Behavioral Cloning

Fabrizio Frigeni, 2017.09.26

Specifications

The goal of this project is to teach a car to steer in order to stay on the road while driving along a given track. It is a supervised learning task, where the input is the image from the front camera of the car, the model is a neural network, and the output is a steering angle.

Implementation

The following implementation steps are described next:

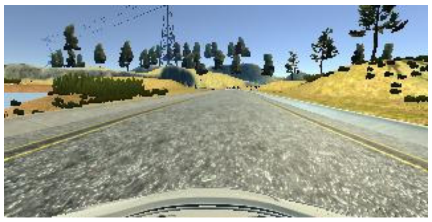
- Data loading and preprocessing
- Model architecture
- Model training
- Test drive

Data loading and preprocessing

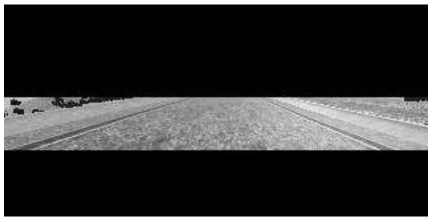
The data is collected while driving along the track, and consists essentially of two sets: front camera images and corresponding steering angles. While other data is also available (throttle, side images...) we do not use them in this project.

Front camera images

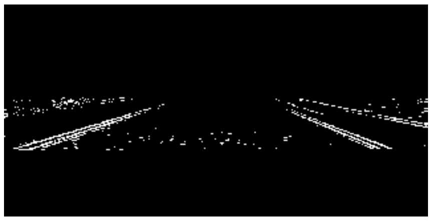
The images taken from the front of the car are in 320x160 RGB format:



We think that color information is not relevant and that the top and bottom parts of the images are of little use. So the first pre-processing steps are to flatten the image to one gray channel and crop to the central region of interest. This step will also help to decrease computational load further on.



In order to recognize the boundaries of the road we can examine the image gradients and filter according to their absolute values along individual axes, their overall magnitude, and their direction. After fine tuning the thresholds we can clearly highlight the lateral road boundaries:



The final image being fed into the neural network is a binary matrix of size 320x40. This should simplify and speed up the learning task considerably.

The whole pre-processing function is defined in the model.py task for training and imported also by the drive.py task for real-time predictions:

```
image_array = pre_process(np.asarray(image))
```

Steering angle

The steering angle is a continuous value in the interval [-1,1] and should ideally be a continuous function of time with limited jerk for a smooth driving experience.

However, considering that the only requirement of this project is to stay on the road, i.e. within the bounding lines, without any specification on how the steering angle changes over time, we decided to simplify the problem and discretize the angle to only three values: -1 (turn left), 0 (no turn), +1 (turn right). In other words, we turn a regression problem into a classification problem, which is usually much easier to train. A finer discretization could have been chosen, but three classes are enough to pass the test.

The following line of code ensures that all values within the (-0.5,+0.5) interval are reduced to zero, all values larger than 0.5 are seen as 1 and all values smaller than -0.5 are seen as -1. In order to build target classes in Keras and train with categorical cross-entropy loss, we need to shift the values to {0,1,2}. That explains the final +1 in the expression.

```
center_angle = np.clip(np.trunc(center_angle*2.),-1,1)+1
```

On the other hand, at prediction time we need to revert the classes values to their original steering angles, so we added the following line of code to the drive.py task:

```
steering_angle = (np.argmax(prediction)-1.)/4.
```

The steering prediction essentially turns into a bang-bang controller, driving always straight and steering only when getting close enough to a side line. In order to reduce the strength of the steering we reduced the final output value dividing it by 4.

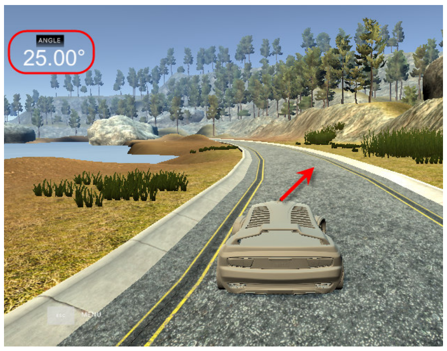
Model architecture

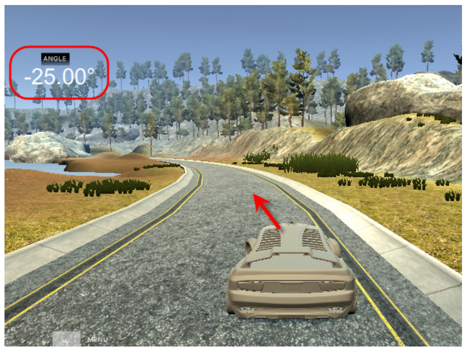
Since we deal with images at the input and discrete classes at the output, we designed the model with two convolutional layers at the bottom and two fully connected layers at the top, capped by a softmax layer for classification.

All layers include ReLU non-linear activation functions. Convolutions are all done with (3x3) kernel size, the first layer with 32 filters and the second with 64. Pooling layers are added to downscale by a factor of 2. Each convolutional layer also has a 50% dropout to reduce overfitting.

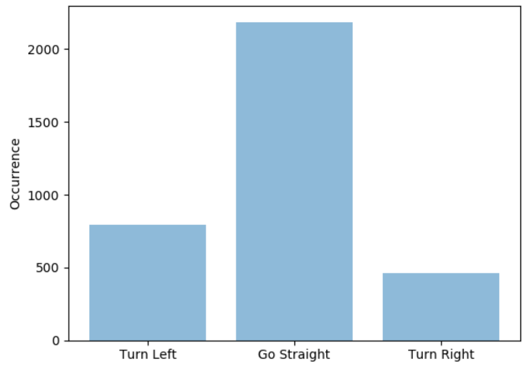
Training

Training data was collected while manually driving the car around the track. Since our discretized controller will mainly deal with non-zero actions only when getting close to a side line we spent most of the training time starting from a position close to a line and driving away from it.





At the end we collected over 3400 examples with the majority being classified as “go straight” and the rest as either ”turn left” or “turn right”, keeping in mind that all the (-0.5,+0.5) data goes under the “go straight” umbrella to avoid continuous oscillations during test drives.



Since left turns appear more often than right turns we also decided to randomly flip the training images (and their target labels) to balance the dataset.

```
if (np.random.random()>0.5):
        center_image = np.fliplr(center_image)
        center_angle = -center_angle
```

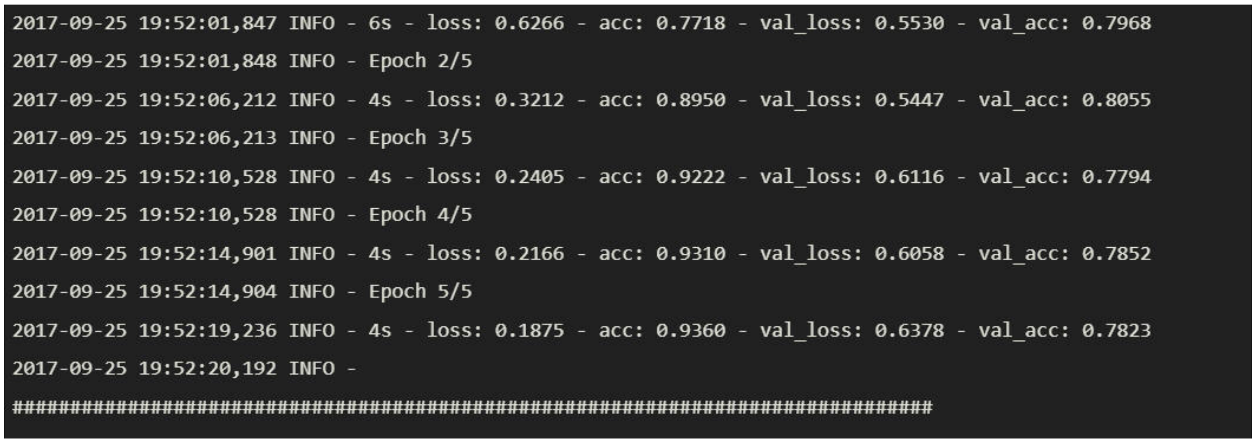
The training of the network is executed by an Adam optimizer, which improves stabilization by adapting the learning rate for each parameter of the system, based on the behavior of the first and second moment of their gradients.

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

Since we are dealing with a classification problem we use the categorical cross-entropy loss function and evaluate the accuracy on the training and validation set. The latter is extracted from the former, by taking its last 20% of data. Note that the training set is initially shuffled.

```
model.fit(X_train, y_train, verbose=2, epochs=5, validation_split=0.2)
```

Five epochs were enough to reach a good accuracy on the training set (93%) and validation set (78%). When adding more epochs we observed that the validation accuracy started to decrease, which is a clear sign of overfitting. The reason is most likely that 3k examples are not many for the capacity of the chosen network.



```
2017-09-25 19:52:01,847 INFO - 6s - loss: 0.6266 - acc: 0.7718 - val_loss: 0.5530 - val_acc: 0.7968
2017-09-25 19:52:01,848 INFO - Epoch 2/5
2017-09-25 19:52:06,212 INFO - 4s - loss: 0.3212 - acc: 0.8950 - val_loss: 0.5447 - val_acc: 0.8055
2017-09-25 19:52:06,213 INFO - Epoch 3/5
2017-09-25 19:52:10,528 INFO - 4s - loss: 0.2405 - acc: 0.9222 - val_loss: 0.6116 - val_acc: 0.7794
2017-09-25 19:52:10,528 INFO - Epoch 4/5
2017-09-25 19:52:14,901 INFO - 4s - loss: 0.2166 - acc: 0.9310 - val_loss: 0.6058 - val_acc: 0.7852
2017-09-25 19:52:14,904 INFO - Epoch 5/5
2017-09-25 19:52:19,236 INFO - 4s - loss: 0.1875 - acc: 0.9360 - val_loss: 0.6378 - val_acc: 0.7823
2017-09-25 19:52:20,192 INFO -
################################################################################
```

While the project specifications require the use of a generator to supply images during training we actually preferred not to use it (although implemented it anyway in the model.py task).

The advantage of the generator is that it reads batches of data from disk only when needed thus avoiding filling up the memory. However, we trained our model on FloydHub and noticed that:

1. Their GPU has more than enough memory to load the whole dataset before training
2. The continuous reading from disk is actually very slow (they don't use SSDs) and makes the whole training much slower than needed

Therefore we decided to avoid the generator completely.

Test drive

A test drive was performed on track 1 and succeeded relatively quickly. The run1.mp4 movie shows a full lap of the track in autonomous mode.

Comments

While the model performed well on track 1, it did not work on track 2. This is not surprising because all training data was collected on track 1 and lighting conditions on the two tracks are quite different.

A much larger dataset with various light conditions should be collected for better generalization.

Using a finer discretization of the output, or even a regression head, would make the drive smoother at the cost of longer training time.