Behavioral Cloning Project

1 Project Goals

1. Use the simulator to **collect driving data**.
2. **Design** and **train** a neural network that will **safely steer** the car throughout track #1 and optionally track #2.
3. Expound the underlying **principles** of the design.
4. Discuss slightly **more productive approaches** (my addition-feel free to skip section).

2 The Model

The core idea behind the model that, in my opinion, is suitable (as an abstract design-details may vary) for autonomous steering and throttle-break control is the ability to "remember" past driving states and captured images. In other words, the neural network should have memory of what happened in the past in the form of some high-dimensional state vector stored away in a layer located mid-way to the top.
It is clear that training a purely feed-forward network will not only probably introduce the need to train forever, but is also a mechanism that memorizes images independently from the training set without "capturing" their evolution in time in relation to the car's motion. In short, it is not logical to make decisions on how to drive a vehicle based solely on what the driver sees in the present without at least considering what happened in the past.

2.1 Network Architecture

The network is essentially comprised of two merged distinct sequential branches containing a *recurrent neural network layer* either directly above the input

layer or the convolution layer. I used *Long Short-Term Memory* (LSTM) for the recurrent layer, primarily because these units are more popular at the moment and I really didn't have the time to thoroughly study the details of their functionality and make comparisons with other types of recurrent neurons. In retrospect, they did their job, but I can't really guarantee that they are an ideal choice. The two branches merge into a dense layer which connects to a second dense layer that yields the output, a vector containing the new values for *steering angle and throttle* . The architecture is depicted in Figure 1.

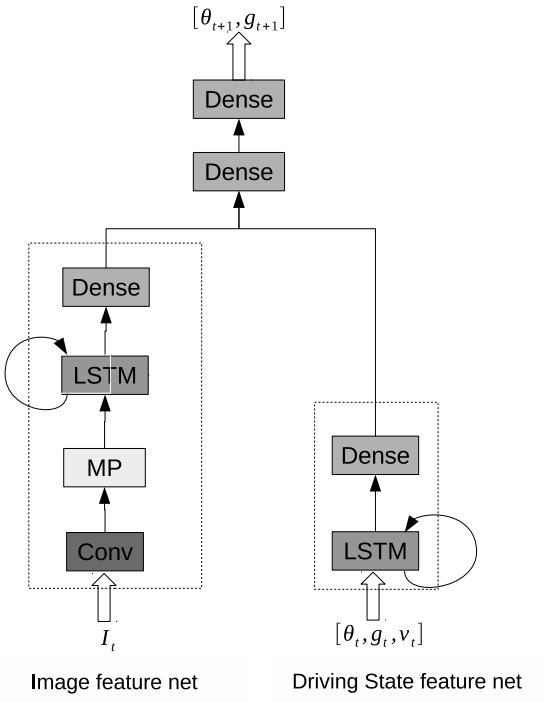

Figure 1: The architecture of the neural network. It comprises two sequential models, both of which contain LSTM layers above the initial feature layers in order to "remember" past states. θ_t is the steering angle t, g_t is throttle and v_t is speed at time t.

2.1.1 Input Preprocessing and Layer Configurations

The input images are standardized by subtracting 128 from all pixels and then dividing by 128 before being fed to the convolutional layer. The convolution filter has size 12×12 with an output depth of 32 and a stride of 3 in both vertical and horizontal directions. In the *image feature branch*, the convolution layer is followed by a max-pool layer comprised of 8×8 - dimensional filters with a stride of 2.

Wiring the convolutional layer to the LSTM layer. Interfacing the LSTM layer to the convolutional/max-pool layer is a bit tricky. The LSTM neuron is designed to accept (usually temporal)sequences of one-dimensional tensor input (vectors). To "plug-in" the convolutional output to a single LSTM neuron, one has to configure the LSTM for input sequences of 1 term only and each term being a vector of dimension equal to the convolution output depth. Unfortunately, configuring the LSTM input size doesn't take us immediately out of the woods. The convolutional layer neurons are 2D arranged in space, while the LSTM layer contains the neurons in a 1D arrangement. It is therefore necessary to r̂eshape the 2D array of max-pooled convolutional outputs to a 1D array of such outputs. Luckily, in the *driving state branch*, the LSTMs simply interface with the network input which is a 3D vector and therefore the only requirement is to configure their output. In both branches, LSTM layers interface with dense layers in the forward dimension without requiring any reshaping.

The two branches eventually merge into a dense layer which, in turn leads into the top dense layer which outputs the new values for steering angle and throttle. All lower-to-upper layer connections following the recurrent layers have *relu* activations excluding LSTM outputs which are *sigmoids* (otherwise the mean squared cost function would "explode") and the top layer which has purely linear activation in order to account for the negative values in steering angles.

3 Training

The training approach is basically what is more-less implied by the exercise: Drive the car around and then train the neural network to reproduce the user input by correlating it with the camera feedback and the previous inputs through a *Mean Square Difference/Error* (MSE) cost function. I used 3-4 distinct sequences of 5 laps from each track in which the car travels either slightly to the right, or in the middle of the road. There was no particular consistency in the respective recordings in terms of the car position, so sometimes is near the right lane and sometimes in the center. Additionally, a single recording with a few "awkward" situations from which the car should be able to recover was recorded. These situations were essentially "picked-up" during test-runs after a certain amount of training (at least 40 epochs).

3.1 Generating training and validation data

Following a few casual training runs of 40-80 epochs, I observed that resizing the image to a size no smaller than 110×160 produce more-less the same descent patterns, so I decided to train at this particular scale, which is roughly $\frac{1}{4}$ of the original size. Although provisions were made (see code) for using grayscale and/or the side images, I eventually decided to use the RGB center image only, primarily because of two reasons: a) Training time and, b) I had a lot of faith in the LSTMs overcoming the need for excessive training and/or additional data.

3.2 Network Structure and Configuration

As shown in figure 1, the entire network comprises **two keras sequential models** merged at the top, each one loosely corresponding to a feature memory on video input and driving.

Video Feature branch. The video feature model at the first layer uses a convolution with filters of size 12×12 and output depth 32, followed by a 2D max-pool layer with filter size 8×8 , which becomes rearranged into a 13×22 array of 32D outputs that get feed into a recurrent layer with 160 LSTM neurons. The last layer of this "branch" is a dense layer with 80 neurons.

1. Input Image
    - Shape:$[110, 160, 3]$
2. 2D Convolution
    - Filter size: 12×12
    - Output depth: 32
    - Strides: $[3, 3]$
    - Layer output shape: $[33, 50, 32]$
3. 2D Max-Pooling
    - Filter size: 8×8
    - Output depth: 32
    - Strides: $[2, 2]$
    - Layer output shape: $[13, 22, 32]$
4. 2D-to-1D Reshaping
    - Layer output shape: $[13 \times 22, 32]$
5. Long-Short-Term Memory (LSTM)
    - Input shape: $[1, 32]$
    - Layer output shape: 160 (sigmoid)
6. Dense Layer
    - Output shape: 80 (relu)

Driving State Feature branch The second branch is essentially a memory layer (LSTMs again) followed by the usual dense layer. In this case, the input is just a 3D vector containing steering angle θ_{t-1}, the throttle g_{t-1} and speed v_{t-1}. Thus, there is no need for a convolution; inputs are fed directly to the recurrent layer.

1. Long-Short-Term Memory (LSTM)
    - Input shape: $[1, 4]$

- Layer output shape: 8 (sigmoid)

2. Dense Layer
   - Layer output shape: 20 (relu)

Merged Model. The merged model receives $80 + 20 = 100$ inputs in total from the two merged branches. Two more dense layers are stacked on the input, with the second delivering the two outputs of the network, which are the new steering angle θ_t and throttle g_t.

1. Dense Layer
   - Input shape: $[1, 100]$
   - Layer outputs: 30 (relu)
2. Dense Layer
   - Layer output shape: 2 (linear)

An observation. I would like to note that, due to the fact that the max-pooled outputs are being reshaped into $13 \times 22 = 286$ 32-deep outputs, my initial though was to add at least 300 LSTM neurons. Interestingly, with incremental training, I noticed that 160 LSTM neurons would produce results faster, without any noticeable difference in performance.

Data generator. I have implemented a generic data generator function to be used for both training and validation. The data generator can be configured to retrieve a list of directories with names having a specific prefix (by default, this prefix is "TRAIN_" and "VALIDATION_" respectively). It then seeks and opens the *driving_log.csv* file inside these directories. For each image file and respective driving state entries contained in *driving_log.csv* , an input and tensor are prepared and stored inside respective batch lists, which, by default are 64-entries long. If a batch is incomplete at the end of a file, which is a very possible scenario, the remaining entries will be filled-in from the next file in the list. Given that we are trying to train with memory, this could be perceived as a problem, since it stitches driving states and images that will be most probably unrelated to each other. On first glance, this maybe detrimental for the training process, but on second

thought it is only a relative small portion of the data, but most importantly, it could be thought of as some sort of **"natural" drop-out** because it will "encourage" the LSTMs to be more robust to abrupt changes in the driving state or to initializations from various situations on the track. Finally, each epoch is comprises 2042 batches, which proved to be a reasonable figure in practice, because it allows for frequent monitoring of the training progress with autonomous test-runs.

3.3 Training in practice

Overfitting in this network is peculiar. It presents itself as the convergence to a set of outputs that are very close to the dominant set of user-outputs (for instance, casual cruising in slightly curved road with approximately constant steering angle and throttle). From this training point onwards, convergence is "stuck" in a plateau and no improvement occurs, since it is preferable, for instance, to "sacrifice" behavior in sharp turns in favor of behavior that appears more in the training set, such as causal driving in straight lines.

On dropouts . From what I read dropouts have become a significant tool against overfitting and become more and more popular. Hinton's interpretation of the way dropouts work (i.e., force other neurons toi compensate for the neuros who don't receive updates by backpropagation) is probably close to the truth. My concern is that it is a method that has no formal mathematical explanation (so far) as to why it works and I would rather avoid using it, if I can find a substitute which has a similar effect, but also can be interpreted in loose mathematical terms. This does not mean that dropout is bad, I am just in favor of using it when all else fails. In this case, here are a couple of arguments:

- The LSTMs are making sure that the network does not overfit each image, because they "demand" to fit the driving history as well. So drop-outs are not necessary in that regard.
- Most importantly, the network overfits because of a largely **ill-posed cost function**. In particular, the cost function is a MSE on the user input. This is a very "weak" measure of fitness, because it expects from the network to infer the *actual intention* of the user, which is to keep

the car in the middle of the road. This is probably a much larger task that requires a trully deep network, much more data and time at our disposal. However, I am discussing a potential solution to this problem in section 5 at the end of the report.

Coping with overfitting by increemental training. In order to avoid transition from a useful set of weights to an overfitted one, my strategy was to train the network incrementally in small sequences of 20-40 epochs and do test-runs at the end of each. If the car appears to be doing well for a bit, then I train further for a few epochs and retry a test-run using, a) Fixed throttle and, b) Network output throttle. If the results are encouraging (i.e., good driving for a distance and reasonably good steering indicating intent to stay on the road), then the network is kept and improved for a few epochs. If the car drives well under the new weights, then we keep the solution. Otherwise, if things got a bit worse with training, more data are generated and we train again with the augmented dataset and repeat the refining process. This is a challenging fine-tuning process and, although it didn't take much time to find something that works (in the ball-park of 150 epochs), I was relieved to see that the LSTMs did the job!

4 Running the network in Track #1

By default the code in *drive.py* will use the steering angle output of the trained model to drive the car at a constant throttle value of 0.15. This level of actuation gives enough time to the network to safely compensate for transients in its motion that may potentially throw it off the road. I have observed that cruising is safe for many laps and **the driving patterns are different every single time**! This means that the car will make a turn in various different ways that, sometimes look very different. This is very interesting, but it is also indicative of the fact that, to a certain extent, the network has actually resolved the *intent* in the user inputs rather than just associating them with the input images. Figure 2 illustrates how the car steers in the very same turn (second after the bridge) in the first and second lap respectively and it becomes obvious that the driving patterns are different.

(a) Lap #1 turn. (b) Lap #2 turn.

Figure 2: Screenshots of the car at the second turn after the bridge. Clearly, the car is taking the turn in a different way than the previous lap, as a result of accumulated "driving memories".

Throttle control. The network is **trained to control both steering angle and throttle** and the resulting driving behavior is generally acceptable, but in certain occasions it may lead to an *unstable oscillating motion* of the car from one side of the road to the other, because at the current speed levels, the steering angle is not rectified quickly enough by the network to keep it on track. The reason for this rare but not unlikely instability lies primarily with the lack of training examples which are very hard to provide, even with the simulator. The safe workaround was to simply fix the throttle to 0.13. I have been running the network at this throttle levels for a great number of laps and observed no "accidents", with the exception of touching the fence twice in two different locations while turning. I should note again, that **motion is far from deterministic in the input images but rather it depends on the overall driving state history of the car**. Thus, although *extremely-super unlikely*, the car may actually get out of track or touch the fence, in which case, please, just push the "break" key and re-enter the track from the beginning. That should do it.

Safe cruise mode. To run the network at the *default (and safe)* throttle levels (0.13), simply type:

```
python drive.py model.h5t
```

Fast cruise mode. To use a somewhat faster, fixed throttle level (0.2), type:

```
python drive.py model.h5t --throttle raceme
```

Auto cruise mode. Finally to use automatic throttle (network output), type:

```
python drive.py model.h5t --throttle auto
```

5 A more robust cost function

In this section, I would like to briefly give a few arguments as to why the cost function for raw behavioral cloning many not be the best fitness measure. The reason for this is that the network is forced to "infer" through training the actual intention of the driver/user to move the car to the middle of the road as it realizes this position through the convolutional feature layer. For something like that to be implicitly inferred through training is very-very hard and potentially dangerous, as overfitting can eventually lead to a very good network-driver which may not react well in rare on one hand, yet unforeseen on the other, circumstances.
A simple workaround (and potentially much more robust) would be to define a cost function on the car's displacement from its current position and model the predicted outputs using the Ackerman steering model [1]. Provided that we can log the exact *location and heading* (aka "pose") of the car (probably by means of accurate GPS),we can formulate quadratic error terms which relate the network's prediction with the actual pose of the car following user input at time t as follows:

$$C\left(\theta_{t-1}, g_{t-1}\right) = \left(y_t - f\left(\theta_{t-1}, g_{t-1}\right)\right)^2 \tag{1}$$

where θ_t and g_t are the network outputs at time t and y_t is the vector containing the pose of the vehicle relative to its position and heading at time $t-1$, as recorded during the training runs. Function f returns the *Ackerman* predictions for the vehicle heading and position (relative to y_{t-1}). Therefore, to train the network, we now simply need to propagate the gradient of f into the optimizer and follow the usual routine.

The benefits. The benefits of employing the cost function in equation 1 are significant because they directly link the driver's intent with the network's output through the Ackerman steering model. Thus, training now does not have to implicitly infer this relationship through the input images (i.e., convolutional features and respective memory cells) and at the same time relate the actual raw inputs with this intent at all times. This is very difficult and it probably needs tons of training data to make sure that the network actually "got it". On the other hand, with the cost function of eq. 1, training does not have to do this (i.e., try to implicitly infer what the user is trying to do), but rather gets it in a silver platter by the simple kinematic model; thus the "bulk" of training efforts now focus completely in correlating the desired relative heading and position in the training data with the resulting input images. This should give a set of weights which, I am conjecturing, will be much more robust to untrained actions and within significantly fewer training runs.

References

[1] Jürgen Ackermann and Wolfgang Sienel. Robust yaw damping of cars with front and rear wheel steering. *IEEE Transactions on Control Systems Technology*, 1(1):15–20, 1993.