ECE-5831 – Project Group 1 Final Report:

End-To-End Learning for Self-Driving Cars

Team Members:

Suhasini Kalaiah Lingaiah

John Orlowski

Introduction:

In 2016, researchers from NVIDIA corporation demonstrated the use of a convolutional neural network (CNN) to map pixels from a vehicle forward facing camera directly to steering commands without the use of path planning or any other explicit decompensation, such as lane marking [1]. The results of this approach were remarkably effective. The NVIDIA team was able to achieve autonomous driving for 90% of the driving time on varying road surfaces under different conditions, using the driver's steer angle as the training value. The NVIDIA experiment further demonstrated the efficacy of CNNs in pattern recognition/Machine Learning tasks.

Proposal:

Project Team One proposed to recreate the NVIDIA CNN, following the same architecture established in [1]. A baseline performance of the NVIDIA model was to be achieved by using a training simulator provided by Udacity [2]. After successfully training the baseline CNN and testing the performance on the simulator, the team was to then attempt to modify the architecture to achieve better performance. Metrics for performance that were proposed to be included but were not limited to: accuracy of steer angle from training and smoothness of steering in autonomous mode.

Smoothness of steering was to be identified through observing a video of the simulation in action. Possible points of improvement included synthesizing 3D representations of images taken with the simulated vehicle's cameras to create more precise viewpoint transformation. It was hoped that with a more precise viewpoint transformation the CNN would become more robust to visual obscurities such as rain, snow, and fog. Also, with a precise 3D viewpoint, it was hoped that it would be possible for distance to be inferred for use by higher-level vehicle dynamic control models without adding the additional complexity and cost of high-end LIDAR sensors.

Approach:

Our approach was to center around using the Udacity Self-Driving Car Simulator as a dynamic data source for both training and testing. The simulator was a tool that was built specifically for Udacity's Self-Driving Car Nanodegree, with the goal of teaching students to train models to navigate various roads. It was created using the Unity game engine and is an open-source project available on GitHub. It operates like a video game, generating a series of images once the program is started up. There are multiple tracks to choose from, as well the ability to generate new tracks if needed.

For training data, we used a series of captured images from the simulator, along with the relevant data on the steering angle, throttle position, and speed, to train a CNN to perform on a given track. Each observation shall consist of three .jpg images (left, right, and center), plus the data mentioned above stored in a related .csv file. The data was easy to manipulate in the current format.

We implemented NVIDIA's CNN sequential model with Python Keras, by matching the filter size and using the 2x2 strided convolution in each layer as in the NVIDIA design for each convolutional layer, so the total number of

parameters match. NVIDIA does not deploy local max pooling to gradually decrease the size of the feature activated images at each layer of the CNN. The input layer of the NVIDIA CNN accepts a 66 x 200 x 3 YUV format image; in order for the number of parameters at each layer of the CNN to match with the Keras implementation, we pre-processed the images generated by the Udacity simulator using OpenCV, to convert from RGB -> YUV format as well as resized the images to the correct size.

Results:

The final results of the project are presented in the included video, which shows the constructed model successfully completing multiple runs around the lake track without incident. The code used to achieve this model can be found in the ECE_583_Project_Team_1_Steering_Angle_CNN Python notebook on GitHub. The script used to execute the model and to control the throttle is called drive.py and can also be found on the team GitHub.

The key finding of our project concluded that a few different things were crucial to ensuring success. Collecting a large dataset was necessary to produce good results. This included running around each track many times, in both directions, using external controllers and recording the results. Additionally, we relied on external datasets provided by Udacity. Another important factor was to include a variety of modifications to the input images, resulting in a more robust dataset. These modifications are described in more detail below. A final breakthrough in our process was to increase the number of samples passed in at a time during training, increasing the training set and allowing the model to successfully navigate the lake track.

Modifications and attempted improvements:

While researching previous attempts to solve the problem of a self-driven car based on this simulator and architecture, one consistent theme was the willingness of the CNN to over-predict a neutral steering angle. Additionally, with only two tracks to train our model on, there was a chance that the CNN would train itself to be overfitted to only one or the other in certain ways, rather than generalizing shared characteristics of the tracks. To improve the overall robustness of our training data, several image augmentation functions were created with the goal of adding various changes to the imagery before passing it into the CNN. Often, these changes also include corresponding steering angle changes, with the goal of creating a dataset that is robust and includes a wider distribution of steering angles.

The horiz_flip() function was created to do a horizontal flip of the input image, while also flipping the steering angle to match by changing the sign. This was to help make sure we have a CNN that was able to turn left and right with ease, instead of overfitting the track and becoming biased towards one direction or the other.

```
def horiz_flip(img, steer_ang):
    ret_val = (cv2.flip(img,1), -steer_ang)
    return ret_val
```

The image_darkening() function simply made all pixels within an image darker, simulating some of the different conditions seen in the Jungle track, which had more shadows. This could also help to account for different weather conditions.

```
def image_darkening(img, low=0.6, high=0.9):
    img = img.astype(np.float32)
    scalar = np.random.uniform(low, high)
    img = scalar*img

    return img.astype(np.uint8)
```

The add_shadow() routine is similar to the image_darkening() in terms of the purpose, but was focused on drawing the shadows of individual objects. It created a four-sided polygon using random positions as the corners (two near the middle of the image and two near the bottom) and filled in that portion of the image with a randomly scaled darker pixel.

```
def add_shadow(img, low=0.4, high=0.7):
    cols, rows = (img.shape[0], img.shape[1])

    top_y = np.random.random() * rows
    bottom_y = np.random.random() * rows
    bottom_y_right = bottom_y + np.random.random() * (rows - bottom_y)
    top_y_right = top_y + np.random.random() * (rows - top_y)

    bottom_offset = np.random.uniform(cols,cols*.7)
    top_offset = np.random.uniform(0.5*cols,0.35*cols)

    if np.random.random() <= 0.5:
        bottom_y_right = bottom_y - np.random.random() * (bottom_y)
        top_y_right = top_y - np.random.random() * (top_y)

    poly = np.asarray([[ [top_y,top_offset], [bottom_y, bottom_offset], [bottom_y_right, bottom_offset], [top_y_right,top_offset]]], dtyp
e=np.int32)

    mask_weight = np.random.uniform(low, high)
    origin_weight = 1 - mask_weight

    mask = np.copy(img).astype(np.int32)
    cv2.fillPoly(mask, poly, (0,0,0))

    return cv2.addWeighted(img.astype(np.int32), origin_weight, mask, mask_weight, 0).astype(np.uint8)
```

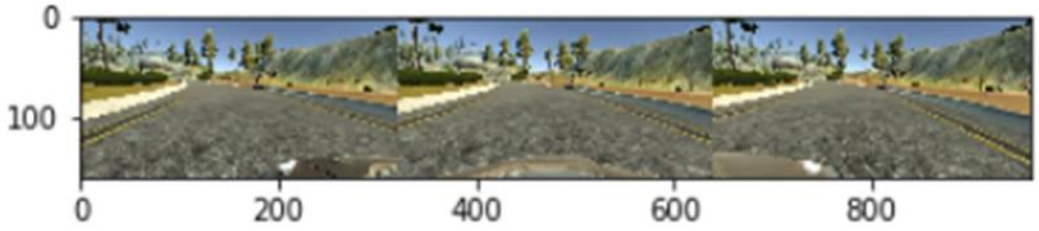
The rand_shifts() function was one of several different attempts to keep the CNN from becoming too closely biased towards a steering angle of zero. This shifted the images by a few pixels vertically, but potentially by plus or minus 50 pixels horizontally. The vertical shifts were thought to have no impact on steering, but the horizontal shifts also included a shift in the steering angle. The proposed amount of the shift (.35 rad per 100 pixels) was determined experimentally, after trying a few values.

```
def rand_shifts(img, steer_ang):
    rows, cols = (img.shape[0], img.shape[1])
    x_shift = np.random.randint(-50, 50)
    y_shift = np.random.randint(-10, 10)

    # Change steer angle based off horizontal shift
    steer_ang += x_shift * (.35/100)
    translation_matrix = np.float32([[1, 0, x_shift],[0, 1, y_shift]])
    img = cv2.warpAffine(img, translation_matrix, (cols, rows))

    return img, steer_ang
```

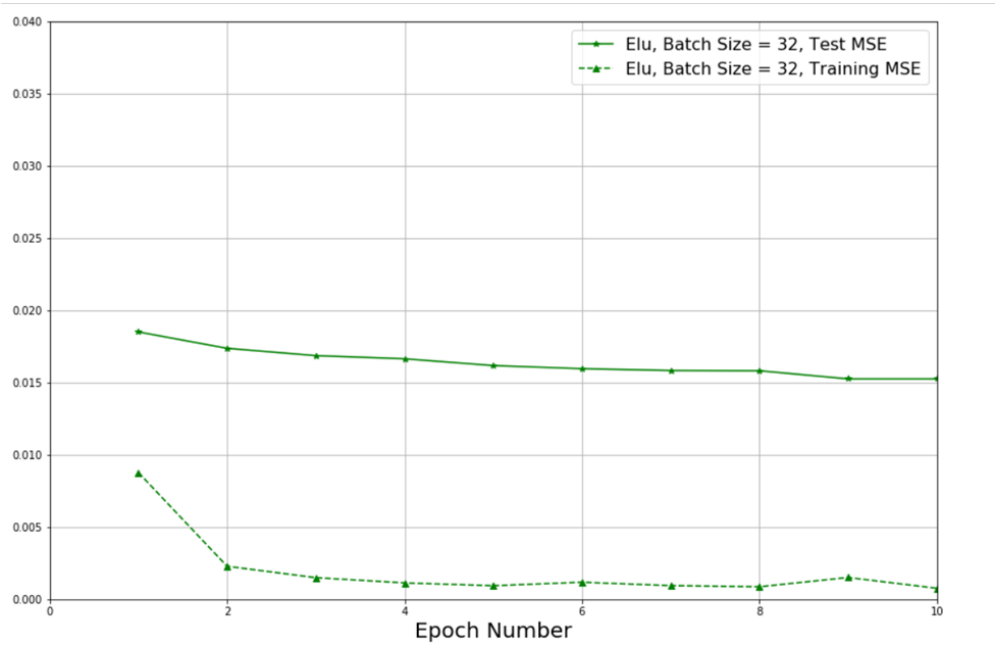
Finally, the augment_image() function combined all image subfunctions. This function took in all images in the training set, randomly modifying them based on a different probability for each augmentation. The probabilities were chosen somewhat arbitrarily, based on a simple understanding of how big of an impact we wanted for each change to the image. In the future, these probabilities could be passed into the function and then varied, almost like another hyperparameter.

```
def augment_image(img, steer_ang):

    #probability of each change
    p1 = .3
    p2 = .05
    p3 = .1
    p4 = .3

    if np.random.random() <= p1:
        img, steer_ang = horiz_flip(img, steer_ang)

    if np.random.random() <= p2:
        img = image_darkening(img)

    if np.random.random() <= p3:
        img = add_shadow(img)

    if np.random.random() <= p4:
        img, steer_ang = rand_shifts(img, steer_ang)

    return img, steer_ang
```

Another final image augmentation was a collage of the left, right, and center camera as can be seen below:



The image augmentation based on this alone was able to decrease the MSE significantly on its own, as can be seen, also below:



A further modification that was experimented with was manipulation of raw steering data to distribute the data more evenly and overcome a strong bias towards a steering angle of 0 degrees in training. The modification came in the form of applying a statistical distribution to the data to reorganize it and create more continuous values in the steering data. The hope in creating more continuous values in the steering data was that the steering of the car itself in autonomous mode would become smoother. However, before the modification could be included into the base code, a quantifiable metric had to prove the efficacy of the modification and not just a qualitative improvement in smooth steering. As can be seen below, the normalized pdf applied to the steering data did in fact, “smooth out,” the bias towards 0 degrees steer angle; however, when training was applied, the MSE performance against the baseline showed

a degradation in performance. The new MSE generated was 0.2, whereas the baseline MSE, including the other image enhancement improvements, was closer to 0.02. Since there was no quantifiable improvement in model performance, this feature was not included in the final release of the project code.