HW_4_official_final

December 18, 2020

[1]:
```
from tqdm import tqdm
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as image
import cv2
import skimage.morphology as skim_morph
import scipy.ndimage.measurements as scipy_im_measure
import pandas as pd
from scipy import signal
import math
import scipy.spatial.distance as scp_dist
import random
import scipy.ndimage as scp_nd
```

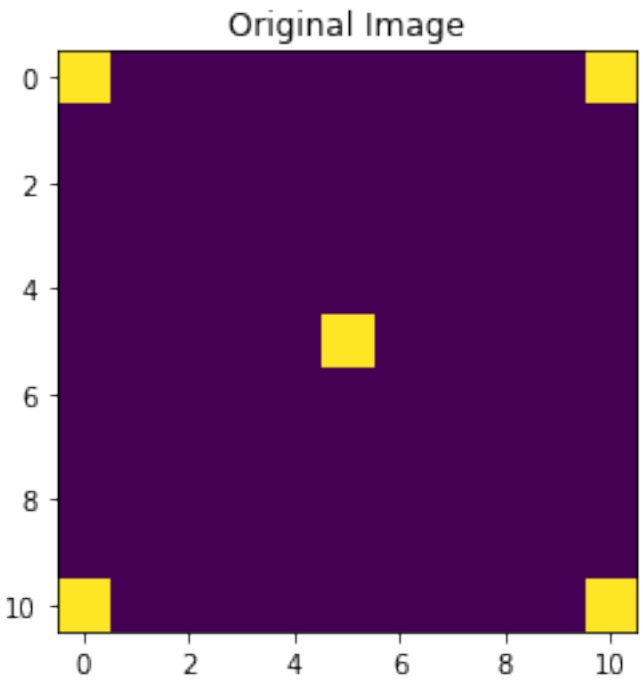
Problem 1: Hough Transform

Reference: https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_gradients/py_

https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_thresholding/py_thresho

Reference: https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_houghlines/py

[2]:
```
test = np.zeros((11,11))

test[0,0] = 1
test[0,-1] = 1
test[-1,0] = 1
test[-1,-1] = 1
test[5,5] = 1

plt.imshow(test)
plt.title('Original Image');
```



[3]:
```
theta_rad = np.arange(-90,91)*np.pi*(1/180)
rho_axis = np.arange(-14,15)
```

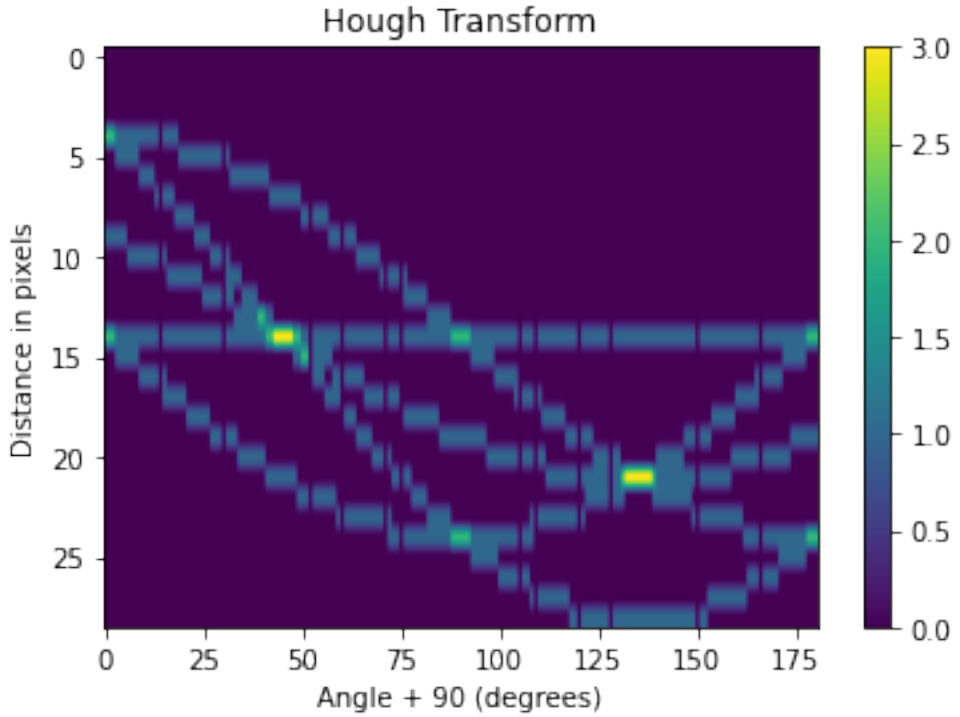
[4]:
```
ac = np.zeros((29, 181))
```

[5]:
```
# looping through image
for i in range(test.shape[0]):
    for j in range(test.shape[1]):
        if(test[i][j]==1):
            rho = i*np.cos(theta_rad) + j*np.sin(theta_rad)

            # round rho
            rho_rounded = np.round_(rho).astype(int)

            # convert rad theta to deg theta
            theta_deg = (theta_rad * (180/np.pi)).astype(int)

            # increment accumulator
            ac[rho_rounded, theta_deg] = ac[rho_rounded, theta_deg] + 1
```

Rolling:

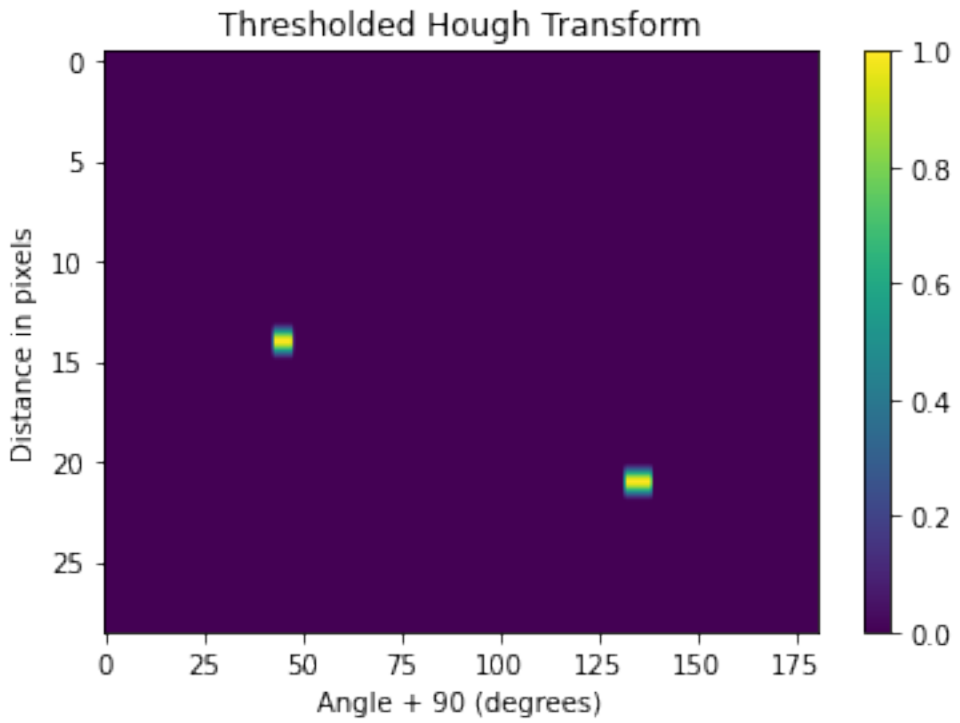
[6]:
```
ax_horiz_roll = np.roll(ac, 90, axis = 1)
ax_vert_roll = np.roll(ax_horiz_roll, 14, axis = 0)
```

[7]:
```
plt.imshow(ax_vert_roll, aspect = 'auto')
plt.title('Hough Transform')
plt.colorbar()
plt.xlabel('Angle + 90 (degrees)')
plt.ylabel('Distance in pixels');
```



b. Threshold the HT by looking for any (,) cells that contains more than 2 votes then plot the corresponding lines in (x,y)-space on top of the original image.

[8]:
```
thresh_ht = np.where(ax_vert_roll>2,1,0)
plt.imshow(thresh_ht, aspect = 'auto')
plt.title('Thresholded Hough Transform')
plt.colorbar()
plt.xlabel('Angle + 90 (degrees)')
plt.ylabel('Distance in pixels');
```

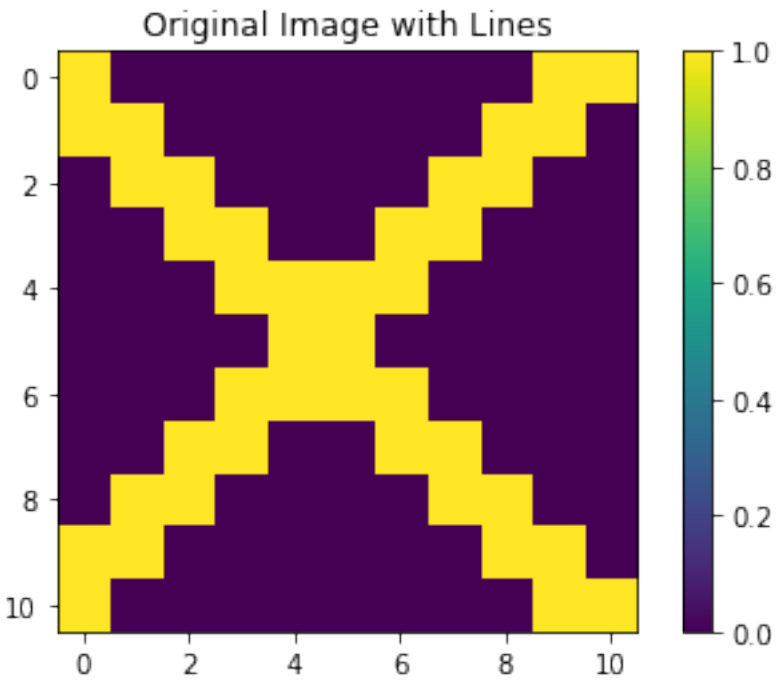


[9]:
```
# finding the indices of most votes

dist, angle = np.where(thresh_ht>0)

orig_dist = dist - 14
orig_angle = angle - 90
```

[10]:
```
for i in range(orig_dist.shape[0]):

    rho_0, theta_0 = orig_dist[i], orig_angle[i]

    x = np.arange(11)

    y = np.floor((rho_0 - (x*np.cos(theta_0*(np.pi/180))))/(np.sin(theta_0*(np.
↪pi/180)))).astype(int)

    test[x,y] = 1
```

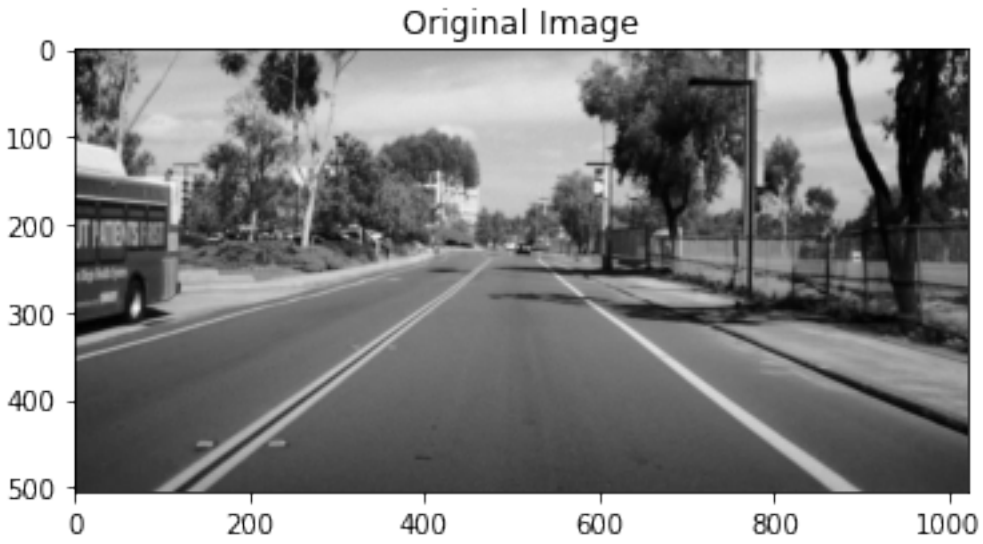
[11]:
```
plt.imshow(test)
plt.colorbar()
plt.title('Original Image with Lines')
```

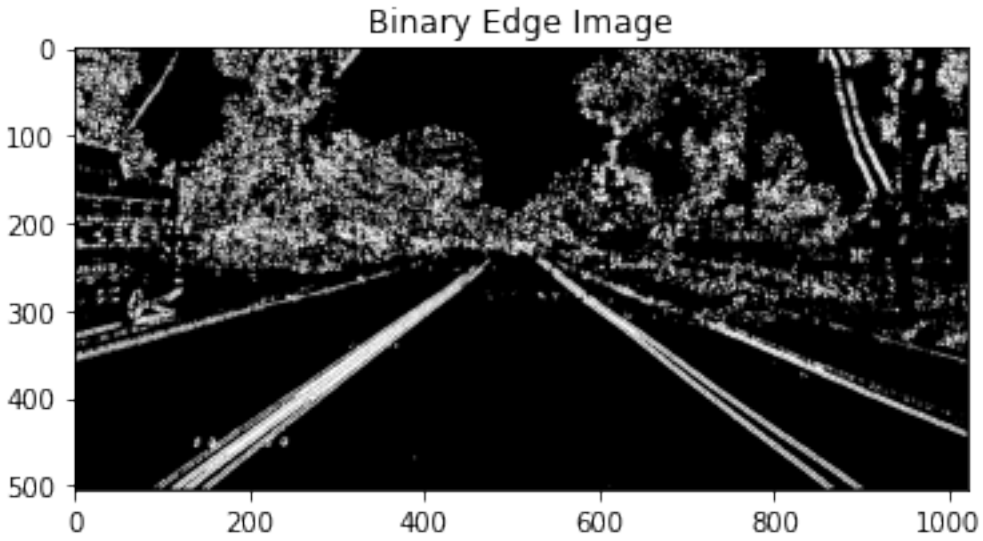
[11]:
```
Text(0.5, 1.0, 'Original Image with Lines')
```



Part 3

[12]:
```
img = plt.imread('lane.png')
img = img[:,:,0]
plt.figure()
plt.imshow(img, cmap = 'gray');
plt.title('Original Image')

img_64_sobel = cv2.Sobel(img,cv2.CV_64F,1,1,ksize=7)
abs_sobel64f = np.abs(img_64_sobel)
img_thresh = np.uint8(abs_sobel64f)
ret3,th3 = cv2.threshold(img_thresh,0,1,cv2.THRESH_OTSU)
plt.figure()
plt.imshow(th3, cmap = 'gray');
plt.title('Binary Edge Image');
```





[13]:
```
theta_rad_lane = np.arange(-90,91)*np.pi*(1/180)
rho_axis_lane = np.arange(-1140,1141)
```

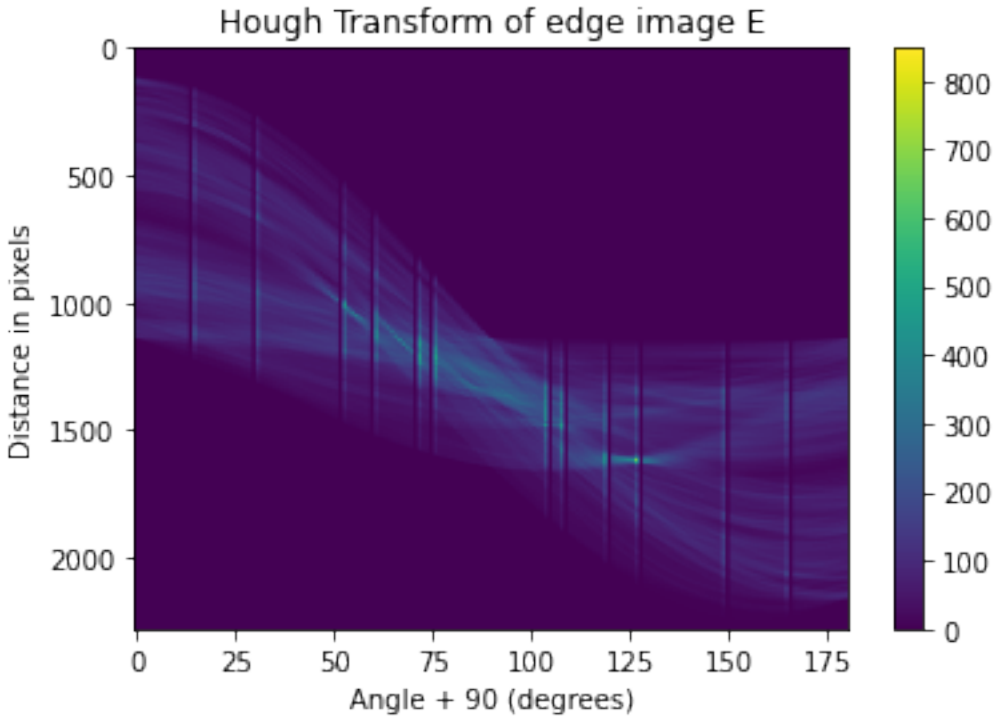
[14]:
```
ac_lane = np.zeros((2281, 181))
```

[15]:
```
# looping through image
for i in range(th3.shape[0]):
    for j in range(th3.shape[1]):
        if(th3[i][j]==1):
```

```
            rho_lane = i*np.cos(theta_rad_lane) + j*np.sin(theta_rad_lane)

            # round rho
            rho_rounded_lane = np.round_(rho_lane).astype(int)

            # convert rad theta to deg theta
            theta_deg_lane = (theta_rad_lane * (180/np.pi)).astype(int)

            # increment accumulator
            ac_lane[rho_rounded_lane, theta_deg_lane] =␣
↪ac_lane[rho_rounded_lane, theta_deg_lane] + 1
```

[16]:
```
ax_horiz_roll = np.roll(ac_lane, 90, axis = 1)
ax_vert_roll = np.roll(ax_horiz_roll, 1140, axis = 0)
```

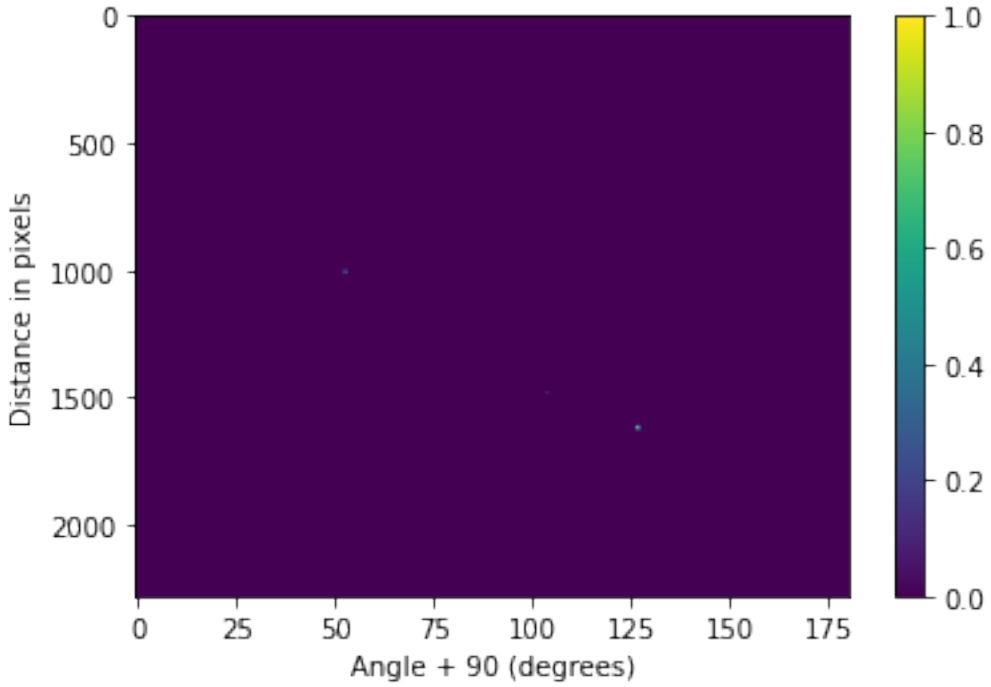
[17]:
```
plt.imshow(ax_vert_roll, aspect = 'auto')
plt.title('Hough Transform of edge image E');
plt.colorbar()
plt.xlabel('Angle + 90 (degrees)')
plt.ylabel('Distance in pixels');
```



[18]:
```
thresh_HT_lane = np.max(ax_vert_roll)*0.75
```

[19]:
```
thresh_HT_lane_applied = np.where(ax_vert_roll>thresh_HT_lane, 1, 0)
```

[20]:
```
plt.imshow(thresh_HT_lane_applied, aspect = 'auto')
plt.colorbar()
plt.xlabel('Angle + 90 (degrees)')
plt.ylabel('Distance in pixels');
```



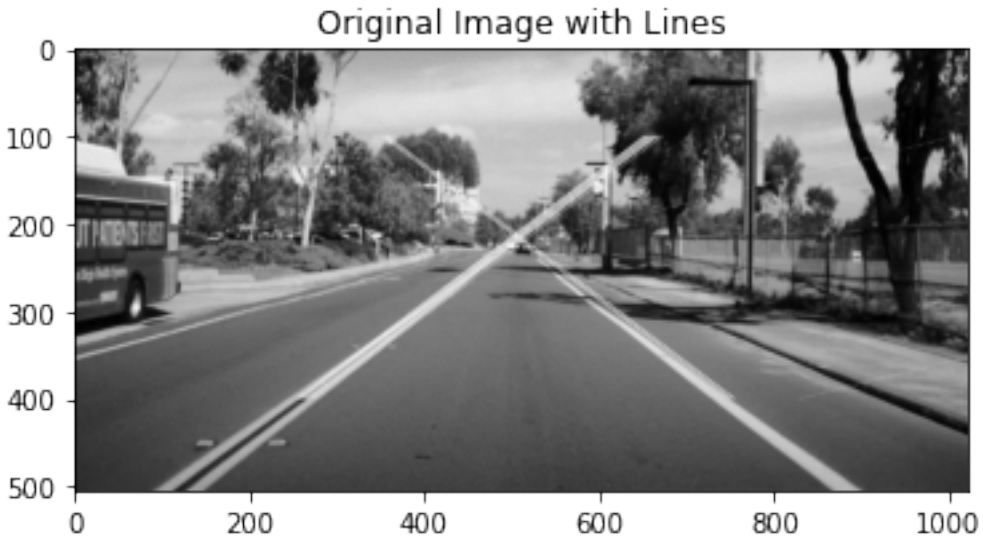
[21]:
```
# plotting lines

dist_lane, angle_lane = np.where(thresh_HT_lane_applied>0)


orig_dist_lane = dist_lane - 1140
orig_angle_lane = angle_lane - 90
```

[22]:
```
lane_copy = img
```

[23]:
```
for i in range(orig_dist_lane.shape[0]):

    rho_0, theta_0 = orig_dist_lane[i], orig_angle_lane[i]

    x = np.arange(400)

    x = x[::-1]
```

```
    x = x[:300]

    y = np.floor((rho_0 - (x*np.cos(theta_0*(np.pi/180))))/(np.sin(theta_0*(np.
↪pi/180)))).astype(int)

    lane_copy[x,y] = 1
```

[24]:
```
plt.imshow(lane_copy, cmap = 'gray')
plt.title('Original Image with Lines');
```



Part iv

[25]:
```
# using the thresholded HT, we must rethreshold using a
# specified range of theta this time.

# the angles seem to be 37 and -37 degrees based on my
# output from the previous part of the problem

# so I will rethreshold so that only the angles -37 and 37 are
# allowed
```

[26]:
```
print(orig_dist_lane)
print(orig_angle_lane)
```

```
[-136 -135 -133 -132  341  472  473  474  475  476  477  478  479  480
  481  482  485]
[-37 -37 -37 -37  14  37  37  37  37  37  37  37  37  37  37  37  37]
```

[27]:
```
# from above observation, we should not allow index 4 of both
# distance array and angle array

new_dist_lane = np.delete(orig_dist_lane,4)
new_angle_lane = np.delete(orig_angle_lane,4)
```

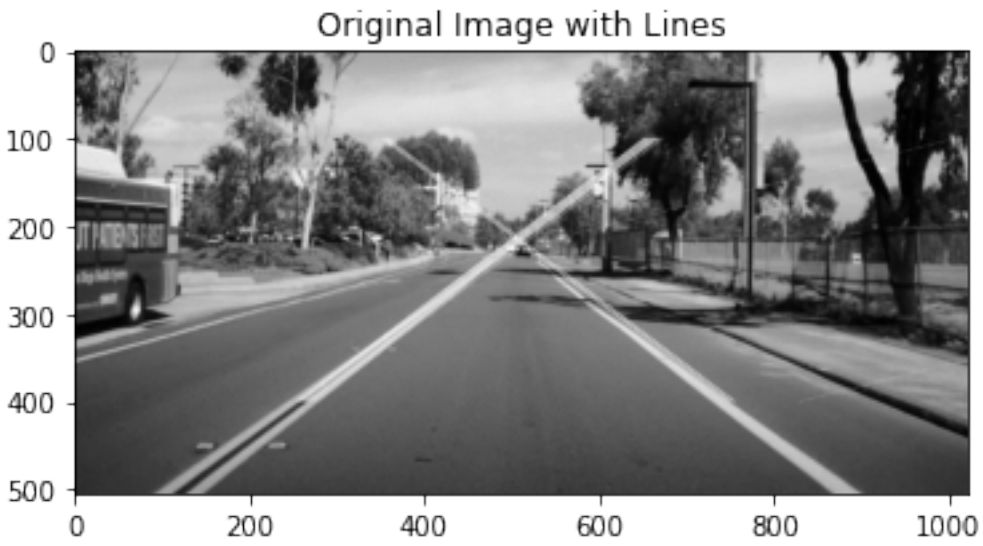
Now the only angles are -37 and 37. These are the theta values I got for 2(iv).

[28]:
```
new_angle_lane
```

[28]:
```
array([-37, -37, -37, -37,  37,  37,  37,  37,  37,  37,  37,  37,  37,
        37,  37,  37])
```

[29]:
```
# now rethresholding

lane_copy_new = img

for i in range(new_dist_lane.shape[0]):

    rho_0, theta_0 = new_dist_lane[i], new_angle_lane[i]

    x = np.arange(400)

    x = x[::-1]

    x = x[:300]

    y = np.floor((rho_0 - (x*np.cos(theta_0*(np.pi/180))))/(np.sin(theta_0*(np.
↪pi/180)))).astype(int)

    lane_copy_new[x,y] = 1
```

[30]:
```
plt.imshow(lane_copy_new, cmap = 'gray')
plt.title('Original Image with Lines');
```



Problem 2: K Means Segmentation

[31]:
```
def createDataset(im):
    return(im.reshape(-1).reshape(im.shape[0]*im.shape[1],3))
```

[32]:
```
def new_centers(features, centers):

    # given features and centers, this calculates new centers

    dist_mat = scp_dist.cdist(features, centers) # calculating distance array

    # calculating idx
    idx = np.argmin(dist_mat, axis = 1) + 1 # calculating group


    all_1 = np.where(idx == 1)
    mean_1 = np.mean(features[all_1], axis = 0)

    all_2 = np.where(idx == 2)
    mean_2 = np.mean(features[all_2], axis = 0)

    all_3 = np.where(idx == 3)
    mean_3 = np.mean(features[all_3], axis = 0)

    all_4 = np.where(idx == 4)
    mean_4 = np.mean(features[all_4], axis = 0)

    all_5 = np.where(idx == 5)
    mean_5 = np.mean(features[all_5], axis = 0)
```

```
    all_6 = np.where(idx == 6)
    mean_6 = np.mean(features[all_6], axis = 0)

    all_7 = np.where(idx == 7)
    mean_7 = np.mean(features[all_7], axis = 0)


    # stack all means together to form new centers matrix
    centers2 = np.vstack((mean_1, mean_2, mean_3, mean_4, mean_5, mean_6,␣
↪mean_7))

    return(centers2)
```

[33]:
```
def kMeansCluster(features, centers):

    max_iter = 1000


    for i in tqdm(range(max_iter)):

        centers2 = new_centers(features, centers)

        if(np.array_equal(centers2, centers)):

            dist_mat = scp_dist.cdist(features, centers2)
            idx = np.argmin(dist_mat, axis = 1) + 1 # calculating group

            return(idx, centers2)
        else:
            centers = centers2

    # given new centers, calculate idx
    dist_mat = scp_dist.cdist(features, centers2)
    idx = np.argmin(dist_mat, axis = 1) + 1

    return(idx, centers2)
```

[34]:
```
def centers_given_idx(features, idx):

    all_1 = np.where(idx == 1)
    mean_1 = np.mean(features[all_1], axis = 0)

    all_2 = np.where(idx == 2)
    mean_2 = np.mean(features[all_2], axis = 0)

    all_3 = np.where(idx == 3)
```

```
    mean_3 = np.mean(features[all_3], axis = 0)

    all_4 = np.where(idx == 4)
    mean_4 = np.mean(features[all_4], axis = 0)

    all_5 = np.where(idx == 5)
    mean_5 = np.mean(features[all_5], axis = 0)

    all_6 = np.where(idx == 6)
    mean_6 = np.mean(features[all_6], axis = 0)

    all_7 = np.where(idx == 7)
    mean_7 = np.mean(features[all_7], axis = 0)


    # stack all means together to form new centers matrix
    centers2 = np.vstack((mean_1, mean_2, mean_3, mean_4, mean_5, mean_6,␣
↪mean_7))

    return centers2
```

[35]:
```
def mapValues(im, idx):
    # find features
    features = createDataset(img)
    # calculate centers given idx and features
    centers2 = centers_given_idx(features, idx)

    # segment image

    features_new = features

    all_1 = np.where(idx==1)
    features_new[all_1] = centers2[0,:]

    all_2 = np.where(idx==2)
    features_new[all_2] = centers2[1,:]

    all_3 = np.where(idx==3)
    features_new[all_3] = centers2[2,:]

    all_4 = np.where(idx==4)
    features_new[all_4] = centers2[3,:]

    all_5 = np.where(idx==5)
    features_new[all_5] = centers2[4,:]

    all_6 = np.where(idx==6)
```

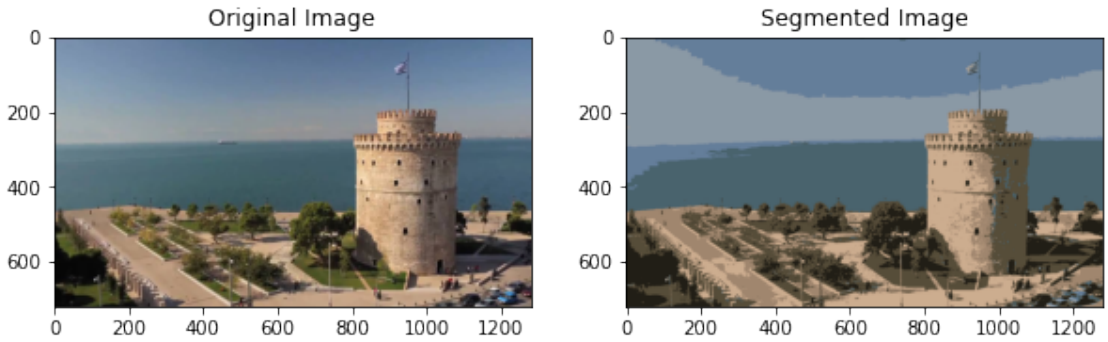
```
    features_new[all_6] = centers2[5,:]

    all_7 = np.where(idx==7)
    features_new[all_7] = centers2[6,:]

    seg_img = np.reshape(features_new, (img.shape[0],img.shape[1],3))

    return(seg_img)
```

[36]:
```
# given image, segment it using k means

img = plt.imread('white-tower.png')


# step 1: calculate features given image



features = createDataset(img)

# step 2: initialize centers

random.seed(5)
nclusters = 7
idx = np.random.randint(0, features.shape[0], (nclusters, 1))
centers = features[idx, :]
centers = np.squeeze(centers)

# step 3: calculate new centers using 10000 iterations of k-means

idx, centers = kMeansCluster(features, centers)

# step 4: segment image using idx

seg_img = mapValues(img, idx)
```

```
  5%|          | 53/1000 [00:03<00:59, 15.92it/s]
```

[38]:
```
img = plt.imread('white-tower.png')

fig = plt.figure(figsize = [10,5])
ax1 = fig.add_subplot(121)
ax1.imshow(img)
ax1.set_title('Original Image')
ax2 = fig.add_subplot(122)
ax2.imshow(seg_img)
ax2.set_title('Segmented Image')
```

[38]: Text(0.5, 1.0, 'Segmented Image')



My centers after segmentation were:

[40]:
```
centers
```

[40]:
```
array([[0.54425454, 0.6000721 , 0.6505426 ],
       [0.33977142, 0.3024052 , 0.22497821],
       [0.80300707, 0.6790147 , 0.56182367],
       [0.62938166, 0.5276112 , 0.4348132 ],
       [0.13083686, 0.1179914 , 0.08717921],
       [0.28665406, 0.38855982, 0.42901593],
       [0.39597246, 0.49507087, 0.60606676]], dtype=float32)
```

[ ]:

HW 4 Question 3

December 18, 2020

1. Please complete the FCN network, the fcn8s in ptsemseg/models/fcn.py. And briely describe the model structure. Instead of a simple line structure, the neural network model for FCN follows a DAG (Directed Acyclic Graph) structure. It can be divided into convolutional and pooling blocks.

1. The first convolutional block convolves with a 3x3 kernel and takes as input a 3 channel color image, and outputs a 64 channel feature map. This is followed by activation using a ReLU. The process of convolve + ReLU is repeated twice.

MaxPooling is performed on the output using a 2x2 kernel and a stride of 2.

2. The second convolutional block convolves with a 3x3 kernel and takes as input a 64 channel feature map, and outputs a 128 channel feature map. This is followed by activation using a ReLU. The process of convolve + ReLU is repeated twice in this block as well.

Maxpooling with 2x2 and stride 2 is then performed.

3. For the third convolutional block. Kernel size is 3x3. Input is 128 channel and output is 256 channel. This is followed by activation layer using ReLU. Convolve + ReLU is repeated thrice.

Maxpooling using 2x2 and stride 2 is performed on the output.

4. For the fourth convolutional block, everything remains the same as the third layer except the input feature map is 256 channels and the output feature map is 512 channels. Even the maxpooling hyperparameters remain the same as conv block 3.

5. The fifth conv block is identical to the 4th conv block except the input feature map is 512 channels and so is the output feature map.

6. On the output of the fifth block, conv with 7x7 kernel is performed. Then ReLU is applied. Then, conv with kernel 1x1 is performed with dropout. Then conv with kernel 1x1 with dropout is again performed. The output has num channels equal to the number of classes defined by the user.

7. The output of 6 is upsampled using a transpose convolution (deconvolution) block. For this, the parameters are initialized to bilinear interpolation coefficients, but they are learnable. The output now has the same number of pixels as the input image.

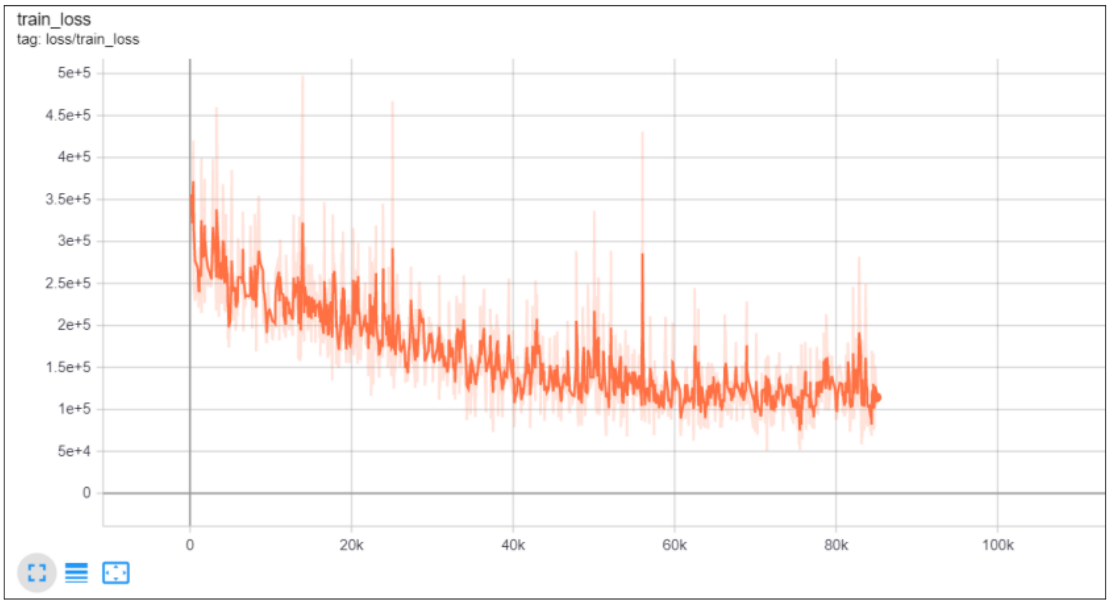
8. The output of 4 is upsampled using a transpose convolution block after passing through a 1x1 convolution layer, so that the output of this block is equal in number of pixels to the input image. Then, the resultant image is added to the output of 7.

9. Similarly, the output of 3 is passed through a 1x1 convolution and then upsampled so that it matches the input image in size. The resultant image is combined with the output of 7 and 8 to give a finer details.

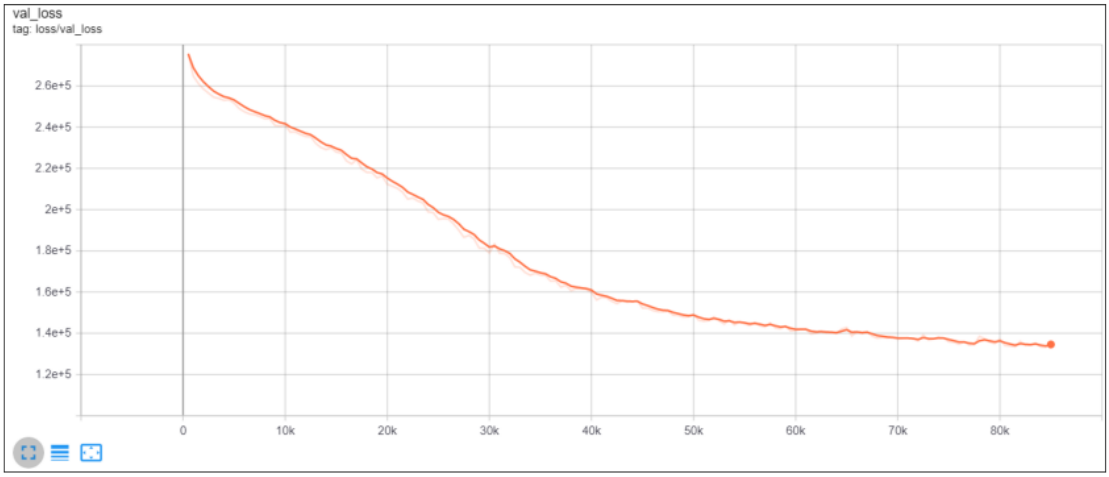
2. Do we use weights from a pre-trained model, or do we train the model from scratch?
We use a pretrained VGG model.

3. Please train the network with CityScape dataset. Visualize the training curves (suggested option: use Tensorboard). Include pictures of the training and validation curve.

[15]:
```
import matplotlib.pyplot as plt
import numpy as np
fig = plt.figure(figsize=[15,15])
ax = fig.add_subplot(111)
img = plt.imread('train_loss.PNG')
ax.imshow(img);
ax.get_xaxis().set_visible(False)
ax.get_yaxis().set_visible(False)
```



[3]:
```
fig = plt.figure(figsize=[15,15])
ax = fig.add_subplot(111)
img = plt.imread('val_loss.PNG')
ax.imshow(img);
ax.get_xaxis().set_visible(False)
ax.get_yaxis().set_visible(False)
```



4. What are the metrics used by the original paper? Do inference (validate.py) on the validation set. Which classes work well? Which classes do not? The metrics used by the original paper are:

1. pixel accuracy
2. mean accuracy
3. mean IU
4. frequency weighted IU

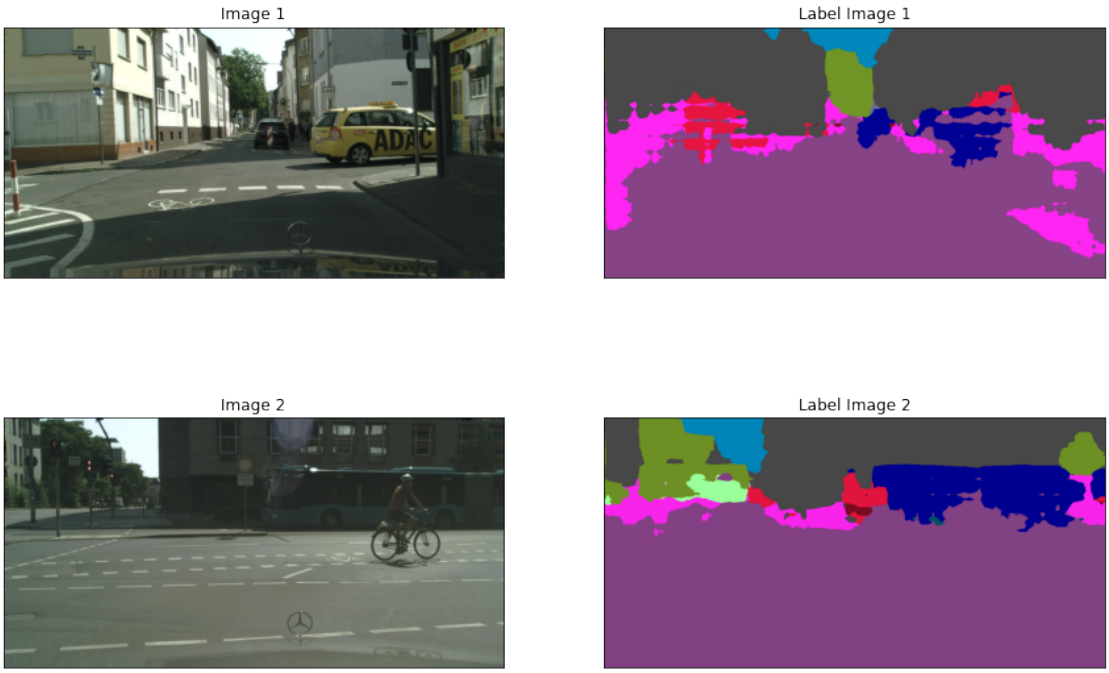
In terms of IOU:

Classes 0, 1, 2, 8, 10, 13, work well - with IOU over 0.4
The rest of the classes don't work so well. They have IOU below 0.4.

5. Can you visualize your results, by plotting out the labels and predictions of the images? Please include at least two examples (HINT: check the unit test in ptsemseg/loader/cityscapes_loader.py)

[9]:
```
one = plt.imread('frankfurt_000000_000294_leftImg8bit.png')
one_label = plt.imread('first_294.png')
two = plt.imread('frankfurt_000001_001464_leftImg8bit.png')
two_label = plt.imread('second_1464.png')

fig = plt.figure(figsize=[15,10])
ax1 = fig.add_subplot(221)
ax1.imshow(one)
ax1.get_xaxis().set_visible(False)
ax1.get_yaxis().set_visible(False)
ax1.set_title('Image 1')
ax2 = fig.add_subplot(222)
ax2.imshow(one_label)
ax2.get_xaxis().set_visible(False)
```

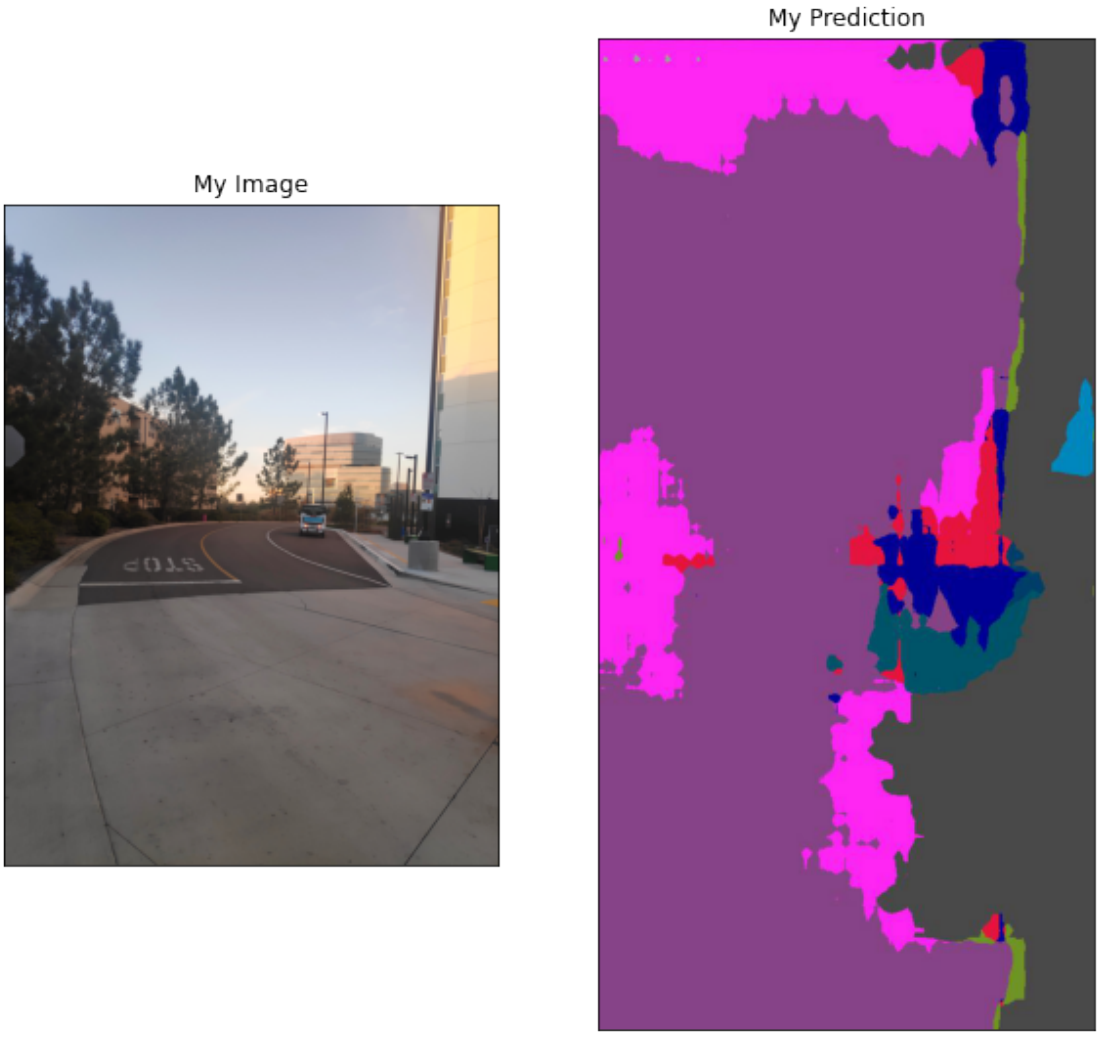
```
ax2.get_yaxis().set_visible(False)
ax2.set_title('Label Image 1')
ax3 = fig.add_subplot(223)
ax3.imshow(two)
ax3.get_xaxis().set_visible(False)
ax3.get_yaxis().set_visible(False)
ax3.set_title('Image 2')
ax4 = fig.add_subplot(224)
ax4.imshow(two_label)
ax4.get_xaxis().set_visible(False)
ax4.get_yaxis().set_visible(False)
ax4.set_title('Label Image 2');
```



6. Please take a photo of a nearby city street, and show the output image from the model. Does the output image look reasonable?

[20]:
```
my_img = plt.imread('my_image.jpg')
my_label = plt.imread('my_prediction.jpg')

my_img = np.transpose(my_img, (1, 0, 2))
my_label = np.transpose(my_label, (1, 0, 2))

my_img = my_img[:,::-1,:]
my_label = my_label[:,::-1,:]
```

```
fig = plt.figure(figsize=[10,10])
ax1 = fig.add_subplot(121)
ax1.imshow(my_img)
ax1.set_title('My Image')
ax1.get_xaxis().set_visible(False)
ax1.get_yaxis().set_visible(False)

ax2 = fig.add_subplot(122)
ax2.imshow(my_label)
ax2.set_title('My Prediction');
ax2.get_xaxis().set_visible(False)
ax2.get_yaxis().set_visible(False)
```



I think the road part of the image has been predicted well. The vehicle too has been predicted

well. The trees and sky have not been predicted well. Overall the prediction is fairly reasonable but a bit worse than I expected.

7. Based on your analysis of the model and training, how can you get better results for prediction? (Give 2 possible options. Change the parameters? Change the network architecture? Or other thoughts? You can checkout the FCN paper [1] and the followup works.)

1. Train on a larger and more varied dataset: Right now, there seems to be bias when it comes to a few labels. This may be because some labels are more common than others in the dataset. If we train on a larger dataset, our model will have less bias than it does right now.
2. Right now, the network works well only in the city road context. If we train on a much more diverse dataset consisting of scenes from all walks of life, then a potential problem would be that the number of potential labels would increase, leading to a lot of false predictions. However, if we first find the context of the image, ie whether it is a road, or a living room, or a restaurant, then based on this, we can narrow down our possible labels (Contextual Encoding)

Reference: https://openaccess.thecvf.com/content_cvpr_2018/papers/Zhang_Context_Encoding_for_CVPR_2

python test.py –model_path ./runs/fcn8s_cityscapes/20201215_232448/fcn8s_cityscapes_best_model.pkl –dataset cityscapes –img_path ./test_2_images/input/frankfurt_000000_000294_leftImg8bit.png –out_path ./test_2_images/output –dcrf 1