END TO END LEARNING FOR A DRIVING SIMULATOR

V.F. ALEXEEV, A.I. STARAVOITAU, G.A. PISKUN, D.V. LIKHACHEUSKI

Belarusian state university of informatics and radioelectronics, Republic of Belarus

Submitted 1 March 2018

Abstract. Convolutional network approach is utilized for training an end-to-end model that would let a car drive itself around the track in a driving simulator by predicting steering angles based on the simulated camera data.

Keywords: machine learning, computer vision, convolutional network, end-to-end learning, Keras.

Doklady BGUIR. 2018, Vol. 112, No. 2, pp. 85-91
End to end learning for a driving simulator
V.F. Alexeev, A.I. Staravoitau, G.A. Piskun, D.V. Likhacheuski

Introduction

Convolutional networks became fairly popular recently and are mainly used for pattern recognition these days after their breakthrough with ImageNet [1]. Their area of application, however, is not limited to image classification, as described in the Nvidia paper [2], where authors trained a convolutional network based model that managed to learn the entire processing pipeline needed to steer an automobile. The authors based their research on a Defence Advanced Research Projects Agency (DARPA) seedling project known as DARPA Autonomous Vehicle (DAVE), where training data included video from two cameras coupled with left and right steering commands from a human operator [3].

A similar, although very simplified approach is applied here in order to train a convolutions based model that would ultimately learn to steer a car in a driving simulator. The fact that a simulator is used here obviously adds additional constraints on the data, therefore making it only relevant in a context of the said virtual driving simulator; nevertheless it would be a good illustration of various techniques for designing, training and regularizing machine learning model based on convolutional network, and of a range of tasks that could be achieved with a relatively simple architecture.

Dataset

Data has been collected using a slightly modified open-source driving simulator recently released by Udacity [4]. The version used in this project allows retrieving simulated data from three front-facing cameras that record frames sequences as if they were mounted on the vehicle. It also allows recording vehicle steering angles for collecting training data and supplying steering angles in a vehicle control pipeline for testing.

The described driving simulator has two different tracks. One of them was used for collecting training data, and the other one – never seen by the model – as a substitute for test set. Essentially, both tracks allow the same set of controls and provide the same data, albeit different landscapes and overall mapping, variations of curvature and slope, etc. Driving simulator also comes with a set of tools for supplying driving instructions into the simulator in real time from a trained Keras model.

The driving simulator would save frames from three front-facing «cameras», recording data from the car's point of view, as well as various driving statistics like throttle, speed and steering angle. Camera data is going to be used as model input, and then model is expected to predict the steering angle in the [–1; 1] range.

A dataset containing approximately one hour worth of driving data was collected around one of the given tracks. This would contain both driving in «smooth» mode (staying right in the middle of the road for the whole lap), and «recovery» mode (letting the car drive off centre and then interfering to steer it back in the middle). While the «smooth» mode allows the model to learn the overal high-level strategy when the car is in the middle of the road, «recovery» mode would embrace edge cases for when the car needs to manoeuvre and there is a risk of going off track. This would teach the model how to recover from a poor position or orientation.

Just as one would expect, resulting dataset was extremely unbalanced and had a lot of examples with steering angles close to 0 (e. g. when the wheel is «at rest» and not steering while driving in a straight line). Small angles were then rejected prior augmenting the data, thinking that those < 0.05 angles that would appear in the dataset as a result of using side cameras wouldn't perturb distribution of angles significantly.

A designated random sampling was applied, which ensured that the data is as balanced across steering angles as possible. This process included splitting steering angles into *n* bins and using at most 200 frames for each bin. Histogram of the resulting dataset looks fairly balanced across most «popular» steering angles (Fig. 1).

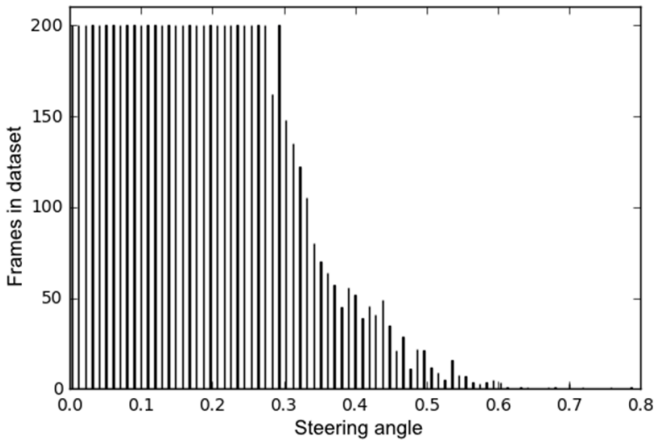

Fig. 1. Training dataset histogram

Mind that dataset balancing will be applied across absolute values, thinking that after applying augmentation horizontal flip both positive and negative steering angles will be used for each frame anyway. The dataset was balanced before augmentation, so there was no need to reject small angles afterwards. One shouldn't expect augmentation to change distribution significantly for additional balancing to be necessary.

Ideally, dataset should be balanced across all steering angles, of course. However, this would require a significant amount of training data, and should not be the bottleneck for this particular model performance on the test set, which essentially another track, previously unseen by the model. As noted earlier, number of examples in each bin is capped at 200, so that model does not become biased towards going straight ahead (e.g. predicting a steering angle of 0), instead of staying on the road.

The number 200 was chosen empirically, analysing collected data across various steering angles, and mainly making sure that examples are equally distributed across most popular angles that model is expected to predict in most cases, i.e. in the [–0.3; 0.3] range.

Data augmentation

After balancing approximately one hour worth of driving data the collected dataset contained 7,698 sample frames, which most likely wouldn’t be enough for the model to generalize well. However, there are a couple of augmentation tricks that can extend a dataset significantly.

Along with each sample simulator provides frames from 3 camera positions: left, center and right. Although only central camera will be used while driving, one can still use left and right cameras data during training after applying steering angle correction, increasing number of examples by a factor of 3. Steering correction is applied when using frames from side cameras, and after trying a couple of different values ±0.25 seemed to perform best, not causing the car to oscillate too much and still providing expected recovery.

For every batch half of the frames is flipped horizontally and has the sign of the steering angle changed (or, more specifically, each frame is flipped with a probability of 0.5), thus yet increasing number of examples by a factor of 2. This is a very common technique, and could also be used in classification tasks, not only in regression. Quite often one either expects a classification model to predict the same class for a flipped input image, or as some other specific class. Thus, flipping image horizontally (and, in some tasks, vertically) could also increase number of training classification examples by a factor of 2 (or even 4).

Image is then cropped during pre-processing removing top and bottom parts, which are insignificant for training objective; furthermore, choosing the amount of frame to crop at random should increase the ability of the model to generalize. Analysing the frames retrieved from driving simulator, those are 160×320 images, and the hood of the car takes bottom 20 pixels, the road ends exactly 60 pixels from the top. Random variation is applied to these parameters as a part of augmentation pipeline, for the model to generalize better in cases when the car is going up or down the hills. If the frames contained landscape or car hood, it could make it way harder for the model to generalize: for instance, it could start picking up wrong patterns, not crucial for predicting steering angles.

Additionally, a random vertical «shadow» is added to frames by decreasing brightness of a frame slice, hoping to make the model invariant to actual shadows on the road. To provide some more reasoning regarding why shadows were added, there are some distinct features of the road that might not be present throughout whole training track, but that would be nevertheless very useful as training input for the model to generalize well. One of such features is a shadow of surrounding landscape, most likely it won't be well represented in the balanced and normalized training. After all, trying to balance a dataset across every distinct feature would be quite a challenging task; therefore, some of those features are simulated manually, as a part of data augmentation pipeline applied to the training dataset.

After applying said augmentation pipeline to the dataset each frame is pre-processed by cropping top and bottom of the image and resizing to a shape the model expects (vector of shape (32, 128, 3), RGB pixel intensities of a 32×128 image). The resizing operation also takes care of scaling pixel values to [0, 1]. As mentioned earlier, insignificant parts of frames are cropped out, parts that would otherwise encourage the model to learn false patterns.

Data preparation step is essentially trying to cut out any insignificant features from the input data (think hood of the car, or surrounding landscape); or, alternatively, make insignificant features that can't be simply removed represented in wider number of training examples (think shadows), so that their presence does not seem useful for the model while learning, and so that it prevents the model from finding false patterns in data.

To make a better sense of it, let's consider an example of a single recorded frame sample that results in 16 training samples by using frames from all three cameras and applying aforementioned augmentation pipeline. This single example includes three original images retrieved from the simulator, taken with three cameras located on the front of the vehicle (left, center and right), which are then transformed into 16 augmented and pre-processed training samples (Fig. 2).

Fig. 2. Original frames of a single dataset sample

Augmentation pipeline is applied using a Keras generator, which lets applying it in real-time on CPU while the most computationally expensive operation of backpropagation is performed on GPU (Fig. 3).

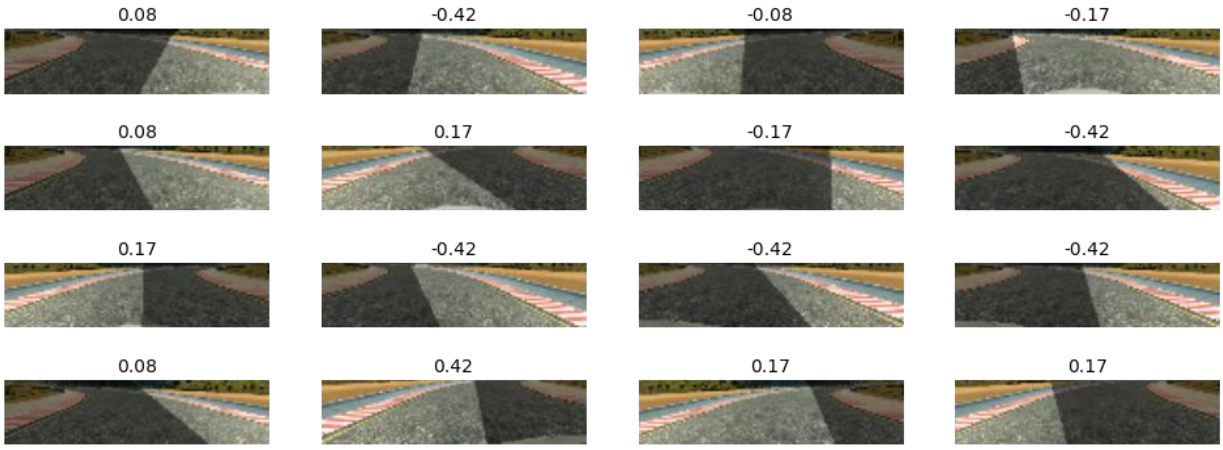

Fig. 3. Augmented and pre-processed frames based on the sample in Fig. 2

Model

Model architecture evolved from the model described in the Nvidia paper [2, p. 5], which was gradually simplified and optimized while making sure it still performed well on both tracks. It was clear one wouldn't need that complicated model, as the dataset in question is way simpler and much more constrained than the one Nvidia team had to deal with when running their model.

Initially, model had 5 convolutional layers and 3 fully connected layers (Fig. 4). It then was gradually simplified by continuously removing layers and decreasing dimensionality (and therefore number of trainable parameters), until the model's performance started decreasing. In fact, there was a boost in performance on the test track at some point, as the model started generalising better with simpler architecture. Eventually model converged to a fairly simple architecture with 3 convolutional layers and 3 fully connected layers of the following dimensions.

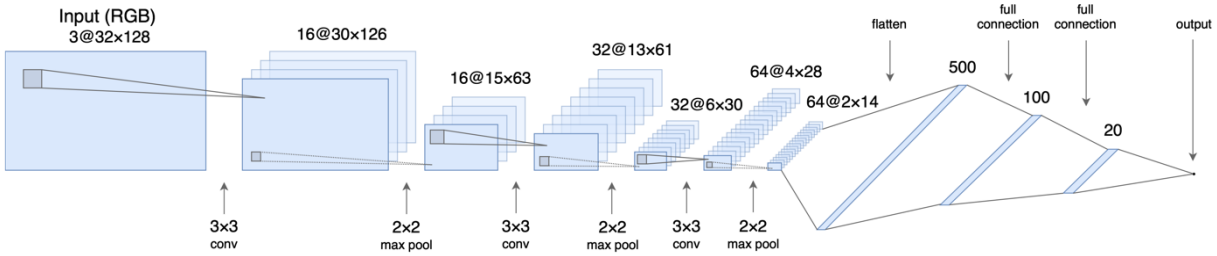

Fig. 4. Model architecture

The technique for architecture and training is similar to the one described in Nvidia paper [2, p. 4], training the weights of the network to minimize the mean squared error between the steering angle output by the network and the angle recorded during data collection in the driving simulator. The network consists of 6 layers: 3 convolutional layers and 3 fully connected layers, all 6 have fairly low dimensionality, especially compared to the original model in Nvidia paper [2, p. 5].

The convolutional layers were designed to perform feature extraction and were chosen empirically through a series of experiments that varied layer configurations, eventually landing on using a non-strided convolution with a 3×3 kernel size.

In general, convolutional networks were inspired by biological processes [8] in which the connectivity pattern between neurons is inspired by the organization of the animal visual cortex. Individual cortical neurons respond to stimuli only in a restricted region of the visual field known as the receptive field. The receptive fields of different neurons partially overlap such that they cover the entire visual field. In a neural network based model convolutional layers apply a convolution operation to the input, passing the result to the next layer, e. g. convolution emulates the response of an individual neuron to visual stimuli.

When it comes to visualizing learned weights, one should typically expect a convolutional layer to contain filters that can detect very basic pixel patterns, like edges and lines. These basic filters are then used by subsequent layers as building bricks to construct detectors of more complicated patterns and figures.

Three convolutional layers with three fully connected layers lead to an output angle value which is essentially a steering command for the simulator. Neurons in a fully connected layer have connections to

all activations in the previous layer, as seen in regular neural networks. Their activations can hence be computed with a matrix multiplication followed by a bias offset.

The fully connected layers are designed to function as a controller for steering, but by training the system end-to-end, it is not possible to make a clean break between which parts of the network function primarily as feature extractor and which serve as controller.

Rectified linear unit (ReLU) incorporating rectifier [6] is used as activation function in the convolutional layers. With strong biological motivations and mathematical justifications rectifier activation function has been used in convolutional networks more effectively than the widely used logistic sigmoid and yield equal or better performance than its more practical counterpart, the hyperbolic tangent.

Another important concept of convolutional networks is pooling [7], which is a form of non-linear down-sampling. There are several non-linear functions to implement pooling; the most common one was used here (max pooling), which seems to perform best in practice [7, p. 8], and which is conveniently implemented in Keras. It partitions the input image into a set of non-overlapping rectangles and, for each such subregion, outputs the maximum. The intuition is that the exact location of a feature is less important than its rough location relative to other features. The pooling layer serves to progressively reduce the spatial size of the representation, to reduce the number of parameters and amount of computation in the network, and hence to also control overfitting. As pooling operation also provides another form of translation invariance, a pooling layer has been added after each convolutional layer.

The pooling layer operates independently on every depth slice of the input and resizes it spatially. The most common form of a pooling layer is used, with filters of size 2×2 applied with a stride of 2. This setup effectively downsamples slices in the input at every depth by 2 along both width and height, discarding 75 % of the activations.

This model can be very briefly encoded in Python with Keras (Fig. 5).

```
from keras import models
from keras.layers import core, convolutional, pooling

   model = models.Sequential()
   model.add(convolutional.Convolution2D(16, 3, 3, input_shape=(32, 128, 3),
activation='relu'))
   model.add(pooling.MaxPooling2D(pool_size=(2, 2)))
   model.add(convolutional.Convolution2D(32, 3, 3, activation='relu'))
   model.add(pooling.MaxPooling2D(pool_size=(2, 2)))
   model.add(convolutional.Convolution2D(64, 3, 3, activation='relu'))
   model.add(pooling.MaxPooling2D(pool_size=(2, 2)))
   model.add(core.Flatten())
   model.add(core.Dense(500, activation='relu'))
   model.add(core.Dense(100, activation='relu'))
   model.add(core.Dense(20, activation='relu'))
   model.add(core.Dense(1))
```

Fig. 5. Listing. Model code

Keras with TensorFlow backend was used here, which is considered a high-level machine learning framework, but has proven to be more than enough for this particular task. It implements all required algorithms and functions, allows various triaining and regularization techniques, as well as works alongside toolset supplied with virtual driving simulator in question.

Regularization

This particular problem should not require any sophisticated regularization given the constraints on the data that a virtual driving simulator implies. Data augmentation and dropout helped the model to generalize enough to drive on a previously unseen track.

It's important to remember to get the model to train nicely and overfit first, then start regularizing. Using this approach, one could first keep training the model till it starts overfitting the training dataset: in this particular case, it was performing better and better on the first training track, where the data was collected, and was performing rather poorly on the test track.

As discussed previously, data augmentation is a common technique for increasing number of training examples without collecting more data. It is also useful for regularizing, as augmentation also helps improving useful feature representation. There is usually a significant amount of useless and misleading information in the training data, and as arbitrary model would try to detect any patterns it could

find, it is a crucial step to try and minimize presence of those misleading pieces of information, or make it hard to draw any conclusions from it by flooding it randomly in a reasonable number of training examples. Quite often data augmentation can have a significant impact on the model performance, especially when you have a general idea of which features could be useful, and which may only confuse the model. In this case it's fairly obvious that a trained model should predict a steering angle based only on the road curvature, and that it shouldn't draw any conclusions from shadows on the road, position of the hood of the car or surrounding landscape.

Dropout, on the other hand, is also a popular regularization technique that works amazingly well, and will most likely drastically improve generalization of the model. Normally one may only want to apply dropout to fully connected layers, as shared weights in convolutional layers are good regularizers themselves. There was a slight improvement in performance when using a dropout on dense layers, thus dropout has been added on 2 out of 3 dense layers to prevent overfitting, and the model proved to generalize quite well.

Training

The model was trained using a stochastic gradient-based optimizer with a learning rate of 10^{-4} and mean squared error as a loss function. 20 % of the training data was used for validation (which means that only used 6.158 out of 7.698 frames were used for training), and the model seems to perform quite well after training for 20 epochs.

A variation of early stopping was used, which involved monitoring training and validation losses, saving model weights after every training epoch. Validation error must then be analyzed and the training process must be stopped manually once validation error stops improving over a reasonable number of epochs. Eventually, the weights from the epoch with the lowest validation loss were used, as noted earlier this would typically happen after 20–30 epochs.

Both formal and informal metrics were used for benchmarking model performance. Formal would include minimizing calculated loss during the training process. Informal means letting the car drive around both tracks (training and test one) by using model's steering instructions, and checking if it goes off the road or behaves unnaturally: for instance, starts oscillating or ignores track turns in some specific conditions.

Keras implementation of Adam optimizer [5, p. 1] was used, a method for efficient stochastic optimization that only requires first-order gradients with little memory requirement. The method computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients; the name Adam is derived from adaptive moment estimation.

Results

The described model performs well, and the car manages to drive just fine on both tracks pretty much endlessly, e. g. without driving off the track without any external intervention. It rarely goes off the middle of the road both on the track it was trained on and a previously unseen track.

Clearly, this is a very basic example of end-to-end learning for self-driving cars. But nevertheless, even considering all limitations of training and validating solely on a virtual driving simulator, it should give an understanding of what a relatively simple convolutional network is capable of. In this case, the model managed to learn the entire task of road following solely «by example», e. g. by looking at a relatively small dataset of the ground truth driving data. Most importantly, it didn't expect any explicit instructions or decomposition of tasks, like road detection, path planning and controls.

References

1. Krizhevsky A., Sutskever I., Hinton G.E. Imagenet classification with deep convolutional neural networks. Advances in Neural Information Processing Systems. Curran Associates, Inc., 2012. P. 1097–1105.
2. End to end learning for self-driving cars / M. Bojarski [et al.]. 2016. arXiv preprint arXiv:1604.07316.
3. Net-Scale Technologies, Inc. Autonomous off-road vehicle control using end-to-end learning, July 2004. Final technical report [Electronic resource]. – URL: http://net-scale.com/doc/net-scale-dave-report.pdf (access date: 01.03.2018).
4. Udacity Inc., Udacity's Self-Driving Car Simulator, GitHub repository [Electronic resource]. – URL: https://github.com/udacity/self-driving-car-sim (access date: 01.03.2018).
5. Diederik P.K., Jimmy B.A. A Method for Stochastic Optimization. 2017. arXiv preprint arXiv:1412.6980v9.
6. Glorot X., Bordes A., Bengio Y. Deep sparse rectifier neural networks. AISTATS [Electronic resource]. – URL: http://jmlr.org/proceedings/papers/v15/glorot11a/glorot11a.pdf (access date: 01.03.2018).

7. Scherer D., Müller A.C., Behnke S. Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition // 20th International Conference «Artificial Neural Networks (ICANN)». Thessaloniki, Greece, 2010. P. 92–101.
8. Subject independent facial expression recognition with robust face detection using a convolutional neural network / Matusugu M. [et al.]. Neural Networks. 2003. № 16 (5). P. 555–559.

Information about the authors

Alexeev V.F., PhD, associate professor, associate professor of information and computer systems design department of Belarusian state university of informatics and radioelectronics.

Staravoitau A.I., master student of information and computer systems design department of Belarusian state university of informatics and radioelectronics.

Piskun G.A., PhD, associate professor, associate professor of information and computer systems design department of Belarusian state university of informatics and radioelectronics.

Likhacheuski D.V., PhD, dean of the computer design department of the Belarusian state university of informatics and radioelectronics.

Address for correspondence

220013, Republic of Belarus,
Minsk, P. Brovka st., 6,
Belarusian state university of informatics and radioelectronics
tel. +44-794-297-06-86;
e-mail: alex.staravoitau@gmail.com
Staravoitau Aliaksei Iharavich