Behavioral cloning

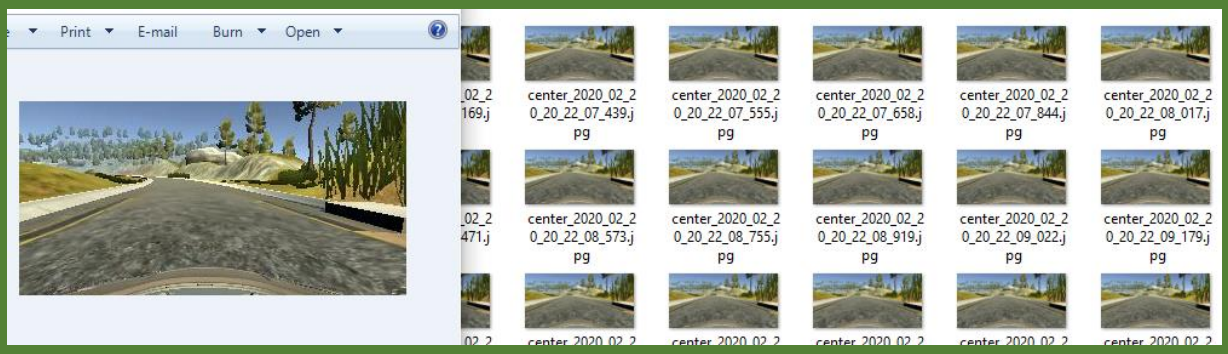
Using a simulator we will collect data through driving the car alone the track in training mode, as the car moves, it takes images and to every image we will record the steering angle, speed, throttle etc. We will then show the dataset to a convolutional neural network that will learn from the data (how to drive). After training we then put the car in a different track for it to apply the model in a new track.

Collecting data.

Try to drive as good as possible, move around the track several times so as to get more training data. Another way to balance the data is to drive the car in the other direction so as to avoid the bias whereby the car is more familiar with one side. The data is in form of a file and images: the file contains the name of the image, the speed of the car at the moment the picture was taken, the throttle and the steering angles; (-) for left and (+) for right and 0 when moving straight.

118	C:\Users\Ta	C:\Users\Taf	C:\Users\Taf	0	1	0	30.1903
119	C:\Users\Ta	C:\Users\Taf	C:\Users\Taf	-0.39683	1	0	30.1682
120	C:\Users\Ta	C:\Users\Taf	C:\Users\Taf	0	0.4575	0	30.0529

The other form of data is the images, the car uses 3 cameras to record pictures at every instance.



Loading data

We track the file with os and read the csv file then use the names we want to label the information. Below is a view of the first 5 values (head).

```
datadir = 'track'
columns = ['center', 'left', 'right', 'steering', 'throttle', 'reverse', 'speed']
data = pd.read_csv(os.path.join(datadir, 'driving_log.csv'), names = columns)
pd.set_option('display.max_colwidth', -1)
data.head()
```

	center	left
0	C:\Users\Tafara\Desktop\track-data\IMG\center_2020_02_20_20_22_06_887.jpg	C:\Users\Tafara\Desktop\track-data\IMG\left_2020_02_20_20_22_06_887.jpg
1	C:\Users\Tafara\Desktop\track-data\IMG\center_2020_02_20_20_22_07_169.jpg	C:\Users\Tafara\Desktop\track-data\IMG\left_2020_02_20_20_22_07_169.jpg
2	C:\Users\Tafara\Desktop\track-data\IMG\center_2020_02_20_20_22_07_439.jpg	C:\Users\Tafara\Desktop\track-data\IMG\left_2020_02_20_20_22_07_439.jpg

The next step is to remove the unnecessary part of the image names, leaving only the tails, which makes the data much easier to read.

```
def path_leaf(path):
  head, tail = ntpath.split(path)
  return tail
data['center'] = data['center'].apply(path_leaf)
data['left'] = data['left'].apply(path_leaf)
data['right'] = data['right'].apply(path_leaf)
data.head()
```

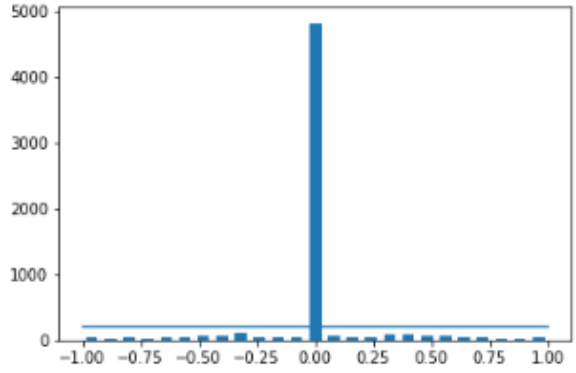
	center	left	right
0	center_2020_02_20_20_22_06_887.jpg	left_2020_02_20_20_22_06_887.jpg	right_2020_02_20_20_22_06_887.jpg
1	center_2020_02_20_20_22_07_169.jpg	left_2020_02_20_20_22_07_169.jpg	right_2020_02_20_20_22_07_169.jpg
2	center_2020_02_20_20_22_07_439.jpg	left_2020_02_20_20_22_07_439.jpg	right_2020_02_20_20_22_07_439.jpg
3	center_2020_02_20_20_22_07_555.jpg	left_2020_02_20_20_22_07_555.jpg	right_2020_02_20_20_22_07_555.jpg
4	center_2020_02_20_20_22_07_658.jpg	left_2020_02_20_20_22_07_658.jpg	right_2020_02_20_20_22_07_658.jpg

Showing the data

We define the number of bars to be shown (25), then draw a histogram using the data (of steering angles). To have the center of the graph we add the segment [: ,1] with [1, :], then multiply by 0.5 to get the original numbers. The 2 numbers beside 0 are added to give 0. Since there is a higher frequency at 0 angle, we have to set a limit for the frequency (set a threshold) so as to remove the bias since the number of values will be almost the same. We then use pltplot which takes (x1, x2) and (y1, y2) to show our threshold.

```
num_bins = 25
samples_per_bin = 200
hist, bins = np.histogram(data['steering'], num_bins)
center = (bins[:-1] + bins[1:]) * 0.5
plt.bar(center, hist, width=0.05)
plt.plot((np.min(data['steering']), np.max(data['steering'])), (samples_per_bin, samples_per_bin))
```

```
[<matplotlib.lines.Line2D at 0x7f8dd2303940>]
```



Removing the bias error

Set an empty list with the samples we want to remove, then set a loop for every bin. For every loop we loop into all the steering data. If the value of point at [i] is > than the current bin and less than the next bi value then we append it into the list [i]. If the list has more than 200 values then it has exceeded the threshold and we need to remove the extra data, but we need to shuffle each time before removing. We then put all the extra values that are above the threshold in a

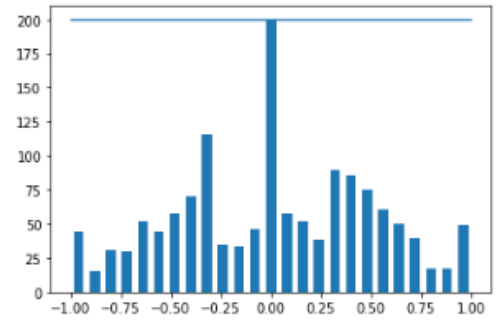
new list. This list is then removed from the data with the remove list extend function; leaving only values either less or equal to the threshold (num per bin), in this case which is 200.

```
remove_list = []
for j in range(num_bins):
  list_ = []
  for i in range(len(data['steering'])):
    if data['steering'][i] >= bins[j] and data['steering'][i] <= bins[j+1]:
      list_.append(i)
  list_ = shuffle(list_)
  list_ = list_[samples_per_bin:]
  remove_list.extend(list_)

print('removed: ', len(remove_list))
data.drop(data.index[remove_list], inplace=True)
print('remaining: ', len(data))
```

Balanced data graph

This is the result of the balanced data, that which is not biased at zero point or any other point with too high frequency. If the car fails to move back to the road after getting off the road then it means the data is probably biased at 0, so we will need to take recovery laps, making more turns, turning off the recorder when moving straight.

```
total data:  6021
removed:  4614
remaining:  1407
[<matplotlib.lines.Line2D at 0x7f8d5db4fe10>]
```



Training

Create a function with parameters data directory and the new data; 2 variables image path and steering. We then loop in all data. We then index our data using iloc, which returns all the data with their index order as shown below.

```
center      center_2018_07_16_17_11_44_413.jpg
left        left_2018_07_16_17_11_44_413.jpg
right       right_2018_07_16_17_11_44_413.jpg
steering   -0.05
throttle    0.642727
reverse     0
speed       1.43401
Name: 12, dtype: object
```

We then use this to get the center, left and right values. We also get the paths of all those images. In the steering array we will put the fourth index, angle. We then put the paths and angles in arrays, and then return the path and steering angle.

```
def load_image_steering(datadir, df):
  image_path = []
  steering = []
  for i in range(len(data)):
    indexed_data = data.iloc[i]
    center, left, right = indexed_data[0], indexed_data[1], indexed_data[2]
    image_path.append(os.path.join(datadir, center.strip()))
    steering.append(float(indexed_data[3]))
  image_paths = np.asarray(image_path)
  steerings = np.asarray(steering)
  return image_paths, steerings

image_paths, steerings = load_image_steering(datadir + '/IMG', data)
```

Splitting the data for training

We use function train test split to split the data from the images and steering, the test size (0.2) is the proportion of the data that we want to use for validation, the default being 0.5. The last variable is just a determinant of how the data will be randomized.

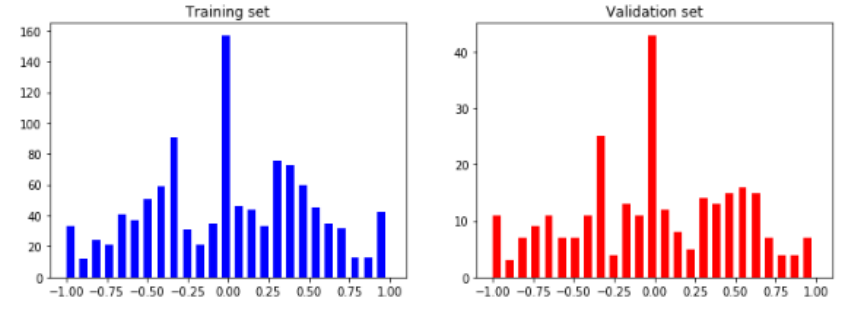
```
X_train, X_valid, y_train, y_valid = train_test_split(image_paths, steerings, test_size=0.2, random_state=6)
print('Training Samples: {}\nValid Samples: {}'.format(len(X_train), len(X_valid)))

Training Samples: 1125
Valid Samples: 282
```

To confirm that both data sets contain data that is uniformly distributed and not biased we will plot the 2 histograms. One row plot, 2 figs, size of (12, 4), the first one (fig[0]) will be for training data against bins, then the second one shows the validation against bins.

```
fig, axes = plt.subplots(1, 2, figsize=(12, 4))
axes[0].hist(y_train, bins=num_bins, width=0.05, color='blue')
axes[0].set_title('Training set')
axes[1].hist(y_valid, bins=num_bins, width=0.05, color='red')
axes[1].set_title('Validation set')
```

Our graphs, even though not exactly the same but they show a general pattern which shows that the data is not biased to one side.

```
Text(0.5, 1.0, 'Validation set')
```



Preprocessing data

Create a function which takes in an image as its argument. The first process is to read the image, then we start by cropping the image (simply by slicing), then the color space is changed to yuv since we are going to use the nvidia model. After that the Gaussian blur function is used to refine the image. We then resize the image to (200, 66), making it small so that computation will be faster. The last stage is to normalize the image, dividing it by 255.

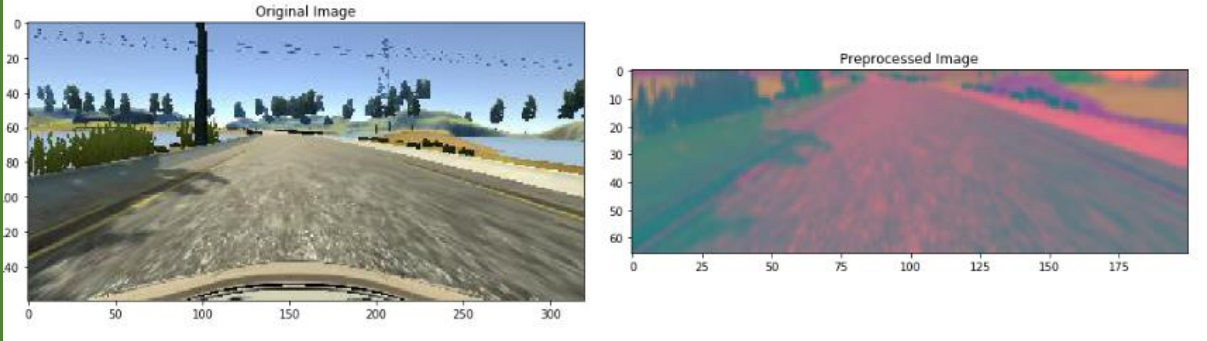
```
def img_preprocess(img):
  img = mpimg.imread(img)
  img = img[60:135,:,:]
  img = cv2.cvtColor(img, cv2.COLOR_RGB2YUV)
  img = cv2.GaussianBlur(img, (3, 3), 0)
  img = cv2.resize(img, (200, 66))
  img = img/255
  return img
```

Comparison

We define our 2 images, one is original and the other will be processed. The 2 are plotted next to each other. We use layout function to avoid overlapping, then we show the 2 images.

```
image = image_paths[100]
original_image = mpimg.imread(image)
preprocessed_image = img_preprocess(image)

fig, axs = plt.subplots(1, 2, figsize=(15, 10))
fig.tight_layout()
axs[0].imshow(original_image)
axs[0].set_title('Original Image')
axs[1].imshow(preprocessed_image)
axs[1].set_title('Preprocessed Image')
```

```
Text(0.5, 1, 'Preprocessed Image')
```



Preprocessing

Map will iterate through all the images, apply the preprocessing image function to every image, keep it as a list and then stored it as an array.

```
X_train = np.array(list(map(img_preprocess, X_train)))
X_valid = np.array(list(map(img_preprocess, X_valid)))
```

Random plotting any image will confirm if the preprocessing function has been applied effectively to all the images.

```
plt.imshow(X_train[random.randint(0, len(X_train) - 1)])
plt.axis('off')
print(X_train.shape)
```

```
(1125, 66, 200, 3)
```

Nvidia model

Define our sequential model, input our data with 24 filters and a 5 x 5 kernel size. Subsample shows the size of pixels checked at a time, so the box will move 2 pixels each time up and sideways. The size of the input image will be 200 x 66 x 3. We will use the elu as the activation method. Continue to add the layers according to the nvidia model. We use the learning rate 0.01 to increase accuracy and then compile the model. We add drop outs after training too many points. Drop outs will turn a fraction of outputs to zero.

```
def nvidia_model():
  model = Sequential()
  model.add(Convolution2D(24, 5, 5, subsample=(2, 2), input_shape=(66, 200, 3), activation='elu'))
  model.add(Convolution2D(36, 5, 5, subsample=(2, 2), activation='elu'))
  model.add(Convolution2D(48, 5, 5, subsample=(2, 2), activation='elu'))
  model.add(Convolution2D(64, 3, 3, activation='elu'))

  model.add(Convolution2D(64, 3, 3, activation='elu'))
  model.add(Dropout(0.5))


  model.add(Flatten())

  model.add(Dense(100, activation='elu'))
  model.add(Dropout(0.5))

  model.add(Dense(50, activation='elu'))
  model.add(Dropout(0.5))

  model.add(Dense(10, activation='elu'))
  model.add(Dropout(0.5))

  model.add(Dense(1))

  optimizer = Adam(lr=1e-3)
  model.compile(loss='mse', optimizer=optimizer)
  return model
```

Model training

We use 30 epochs, a batch size of 100, verbose as 1 and shuffle to true. As the epochs increase the loss values are converging to a small value and that means that the model is working well. To show the results graphically we will plot an output hist.

```
history = model.fit(X_train, y_train, epochs=30, validation_data=(X_valid, y_valid), batch_size=100, verbose=1, shuffle=1)

1125/1125 [==============================] - 9s 8ms/step - loss: 0.8940 - val_loss: 0.2206
Epoch 2/30
1125/1125 [==============================] - 8s 7ms/step - loss: 0.2880 - val_loss: 0.2064
Epoch 3/30
1125/1125 [==============================] - 8s 7ms/step - loss: 0.2736 - val_loss: 0.1941
Epoch 4/30
1125/1125 [==============================] - 8s 8ms/step - loss: 0.2555 - val_loss: 0.1835
```

The training line did not overpass the validation line, and so that means our model did not overfit.

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.legend(['training', 'validation'])
plt.title('Loss')
plt.xlabel('Epoch')
```

```
Text(0.5, 0, 'Epoch')
```