Behavioral Cloning

The goals / steps of this project are the following:

- Use the simulator to collect data of good driving behavior
- Design, train and validate a model that predicts a steering angle from image data
- Use the model to drive the vehicle autonomously around the first track in the simulator. The vehicle should remain on the road for an entire loop around the track.

Step 1:

The initial step involved collecting the data – I used the 'sample driving data' provided in the Project Resources (before using the 'sample driving data', I manually collected the data by driving one lap both clockwise and counter-clockwise, but the output result was not satisfactory, so ended up using the provided dataset.).

Step 2:

To check the model performance, initially I used the center images and steering angles, but the model failed during 'curves', the car would go off-track.

To solve this issue, I took the center, left and right images. For the steering angles, I modified the steering angles, keeping steering angle for center images as it is, adding some bias to the original steering angle to obtain angle for the left image and subtracting some bias from the original angle to obtain the angle for the right image.

```
center_angle = float(batch_sample[3])
left_angle = center_angle + correction
right_angle = center_angle - correction
```

For memory efficiency, I implemented 'generators'.

```
def generator(samples, batch_size):
    num_samples = len(samples)
    while 1: # Loop forever so the generator never terminates
        sklearn.utils.shuffle(samples)
        for offset in range(0, num_samples, batch_size):
            batch_samples = samples[offset:offset+batch_size]
            images = []
            angles = []
            for batch_sample in batch_samples:
                #for center image
                name_c = 'data/IMG/'+ batch_sample[0].split('/')[-1]
                center_image = mpimg.imread(name_c)
                center_angle = float(batch_sample[3])
                images.append(center_image)
                angles.append(center_angle)
                #for left image
                name_l = 'data/IMG/'+ batch_sample[1].split('/')[-1]
                left_image = mpimg.imread(name_l)
                images.append(left_image)
                left_angle = center_angle + correction
                angles.append(left_angle)
                #for right image
                name_r = 'data/IMG/'+ batch_sample[2].split('/')[-1]
                right_image = mpimg.imread(name_r)
                images.append(right_image)
                right_angle = center_angle - correction
                angles.append(right_angle)
            X_train = np.array(images)
            y_train = np.array(angles)
            yield sklearn.utils.shuffle(X_train, y_train)
```

Step 3:

For the model architecture, there was no need to create a dense model having a lot of parameters. The images had fewer features and had to be preprocessed to make them suitable for predicting steering angles.

I used keras.layers.Cropping2D to crop the images focusing only on the road.

```
#normalizing and cropping
model.add(Lambda(lambda x:x/255.0 - 0.5, input_shape=(160,320,3)))
model.add(Cropping2D(cropping=((70,25), (0,0))))
```

For the model, I used two Convolutional Layers with **relu** activation, followed by MaxPooling Layers and Dropout and finally three Fully Connected Layers.

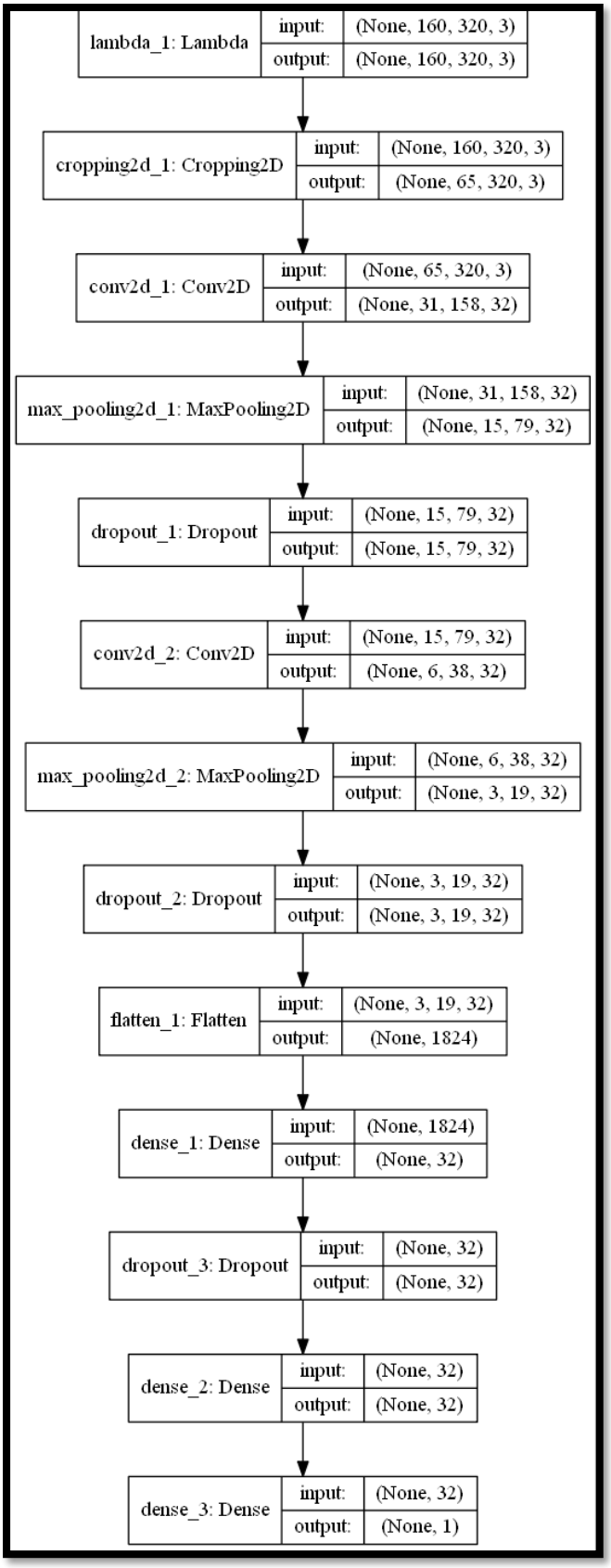
To limit the parameters, I used a 5x5 kernel size for Convolutional Layer, with a stride of 2 and 'valid' padding, keeping the number of filters to 32 for all the two convolutional layers. I maintained the same 32 filters for the two Fully Connected Layers and 1 layer for the final. After testing the model, I added Dropout layers to prevent overfitting.

```
#model
model = Sequential()

#normalizing and cropping
model.add(Lambda(lambda x:x/255.0 - 0.5, input_shape=(160,320,3)))
model.add(Cropping2D(cropping=((70,25), (0,0))))

#layer 1
model.add(Conv2D(32, (5,5), strides=2, activation='relu'))
model.add(MaxPooling2D())
model.add(Dropout(rate=0.3))

#layer 2
model.add(Conv2D(32, (5,5), strides=2, activation='relu'))
model.add(MaxPooling2D())
model.add(Dropout(rate=0.3))

model.add(Flatten())

#fc 1
model.add(Dense(32, activation='relu'))
model.add(Dropout(rate=0.3))
#fc 2
model.add(Dense(32, activation='relu'))
#fc 3
model.add(Dense(1))

model.summary()
```

To visualize the model architecture, I used '*plot_model*' from '*keras.utils.vis_utils*'.

```
#to visualize the model architecture
plot_model(model, to_file='model_plot.png', show_shapes=True, show_layer_names=True)
```

The Model Architecture indicating the input and output shapes:


lambda_1: Lambda
input: (None, 160, 320, 3)
output: (None, 160, 320, 3)
cropping2d_1: Cropping2D
input: (None, 160, 320, 3)
output: (None, 65, 320, 3)
conv2d_1: Conv2D
input: (None, 65, 320, 3)
output: (None, 31, 158, 32)
max_pooling2d_1: MaxPooling2D
input: (None, 31, 158, 32)
output: (None, 15, 79, 32)
dropout_1: Dropout
input: (None, 15, 79, 32)
output: (None, 15, 79, 32)
conv2d_2: Conv2D
input: (None, 15, 79, 32)
output: (None, 6, 38, 32)
max_pooling2d_2: MaxPooling2D
input: (None, 6, 38, 32)
output: (None, 3, 19, 32)
dropout_2: Dropout
input: (None, 3, 19, 32)
output: (None, 3, 19, 32)
flatten_1: Flatten
input: (None, 3, 19, 32)
output: (None, 1824)
dense_1: Dense
input: (None, 1824)
output: (None, 32)
dropout_3: Dropout
input: (None, 32)
output: (None, 32)
dense_2: Dense
input: (None, 32)
output: (None, 32)
dense_3: Dense
input: (None, 32)
output: (None, 1)

(For the accurate result, the model architecture was modified a lot – the first model I implemented had two levels of 3 Conv Layers – 3x3 kernel, strides=2, padding='same', MaxPooling Layer, Dropout Layer, Flatten and FC Layers. This gave almost a million parameters. I tried implementing other architectures which I have used for other projects, but nothing worked. The result was not accurate and plus the computational time was an issue. Then I went for a 5x5 kernel and default attribute values for Conv Layers and limited the layers to 2 and added two FC Layers and an output layer – this gave promising results. I had to modify the number of filters.)

Step 4:

For saving the model, I used callbacks.

```
#checkpoint
checkpoint = ModelCheckpoint("model.h5", monitor="val_loss", mode="min", save_best_only = True, verbose=1)

#for early stopping : if the val_loss is not improving
earlystop = EarlyStopping(monitor = 'val_loss', min_delta = 0, patience = 3, verbose = 1, restore_best_weights = True)

#for reducing the Learning Rate : if the val_loss remains constant or is not improving
reduce_lr = ReduceLROnPlateau(monitor = 'val_loss', factor = 0.2, patience = 3, verbose = 1, min_delta = 0.0001)

callbacks = [earlystop, checkpoint, reduce_lr]
```

For saving the best model, the 'validation loss' was monitored and if there was an improvement between subsequent epochs, then the model having the best/improved 'validation loss' was saved.

If the 'validation loss' is not improving, then there is no point of training the model for the remaining epochs. To save time, 'EarlyStopping' was used, which would stop the model if the 'validation loss' is not improving for 3 subsequent epochs (patience=3).

A model often benefits from reducing the learning rate. So, I have used 'ReduceLROnPlateau', which reduces the learning rate if the 'validation loss' is not improving.

For the optimizer, I used 'ADAM' and 'MSE' for the loss function.

```
model.compile(loss='mse', optimizer='adam')
hh = model.fit_generator(train_generator, steps_per_epoch=np.ceil(len(train_samples)/batch_size),
                    validation_data=validation_generator,validation_steps=np.ceil(len(validation_samples)/batch_size),
                    callbacks=callbacks, epochs=10, verbose=1)
```

Result and Future Improvements:

Using the above-mentioned architecture and the provided dataset, the model worked pretty well in predicting the steering angles even for the curves.

For the most part, the car remained at the center of the lane. There was no going off-track and no touching the ledges.

There were a few times when the car was close to the lane boundaries.

There is room for improvement. A better preprocessing or a better model would do the trick. More data would also be a benefit provided the overfitting aspect is taken care of.

(track 2 is in progress)