current steering angle, throttle and speed and the image. Convert it into an array and predict the steering angle. Adjust the speed and throttle accordingly. Also adjust for min and max speed limit. Send the controls to the car. It will also save the current frame with a timestamp.
5. Define special event handler:
   ```
   @sio.on('connect')
   def connect(sid, environ):
       print("connect ", sid)
       send_control(0, 0)
   ```
   The connect event is special; it is invoked automatically when a client connects to the server.
6. Define a function `send control` to send the steering controls to the server.
7. Now, in the main function: load your saved model in this program.
   `model = load_model(args.model)`
8. Wrap your Flask application with engine io’s middleware.
9. Deploy as an eventlet WSGI server

Note: Besides these two scripts, there is a helping class- utils.py file, which generates a batch of images and associated steering angles. Using this class we can take an image and randomly flip it, translate it, adjust brightness and shadow and preprocess it and adjust the steering angles to generate a batch of images and steering angles. This helper class is used in training script while fitting the model on data generated by this program and also in testing script while for preprocessing the image converting it into a 4D array.