Behavioral Cloning Project

Reflectance Report

Introduction

The objectives of this project are to train a deep neural network so that given front-facing car images of the road, we determine the right steering angle to keep the car on the road. We first use a provided simulator to collect steering angle data with the corresponding image of the road for the steering angles. This data should correspond to good driving behaviour. After, we use these images to build and train a deep neural network and pose this as a regression problem so that the output of the network is the direct steering angle used to control the car to maintain its trajectory on the road. After we train the deep neural network, we will test the performance of this model by plugging this back into the simulator and directly controlling the steering angle of the car. Success is measured by ensuring the car stays on the road for one lap.

Code structure

In this project, we have the following code and file structure. Note that we don't describe or include any utilities required to complete the task as it is tacitly understood that they are not required for understanding the work done here. They are of course included with the repo for completeness:

- `model.py`: Consists of the necessary functions, tools and run-time script to train the model to infer the steering angle. The only requirement is the path to where the training data is. The training data I used is not available in this repo but can be found in the Jupyter notebook that is provided with this repo (more on this later).
- `drive.py`: Consists of the drive script to drive the car in autonomous mode in the simulator. The script has been modified so that we drive at 20 mph and uses Tensorflow 2.0 instead of the separate Keras package (now integrated with TF 2.0).
- `steering.hdf5` containing a trained deep neural network to provide the output steering angle given a front-facing camera image from the simulator.
- `Behavioural_Cloning_Training.ipynb`: Jupyter notebook that downloads the training data, sets up the arguments for training, trains and validates the model, then saves the model for use in the simulator. This uses the `model.py` file to train the network.
- `video.mp4`: A video recording of the simulator car in autonomous mode driving around the track for 1 lap using the trained network previously described.

Quality of Code

Is the code functional?

The `model.py` file is designed to be run in the terminal with command-line arguments to properly train the model. There are default parameters set if you just want to start training this immediately which have been demonstrated to be the acceptable ones for completing the task. The model.py file contains the code for training and saving the convolution neural network. The file shows the pipeline I used for training and validating the model, and it contains comments to explain how the code works.

Is the code usable and readable?

The code follows standard Python PEP8 formatting requirements as well as any docstrings for the functions created to get the job done. The code does use a Python generator to generate image batches so that the entire dataset does not occupy any memory to successfully fold the entire dataset into the training process.

Model Architecture and Training Strategy

Architecture Overview

In order to take advantage of the fact that we have image data, some form of convolutional neural network should be used so that we can find the most optimal filters, weights and biases to automatically extract the right features that are ideal for use in prediction. Some combination of convolutional layers, nonlinear activation layers, then combining this with fully-connected layers for the classification should be used. After much iteration and being inspired by an architecture similar to NVIDIA's architecture for this problem (see here), the final architecture for the network is shown below:

Layer	Details
Normalisation Layer	Normalises the pixels to the range `[-1,1]`
Cropping 2D	Removes the top 70 rows and bottom 20 rows of the image
Convolutional 2D	Kernel size: 5 x 5, strides: 2 x 2, valid padding, 24 filters
ELU	
Convolutional 2D	Kernel size: 5 x 5, strides: 2 x 2, valid padding, 36 filters
ELU	
Convolutional 2D	Kernel size: 5 x 5, strides: 2 x 2, valid padding, 48 filters
ELU	
Convolutional 2D	Kernel size: 3 x 3, strides: 1 x 1, valid padding, 64 filters
ELU	
Convolutional 2D	Kernel size: 3 x 3, strides: 1 x 1, valid padding, 64 filters
ELU	
Flatten	
Dropout	Drop probability: 0.25
Dense	100 output neurons
ELU	
Dropout	Drop probability: 0.25

Layer	Details
Dense	50 output neurons
ELU	
Dropout	Drop probability: 0.25
Dense	10 output neurons
ELU	
Dropout	Drop probability: 0.25
Dense	1 output neuron - steering angle output

This model is defined at line 186 in `model.py`. We have 2D convolutional layers that progressively become larger in depth as we go deeper. We also have larger kernel sizes with larger strides. The reason why we chose to do this is to avoid using pooling. We want to avoid the destructive nature that pooling has on the network and opt to use a larger kernel size with larger strides to have a smoother effect. We also use ELU activation functions to introduce the nonlinearity in order to bridge the linear and nonlinear gap in semantics. We choose this over ReLU as we don't completely remove negative values and the transition between negative and positive regions is gradual. Also take note that we normalise the incoming images so that their values range from `[-1,1]`, we use a cropping layer to remove the top 70 rows and bottom 20 rows as the sky and scene above the horizon are not useful for helping us steer the car. Finally, we introduce dropout at the dense layers to help mitigate overfitting. This is especially important if we are directly regressing a single output value, and this can be prone to overfitting.

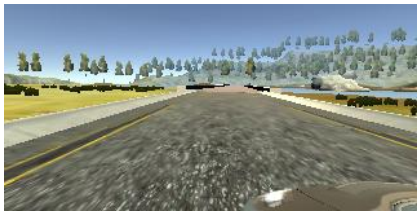
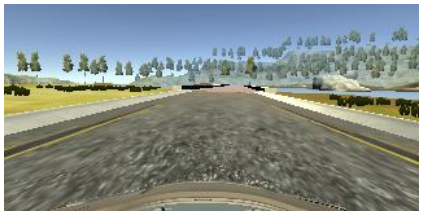
Dataset Creation

The dataset was obtained by using a simulator provided by Udacity where two closed courses were rendered in a virtual environment, along with a virtual vehicle on the track. The objective is to use the virtual car and drive it through the track ourselves so that it can record front-facing camera images, as well as the steering, throttle, when brakes are applied and the velocity as we drive. This serves as training data for our neural network so that we can ultimately predict the required steering angle to keep the car on the track. Take note that the steering angle is the only quantity used in the prediction where the other quantities are calculated and provided automatically when we test out the network.

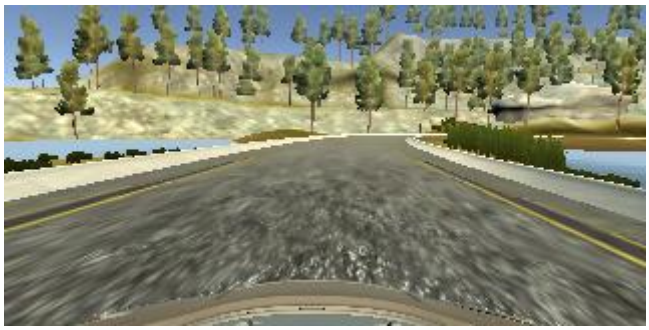
Interestingly, I only used Udacity's sample dataset that they provided instead of collecting my own data. It consists of 9 laps of track one combined with data starting from the side of the road going back to the centre (i.e. a recovery lap). I decided that this was enough for me to proceed onwards. We now have a collection of images and their corresponding steering angles. The simulator not only produces the front-facing camera images for the centre of the car, but they are also produced from the left side of the front and the right side of the front to bootstrap the training. However, the trio all get assigned the same steering angle so it would be prudent to permute the left and right steering angles by a corrective factor to indicate to the network that they need to gravitate towards the centre of the road.

Below is an example of the car driving in the centre of the road, with the corresponding left and right versions of that same scene.

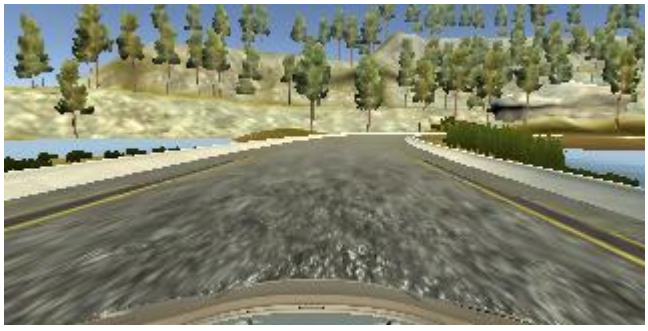
Left	Centre	Right

Left	Centre	Right
		

Also, we have in the dataset examples where the car moves from the left side and right sides of the road back to center so that the vehicle would learn to correct itself. Therefore, the car will eventually align itself to the centre of the road. Below is an example where we start from the right side of the road and eventually converge to the middle.

Right	Centre
	

In addition to the above, a very simple data augmentation was performed where the image was simply mirrored horizontally so that left turns would be right and vice versa. This simple data augmentation can help bootstrap the training to make the model more generalised. This also meant that if we did this mirroring, the corresponding steering angles would be negated.

Right	Flipped
	

Once I obtained all of the data from the simulator, I randomly shuffled the data set and put 80% of the data into a training set, 10% into a validation set and 10% into a test set once the training and validation were complete. The training set was (obviously) used for training the model. The validation set helped determine if the model was over or under fitting. Because the total number of epochs is a hyperparameter, I specified a large number of epochs but created a callback to save the model with the lowest validation error. Therefore, even though we risk overfitting in the end, the point was to specify a high number of epochs so that we will not miss the chance of saving the model that performed the best right before it starts to overfit.

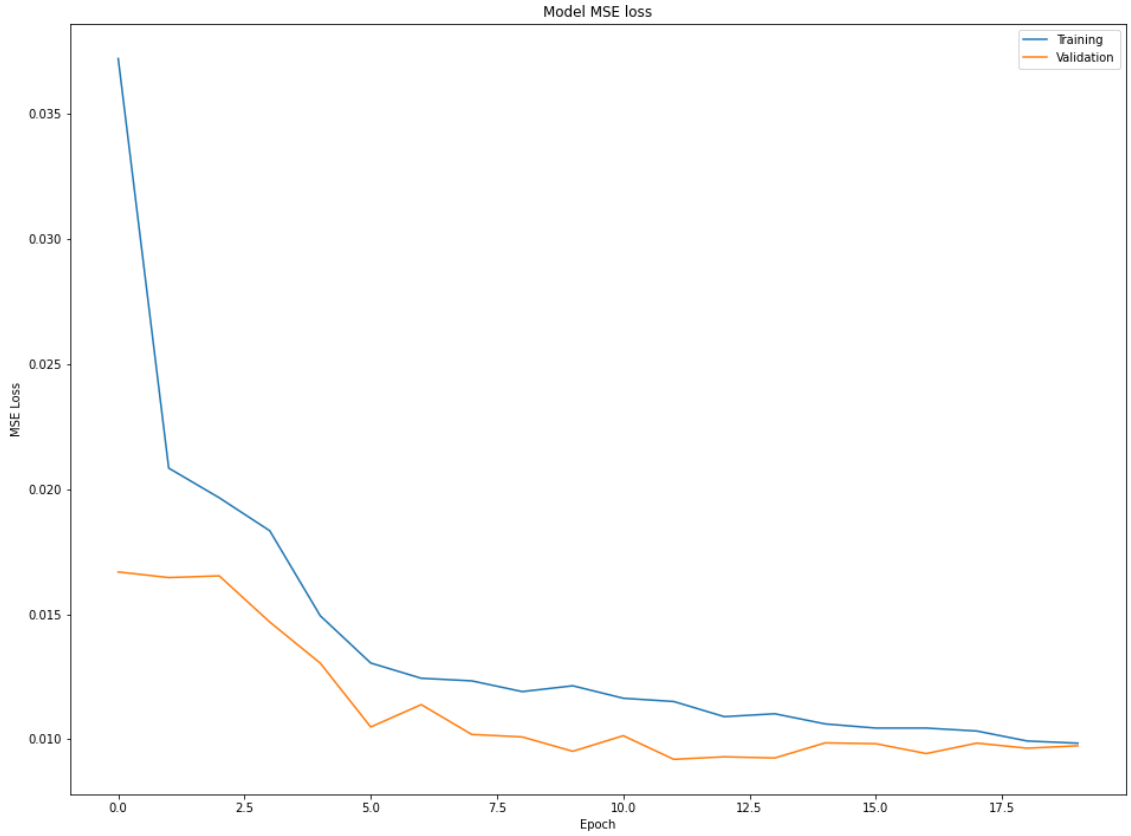
Model Hyperparameters

We describe the hyperparameters for training deep neural network for steering angle output below, as well as justification as to why those parameters were chosen.

- Loss function: Mean Squared Error - A typical loss function used for regression. We are regressing over the steering angle after all.
- Optimiser: I used the Adam optimiser as it is somewhat insensitive to the initial learning rate. Also note that the algorithm is an adaptive one so the insensitivity to the initial learning rate makes sense. However, I used the default learning rate of 0.001.
- Batch size: 32 - This was chosen to leverage between fitting the right number of images in a batch that doesn't exhaust GPU memory and keep the throughput as high as possible versus choosing a batch size that was too high which would artificially decrease the "noise" introduced to make the system generalise better.
- Number of epochs: 20 - Even though this is a very high number, we additionally impose a model checkpoint callback so that the model with the lowest validation error so that we can capture the model the moment before it overfits, thus preventing us from repeated experimentation with the number of epochs.
- Steering correction: 0.2 - For the left and right front-facing camera images, we take the corresponding steering angle that was assigned for the centre image and add this value assigning it to the left image's steering angle, then subtract this value assigning it to to the right image. This is to ensure that we gravitate towards the centre of the lane. 0.2 was the suggested value from the course and I used the same.
- Probability of flipping an image: 1.0 - As stated earlier, we also horizontally flip the images and negate the steering angles to bootstrap the training process. I decided that I wanted every single image to have a flipped counterpart in the training data so I set this to 1.0 to achieve this effect.

Training and Testing Strategy

Using the dataset from the collection strategy previously mentioned, we then used the model hyperparameters above to finally train the model. The training and validation loss curve during training is shown below.



We can see that at about epoch 12 is where we capture the model with the smallest validation error. Anything past epoch 12 we see that there are marginal improvements so choosing the number of epochs to be 20 was a good choice.

As such, the training loss and validation at epoch 31 was 0.0115 and 0.0092 respectively. Once we completed training, we evaluated the error with the test dataset once and only once, and the test set error was 0.0113.

I've attached a separate notebook that connected to a Google Colab Pro instance that demonstrates downloading the dataset, setting up the model and training it. The notebook eventually calls the `main` function which is located on line 236 of `model.py` that performs the whole training process once the proper hyperparameters have been set up and the data downloaded.

Once training completed, the final step was to run the simulator to see how well the car was driving around track one to prove that it can drive autonomously for a single lap. Thankfully, the car stayed on the track for the whole duration, but there were a couple of close calls. At the end of the process, the vehicle is able to drive autonomously around the track without leaving the road.

Proof that the car drove successfully can be found in the `video.mp4` file as part of this submission.