Behavioural Cloning Project Writeup

March 21, 2018

1 Behavioral Cloning Project Writeup

1.1 For the Udacity Self Driving Car Nanodegree

1.2 By Harry Turner on 21st March

---

2 Introduction

Deep convolutional neural networks offer improved performance over existing traditional robotic techniques when controlling an autonomous car. This project explores the application of deep networks to this problem, and describes the approach taken to develop such a network.

3 The Approach

The approach taken in this project is to start simply, and just get something working. From there, complexity is added in bit by bit to improve the performance. This is a useful way of approaching this task, since it isn't clear from the outset what the solution should look like, beyond the fact that it should probably require a convolutional neural network. The approach is described in the order in which the activities were carried out, each separated into its own section.

3.0.1 Exploration of the Simulator

The first step in any project is to explore the dataset, in this case, the simulator. Once the simulator was working, I spent some time driving around the track, and made notes of what the neural network would have to deal with. For example, I noted areas where the tracks had no edges, places with sharp bends, and distracting features on the side of the road that may degrade the performance of the network. An example of a piece of track with no edges is shown below.

3.0.2 Collect Training Data

After I'd spent some time exploring the simulator, I started collecting training data. I defer the discussion of how I chose what data to generate, to a later section dedicated to the training process, called Training. At this stage, it is sufficient to note that I generated the data using the record feature of the simulator, which saved images and steering angles to a folder on my local system.

no-edge.png

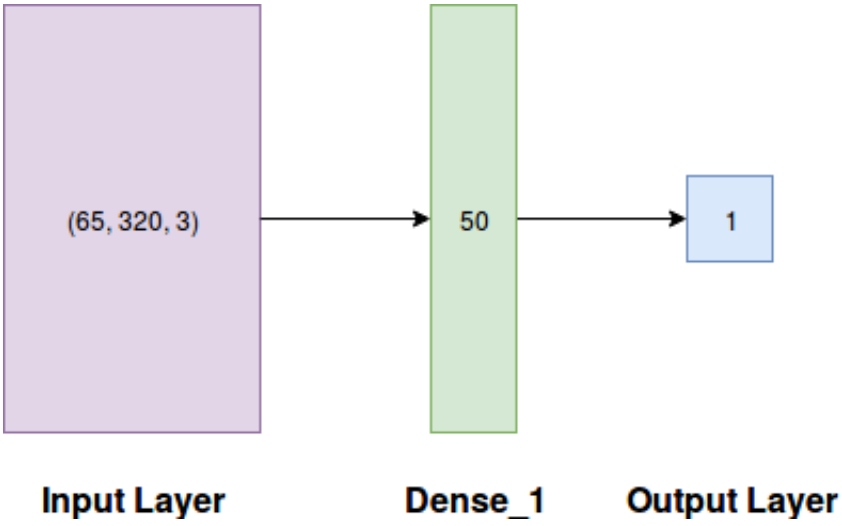
3.0.3 Get Something Working

Once I had some training data, my first goal was to simply get something working. The main reason for this was to confirm that the environment was setup appropriately, and that my network was both able to learn from the data, and control the car (albeit badly). The first network architecture I used was a simple vanilla neural network, with one hidden layer with 50 neurons, built using the Keras framework running on top of Tensorflow. The inputs to the network are flattened and then pass through the network, with the output consisting of just one neuron. Note that no activation function was applied to the output. The Keras code to accomplish this is shown in the code block below, a visual depiction of the architecture is shown in the image below the code block. Note also that the input layer consists of a cropped image, which will be explained shortly.

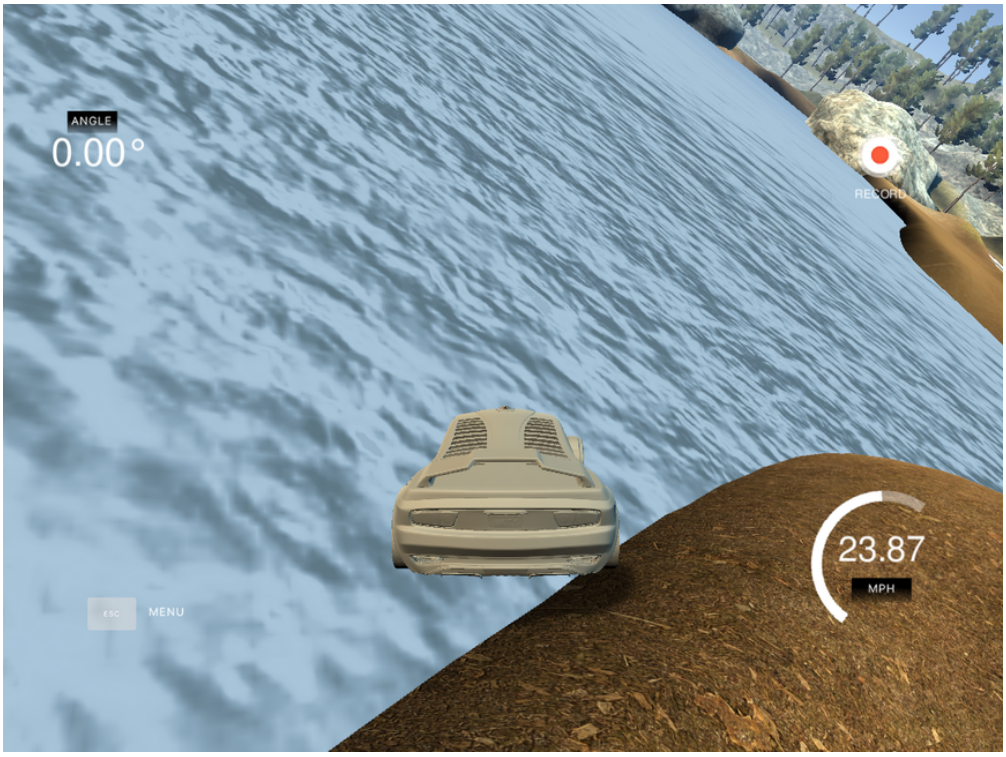
```
In [ ]: model = Sequential()
        model.add(Flatten())
        model.add(Dense(50))
        model.add(Dense(1))
```

This network was quick to train, and when tested in the simulator the steering rapidly bounced back and forth, resulting in the car slowly juddering forward. Eventually, the car turned off the track and drove into the lake. This was what I expected, the network did a terrible job at controlling the car, but it proved that the setup worked, and that all I needed to do was improve the network.

The next five sections discuss the improvements made to the network.



vanilla-network.png



into-lake.png

3.0.4 Improvements: Normalisation and Cropping

A clue to the first method for improving the network was given by observing the validation loss during training. The validation loss was huge! The reason for this is that the inputs to the network were raw pixel values, which ranged from 1 to 255, this meant that the output of the network was large, and the network had to spend a lot of time and effort bringing the weights down to improve the output. The network could avoid this step if I normalised the images before they were input to the network. This actually resulted in a faster training time too.

I implemented normalisation using a Keras Lambda layer, where I scaled each pixel value down to between 0 and 1, and also centred each value about 0, so that the final pixel values ranged between -0.5 and 0.5. The code to achieve this is shown in the code block below.

```
In [ ]: model.add(Lambda(lambda x: x / 255.0 - 0.5, input_shape=(160,320,3)))
```

I also used a Keras crop layer to crop the input images to just the lower half. This is because the road is all the network really needs to navigate, the top half of the image is mostly unnecessary and distracting for the network. The code to accomplish this is shown in the code block below.

```
In [ ]: model.add(Cropping2D(cropping=((70, 25), (0, 0))))
```

3.0.5 Improvements: Artificial Training Set Expansion

Another simple technique that can greatly improve network performance is to use more training data by way of artificially expanding the dataset. For this project, I could have simply driven the car around the track a few more times, however in practice, obtaining more data can be costly. To expand the dataset, I flipped each image, and changed the sign of the steering angle. The effect of this was as if I had driven around the track the other way. The effect of this improvement was to increase training times, but to also improve the accuracy.

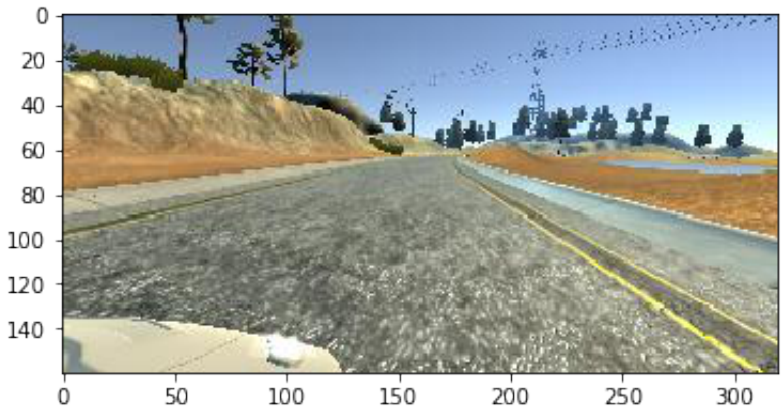
This functionality is implemented in a dedicated data object, which is explained later, and can be found in the file helper.py. However, the short piece of code that actually performs the flipping is shown in the code block below.

```
In [ ]: # For every image and measurement...
        for image, measurement in zip(images, measurements):

            # Append the original image, and a flipped image.
            aug_images.append(image)
            aug_images.append(cv.flip(image, 1))

            # Append the original measurement, and a flipped measurement.
            aug_measurements.append(measurement)
            aug_measurements.append(measurement*-1)
```

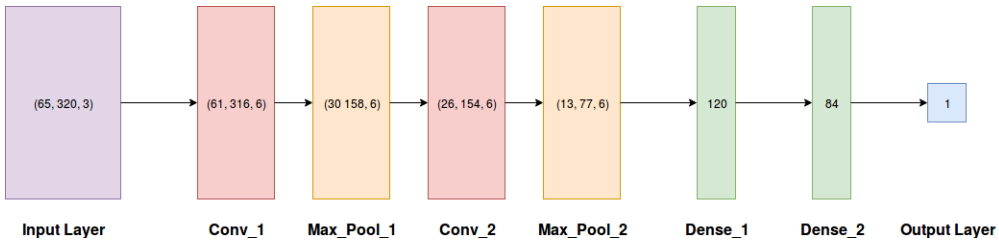
An example of a flipped image is shown below. It isn't obvious that the image is flipped, except by realising that the lake is on the wrong side!

3.0.6 Improvements: LeNet

The next improvement was to replace the simple network with a more sophisticated one. From my previous experience, I know that the LeNet architecture offers sophistication, whilst not being



reversed.png

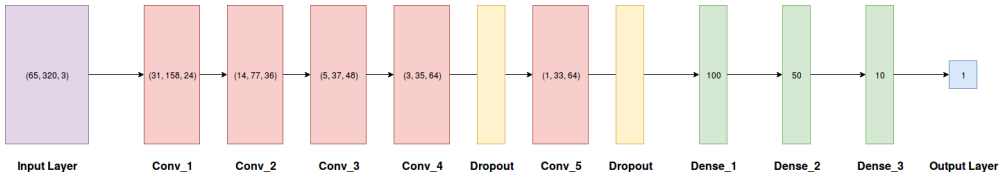


lenet-network.png

too large which would increase training times dramatically. I therefore implemented the LeNet architecture with the input layer consisting of a three channel input image, and the final output layer consisting of only one neuron, rather than the usual ten. I also removed the softmax layer. The Keras code to implement this network is shown in the code block. A visual depiction of the architecture is shown in the image below the code block.

```
In [ ]: model = Sequential()
        model.add(Convolution2D(6, (5, 5), activation="relu"))
        model.add(MaxPooling2D())
        model.add(Convolution2D(6, (5, 5), activation="relu"))
        model.add(MaxPooling2D())
        model.add(Flatten())
        model.add(Dense(120))
        model.add(Dense(84))
        model.add(Dense(1))
```

The result of implementing this network was to dramatically improve the performance of the car. Although training times were increased to about 10 minutes per epoch, the final network was able to navigate the car all the way to the bridge before turning off the road and ending up in the lake again. This was a substantial improvement and suggested that increasing the sophistication of the network still further may improve the performance even more.



nvidia-network.png

3.0.7 Improvements: NVIDIA

Using the LeNet architecture over the vanilla one layer network substantially increased the performance. To see if I could get even more performance, I used a very sophisticated network from NVIDIA, which has been used to control self-driving cars in the real world. This network is much larger, with five convolutional layers and three fully connected layers. This gives the network much more powerful decision making capability, which may allow the car to navigate the track completely. The cost of using such a network is that the training times are increased again, and the network could badly over fit, therefore I added dropout to the last two convolutional layers. The Keras code to implement this network is shown in the code block below. A depiction of the architecture is shown in the image below the code block.

```
In [ ]: model = Sequential()
        model.add(Conv2D(24, (5, 5), activation="relu", strides=(2,2)))
        model.add(Conv2D(36, (5, 5), activation="relu", strides=(2,2)))
        model.add(Conv2D(48, (5, 5), activation="relu", strides=(2,2)))
        model.add(Conv2D(64, (3, 3), activation="relu"))
        model.add(Dropout(0.75))
        model.add(Conv2D(64, (3, 3), activation="relu"))
        model.add(Dropout(0.75))
        model.add(Flatten())
        model.add(Dense(100))
        model.add(Dense(50))
        model.add(Dense(10))
        model.add(Dense(1))
```

After implementing this architecture, the car's performance was sufficient to allow it to reach the bridge and cross it; however during a tight turn after the bridge, with no edges on the road, the car ran off the road into the sand.

At this point, I was happy that my network was sophisticated enough to manage the track, but I realised that I needed some more training data targeted to this particular turn. Specifically, I needed to record data of the car about to run off the track, and steering sharply to get back on course.

3.0.8 Improvements: Targeted Data

The final improvement that enabled my car to navigate the track completely was to record more training data in situations where the car was struggling to manoeuvre properly. In training mode, I drove the car up to the edge of the track in those difficult areas, and turned sharply to get back on course. I did this multiple times for each difficult area, and then augmented this new data by