Deep Learning Project Report (Implementing NVIDIA's "End to End Learning for Self-Driving Cars")

Subject: CSE485 (Deep Learning)

18P6713 Youssef Maher Nader Halim

18P6000 Mostafa Lotfy Mostafa

Table of Contents

Videos of Successful Trials

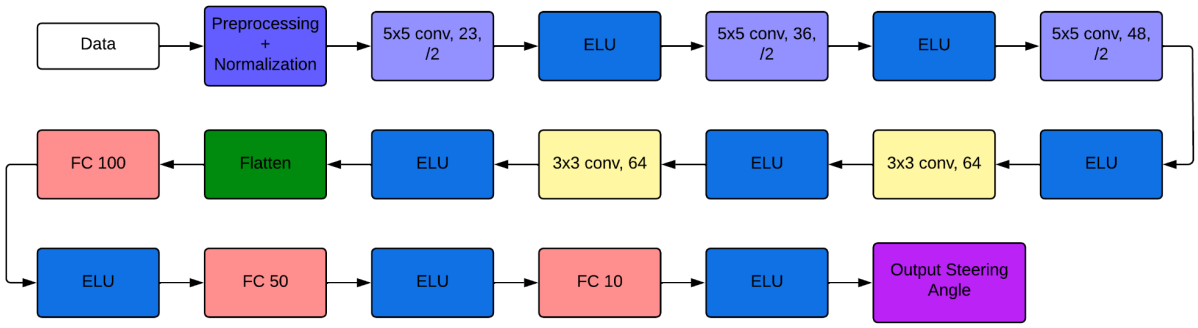
Videos of our model completing one lap successfully on both tracks can be found through this link. https://drive.google.com/drive/folders/1wTubNVjVxjm3bXL2bgWlYDGhILeu_TJF

Introduction

For the Deep Learning course (CSE485), we were tasked with creating a deep learning model capable of predicting the steering angles required to traverse various tracks present in a simulator. To this end, we implemented NVIDIA's proposed architecture from the paper "End to End Learning for Self-Driving Cars) *[1]*. The architecture describes a convolutional neural network that accepts as input, images from a car's front-facing camera and is capable, through training, of predicting the required steering angle for the car to traverse its environment successfully.

Architecture

We used the architecture proposed by NVIDIA in the paper "End to End Learning for Self-Driving Cars) *[1]*. As activation between layers, we used the ELU activation function and added dropout after each fully connected layer for regularization during training. As the problem we have is a regression problem the mean square error was the obvious choice for the loss function as for the optimizer we went with Adam. The input for the model is an image of size 66 * 200 with 3 channels (RGB) which is the same used by NVIDIA. The figure below demonstrates the architecture used.



Creating Training Data

To create the training data the provided simulator was used. For the first track, 4 laps were recorded then the direction of motion was reversed and 4 more laps were recorded. For the second track, a similar methodology was followed however due to the size of the second track we felt it was sufficient to only record two laps in each direction. This data was then compressed into a zip file and uploaded to the Kaggle servers as a dataset. The data was taken at a resolution of 1280*960.

Code

To implement the model, we utilised Kaggle notebooks to work collaboratively on the data and utilize Kaggle's free GPU training time. The code found in the notebook is described in detail within this section.

First, we import some required libraries such as NumPy, matplotlib, os, Keras, cv2, pandas, Scikit Learn, and imgaug. We then specify the paths to the training images and the file driving_log.csv which holds the steering angles and other data for each training image taken in the simulator as well as the path of the image on the device that was used to create the training data. We then create a Pandas dataframe by reading the .csv file. We then get the filenames only of each image in the dataframe and disregard the original path from the original device, this will then be replaced by the new path given by the Kaggle dataset.

The next part of the code was only used briefly as for a previous dataset the images for the first and second track were contained in the same dataset. So in an attempt to train the tracks separately, the data were separated by using the fact that the data for the second track was collected after the data for the first track, so by finding the last image of the first track we only include images taken after the time that picture was taken (this is also helped by the fact that the simulator sets the name of the images to the date and time the picture was taken at). A mask is created using the previously discussed method and a new dataframe is created if needed. (This method, however, became obsolete as in later iterations of the dataset we separated the tracks' images).

We can then visualize the steering angles by plotting a histogram, additionally, we set an upper threshold for the number of samples in each bin, this will later be used to remove data from bins where the sample number is too large. This is to prevent the model from simply predicting a value around 0, as on the first track the driver spends most of the time driving in a straight line.

Next, we create a function to append Kaggle's path to the image filenames and to retrieve the paths of only the centre images (the right and left images were not used as proposed in the NVIDIA paper) and the steering angle of each picture into two NumPy arrays. This function is then called to place the image paths and steering angles into two NumPy arrays. We then call Scikit Learn's train_test_split function to split the data into 80% training and 20% validation, then

plot histograms of both the training and validation data to ensure that they somewhat conform with the original data.

Next, we define a function “augment_image”. This function will be used to add image augmentations to the training data, each data augmentation has a 50% chance to be applied. These augmentations are performed in an attempt to further cover the feature space. The augmentations are as follows:

1. A random ±10% pan along the x-axis and/or y-axis.
2. A random zoom between 1 to 1.2 times
3. A random brightness multiplication between 0.4 and 1.2
4. Randomly flipping the image horizontally (this augmentation also multiples the steering angle by -1 i.e. flips the steering in the other direction)

Next, we define a function “img_preprocess” this function is used to apply the preprocessing proposed in the NVIDIA paper first we crop the image then convert it to the YUV colour space, and then apply gaussian blur and resize the image to the proposed size, finally we multiple all the values of the pixels by 255 to normalize the values.

Next, we define a function “generate_batch” this function is used to generate the batches that will be fed into the convolutional neural network and also applies the image augmentation and preprocessing detailed in the previous two paragraphs.

Next, we define a function “nvidia_model” which is used to construct the model proposed in the NVIDIA paper. This is done by creating a Keras sequential model and then adding the layers as described in the architecture section and then compiling it. With the model created we add a checkpoint call back to save the model with the least mean square error on the validation set for each epoch.

Next, we start the training by calling the “.fit” method and supplying it with batches for both the training and validation sets. (We used a batch size of 100 images, with 300 steps per epoch for the training set and a batch size of 10 images, with 200 steps per epoch for the validation set. The training was done for 30 epochs initially however the model was later trained further.)

We then plot the training and validation mean square error against the epochs and load the weights of the best checkpoint into the model. We then save the model as an h5 file. Finally, we added a code cell to enable us to easily download the h5 file.

Experiments and Failures

On our first attempt, we collected the data at a resolution of 800*600 to reduce the data size, we then tested at a higher resolution, we assumed the simulator would be able to manage the difference in resolution between testing and training. It did not. This caused the model to instantly drive into large bodies of water. We figured this issue out eventually and created a new dataset using the resolution we were testing at.

At first, we tried to train the model just using the raw frames obtained from the simulator. The results of trying to make the model drive the car were very bad even though the loss after training was low. We decided that the data needs to be preprocessed so for this step we applied the following

- Removed useless features by cropping the images so only the road is visible to the model.
- Changed the colour space from RGB to YUV as proposed in NVIDIA's paper
- Added gaussian blur to the images
- Normalized the images so that the values of the pixels are from 0 to 1 instead of 0 to 255
- Resized the images to have the same dimension as the images used by NVIDIA

After retraining and trying the new model we noticed a bit of improvement but the model still didn't generalize well. We concluded that overfitting is the problem. To reduce the overfitting we applied the following

- Added data augmentation so that the data has more variety.
- Added dropout layers between fully connected layers. To introduce regularization.

Initially, the value of the dropout was set to 50% which worked fine for the first track however when testing on the second track the results were far less impressive, thus we decided to set the dropout value between the dense layers to 20%, this resulted in a model that was better able to drive the second track.

After retraining and trying the model we noticed a significant improvement, however, while the model was able to almost fully complete the track it struggled on one particular area of the second track where the model had to perform an "S" turn. Since the model seemed to struggle on only this part of the track we decided to let the model train a bit more and added additional data of the "S" turn to the original data, this did not seem to help much. Another experiment was training the existing model further on data extracted from the "S" turn only without the previous data, but only training it for a limited number of epochs. On the first attempt where we only trained it for one additional epoch, the model was able to perform the "S" turn and was able to complete a lap successfully.

References

[1] End to end learning for self-driving cars - NVIDIA. (n.d.). Retrieved May 29, 2022, from https://images.NVIDIA.com/content/tegra/automotive/images/2016/solutions/pdf/end-to-end-dl-using-px.pdf