Deep Learning CNN for Self Driving Car

Introduction

Solving the self-driving car problem is one of the machine learning industry's main focus nowadays. In this report we will replicate the methodology originated from Nvidia's paper [1].

The main attraction of Nvidia's approach is that minimal training data was used to achieve automated driving while avoiding the need to recognize "human-designated" features. The result of this paper is the trained network's ability to automate steering command across different environment including roads with and without lane markings, on both local roads and highways.

Method

Track Selection

There are two tracks in Udacity's open source car simulator, a desert and a jungle track. The Desert track is an easier track as the entire track is on flat ground. On the other hand, the jungle track has slopes and blind spots, thus it is more difficult to learn. For this project, we have trained our models on both tracks.

(a) Desert Track (b) Jungle Track

Figure 1: Manual data collection via Udacity's simulator.

Data Collection

Data is generated through a recent open source car simulation from Udacity [5]. We manually drive the car simultaneously record the images of the left, front and right cameras, paired with steering angles, throttle and speed. The data is then exported to a csv formatted file.

Data Augmentation

According to the Nvidia's paper, random shifts and rotations were applied to original images in order to teach the network to recover from a poor position or orientation [1]. For our implementation, we utilized several image augmentation techniques which perform various operations on an image's pixel values.

Flipping And Translations

Each image has a 50% chance of being augmented, as defined in our batch_generator function. If it is decided that the image should be augmented, we will randomly pick one picture from center, left and right images. Then the image will be flipped horizontally and the sign of the steering angle will switched. Lastly, we also perform Affine transformation on the images and adjusting the steering angle at a rate of 0.002 per pixel shifted.

Network Architecture

Our CNN model emulates Nvidia's CNN architecture with our several unique modifications.

Specifically, the input dimensions are $[66, 200, 3]$ YUV images. The images will be normalized with hard-coded weights to reduce training time required. Note that the ELU will be used as the activation function for all layers.

The inputs are then convolve with $5 \times 5 \times 24$ filters, then with $5 \times 5 \times 36$ filters, then with $5 \times 5 \times 48$ filters, each with stride 2.

The next 2 layers are to convolve with $3 \times 3 \times 64$ filters twice, with stride 2 for both layers.

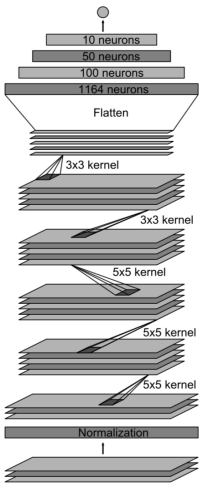

Figure 2: Nvidia's CNN

Next, we implement dropout with probability 0.5, into a fully connected layer with 100 neurons, to 50, to 10 and finally to a single output for the steering angle.

Training

We first load the data from the CSV formatted file from Udacity's simulator with load() from train.py:

```
def load():
    df = pd.read_csv(os.path.join("./data",'driving_log_new.csv'))
    X = df[['center_camera','left_camera','right_camera']].values
    y = df['steer_angle'].values
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=20,
        random_state=101)
    return X_train, X_test, y_train, y_test
```

*Note: The original data is shuffled with a 80/20 split between training and testing.

Afterwards, we initialize the CNN model with make_cnn_model() using the architecture defined in the previous section:

```
def make_cnn_model():
    model = Sequential()
    model.add(Lambda(lambda x: x/127.5-1.0, input_shape=INPUT_SHAPE))
    model.add(Conv2D(24, 5, 5, activation = 'elu', subsample=(2,2)))
    model.add(Conv2D(36, 5, 5, activation='elu', subsample=(2, 2)))
    model.add(Conv2D(48, 5, 5, activation='elu', subsample=(2, 2)))
    model.add(Conv2D(64, 3, 3, activation='elu'))
    model.add(Conv2D(64, 3, 3, activation='elu'))
    model.add(Dropout(0.5))
    model.add(Flatten())
    model.add(Dense(100, activation='elu'))
    model.add(Dense(50, activation='elu'))
    model.add(Dense(10, activation='elu'))
    model.add(Dense(1))
    model.summary()
    return model
```

Lastly, we train the model with train_model():

```
def train_model(model, X_train, X_test, y_train, y_test):
    cp = ModelCheckpoint('model-{epoch:03d}.h5', monitor = 'val_loss', verbose = 0,
        save_best_only=True, mode='auto')
    model.compile(loss='mean_squared_error', optimizer=Adam(lr=1.0e-4))
    model.fit_generator(batch_generator('./data', X_train, y_train, 50, True),
                        20000,
                        15,
                        max_q_size=1,
                        validation_data=batch_generator('./data', X_test,
                            y_test, 50, False),
                        nb_val_samples=len(X_test),
                        callbacks=[cp],
                        verbose=1)
```

Where ModelCheckpoint() prints the validation loss of the current best model, model.compile() optimizes MSE loss via Adam with learning rate 0.0001, and model.fit_generator() generates and trains batches in parallel.

Definitions and justifications for specific techniques

Affine Transformation

This is the general formula for Affine transfomration:

$$\begin{bmatrix} x_i' \\ y_i' \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix} \cdot \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

Since we only want to shift the images, we want:

$$\begin{bmatrix} x_i' \\ y_i' \end{bmatrix} = \begin{bmatrix} 1 & 0 & c \\ 0 & 1 & f \end{bmatrix} \cdot \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}$$

, where c is the horizontal shift amount and f is the vertical shift amount. Specifically, we set the range of x to be some number between 0 to 50 pixels and range of y to be 0 to 20 pixels.

MSE

MSE stands for mean squared error, which is used as the loss function to evaluate the model's accuracy:

$$\text{loss} = l(\hat{y}_i, y_i) = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 \tag{1}$$

, where y_i is predicted steering angle and $\hat{y}_i$ is actual steering angle for the ith image.

ELU activation

ELU stands for exponential linear unit. We chose ELU as our activation function as nature of the positive gradient solves the vanishing gradient problem. Further, we chose ELU over RELU because ELU has a positive gradient for $x < 0$, which reduces time for training as the mean activation value is close to zero [2]. Finally, we chose ELU simply because it is a smooth function, which gives rise to smooth derivatives, while alternatives such as leaky-RELU do not.

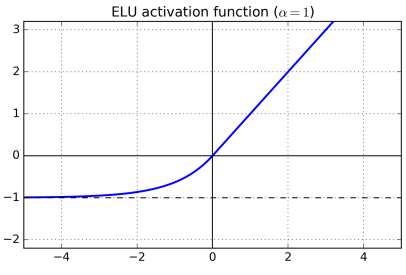

Figure 3: ELU activation function

Dropout

Dropout is as a simple method to regulate and prevent overfitting. This is achieved by randomly selecting neurons in each layer and temporarily suspend their incoming and out-going values during training (see figure 4a). Therefore, a network with n neurons can have potentially 2^n "thinned" neural networks. These networks share weights such that the total number of parameters is bounded by $\mathcal{O}(n^2)$ [3]. At test time, a neural network with all neurons is used, where weights of neurons are scaled by the rate of dropout during training. We will use dropout between the last convolution layer and the first fully connected layer with probability 0.5.

(a) "Thinned" neural network during training.

(b) Full neural network during testing.

Figure 4: Dropout

Adam Optimization

Adam stands for adaptive moment estimation, which is one of the fastest optimization technique that use both first and second moment of the gradient.

Specifically, Adam uses a combination of RMSprop and momentum, with a few salient distinctions [4]. Unlike Adagrad, Adam behaves like RMSprop such that it uses different learning rate for each parameter, where all learning rates are more dependent on recent past gradients. This allows Adam to discard gradient history from the extreme past and thus preserve the learning rates before arriving at a convex bowl.Furthermore, integrating momentum in Adam allows it to overcome local optimal such as saddle points, resulting in faster convergence.

However, in the case of this project, we know that our loss function is convex as we are simply computing MSE. That being said, we used Adam because we discovered that default values [4] learning rate = 0.001, $\beta_1 = 0.9$, $\beta_2 = 0.999$, decay = 0 converges quickly for our problem, without any further extensive experimentation with parameters. The only alteration we have is that we changed the learning rate to 0.0001, which improved the rate of convergence. On the other hand, it would be very time consuming had we choose to use mini-batch SGD, and hand tuned the momentum and learning rate.

Car Simulation

A Python Flask Web application server equipped with web sockets is utilized in order to create a feedback loop from our trained model to the simulator. By design, the simulator is automatically set up to connect to a web socket server on port 4567. On successful connection, the simulator constantly sends car simulation information to the server. Specifically, information such as the current steering angle,throttle, speed, and center image are relayed from the simulator to the server.

After receiving new data, the server relays the center image to the model for a new steering angle prediction and calculates the throttle by the following equation:

$$1 - (\text{steering angle})^2 - (\text{current speed}/\text{max speed})^2$$

Upon experimentation, we found that this equation produces good estimations for the throttle. Specifically, when the car is turning, we would like to slow down. To achieve that, we subtract the steering angle from 1. We also squared the steering angle in order to keep the signs consistent.

We would also like the car to travel as quickly as possible. By dividing current speed over max speed, we get a small number when the car is slow, and subtracting 1 from it yields a large number for throttle. Squaring that number also encourages the car to approach maximum speed when possible.

Combining steering angle and speed, we achieve a desirable behavior such that throttle has a large value when the car just exits from a turn onto a straight road, and small or negative value when approaching a turn with high speed.

Afterwards, the server sends back two inputs back to the simulator, the predicted steering angle, and the new throttle.

Results

Upon experimenting with different sets of training data, we found that the model trained on 5 laps of desert track and 5 laps of jungle track had the best performance.

(a) 1 lap of desert track

(b) 5 laps of desert track and 5 laps of jungle track

(c) 10 laps of desert track and 10 laps of jungle track

Figure 5: Model was trained on desert track for 1 lap.

As shown in figure 5b, the minimum validation loss is at epoch 20, at 0.377. We have tested the performance of the model via the simulator. This particular model was able to complete the basic desert track with a 0% crash rate. It was also able to run the advanced jungle track for several laps with minor issues seldom occurring.

Examples

Here are some examples of the behavior of the model depending on the given training data.

(a) Crashed on desert track.

(b) Crashed on jungle track.

Figure 6: Model was trained on desert track for 1 lap.

(a) Crashed on jungle track, 5 laps training

(b) Crashed on jungle track, 10 laps training

Figure 7: Model was trained on desert and Jungle track for 5 and 10 laps respectively.

References

[1] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin and Arvind Krishnamurthy. TVM: End-to-End Optimization Stack for Deep Learning, 2018; arXiv:1802.04799.

[2] Djork-Arné Clevert, Thomas Unterthiner and Sepp Hochreiter. Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs), 2015; arXiv:1511.07289.

[3] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting, 2014; The Journal of Machine Learning Research, 15(1), pp 1929-1958.

[4] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization, 2014; arXiv:1412.6980.

[5] Udacity. Udacity's car simulator; https://github.com/udacity/self-driving-car-sim

[6] Udacity. Udacity's starter code https://github.com/udacity/CarND-Behavioral-Cloning-P3

[7] Prasad Pai. Data Augmentation Techniques in CNN using Tensorflow https://medium.com/ymedialabs-innovation/data-augmentation-techniques-in-cnn-using-tensorflow-371ae43d5be9