Hao Le
A15547504
ECE 172A Winter 2022 HW3

Academic Integrity Policy: Integrity of scholarship is essential for an academic community. The University expects that both faculty and students will honor this principle and in so doing protect the validity of University intellectual work. For students, this means that all academic work will be done by the individual to whom it is assigned, without unauthorized aid of any kind. By including this in my report, I agree to abide by the Academic Integrity Policy.

1. Histograms and Adaptive Histogram Equalization

i.

```python
def computeNormGrayHistogram(inputRGBImage):

    flatSize = inputRGBImage.shape[0] * inputRGBImage.shape[1]

    histogram = np.zeros(32)

    for row in inputRGBImage:
        for col in row:
            luminosity = 0.3 * col[0] +  0.59 * col[1] + 0.11 * col[2] #convert rgb to grayscale using weighted method, yielding luminosity scalar
            binIndex = int(np.floor(luminosity / 8)) #turn grayscale image to bin index of histogram
            histogram[binIndex] = histogram[binIndex] + 1 #increment bin

    histogram = histogram / flatSize #normalize histogram bins by dividing by total pixel count

    return histogram #return histogram as 32 vector
```

ii.

```python
def computeNormRGBHistogram(inputRGBImage):

    flatSize = inputRGBImage.shape[0] * inputRGBImage.shape[1]

    histogram = np.zeros([3,32]) #2d vector, where 3 rows represent the 32 vectors for each channel

    for row in inputRGBImage:
        for col in row:
            for i in range(3): #iterate through RGB
                value = col[i]
                binIndex = int(np.floor(value / 8)) #turn grayscale image to bin index of histogram
                histogram[i][binIndex] = histogram[i][binIndex] + 1 #increment bin corresponding to color

    histogram = histogram / flatSize #normalize histogram bins by dividing by total pixel count

    return np.concatenate((histogram[0],histogram[1],histogram[2]),axis=None) #unfold the 2d array into a 1d array of 96 elements
```

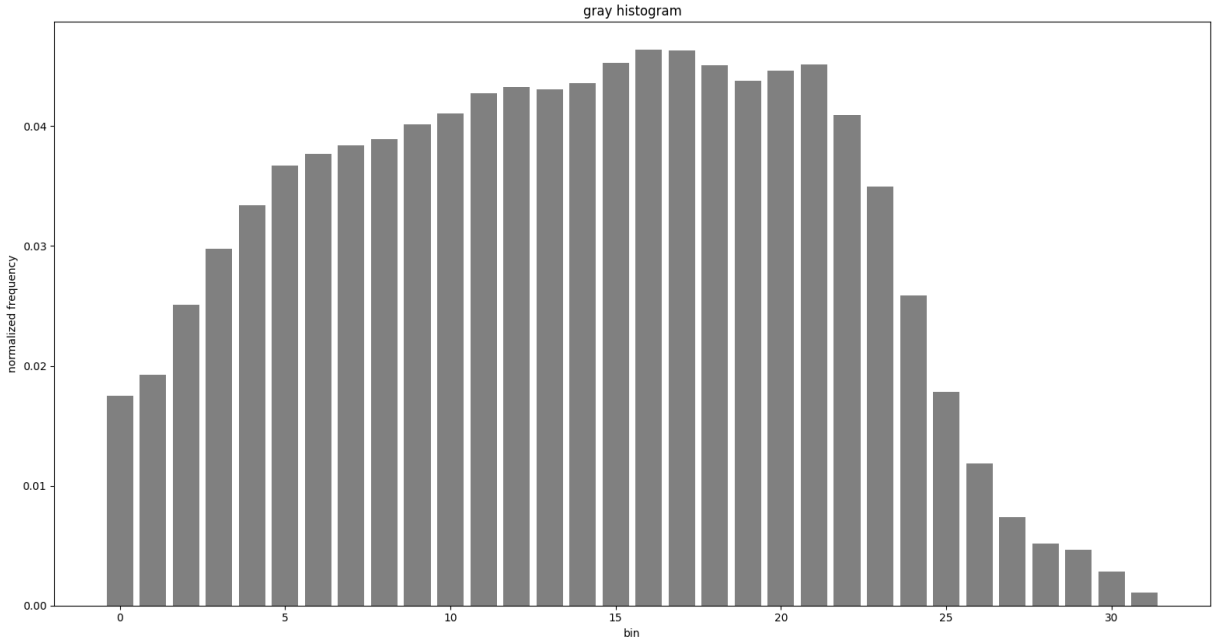
iii.

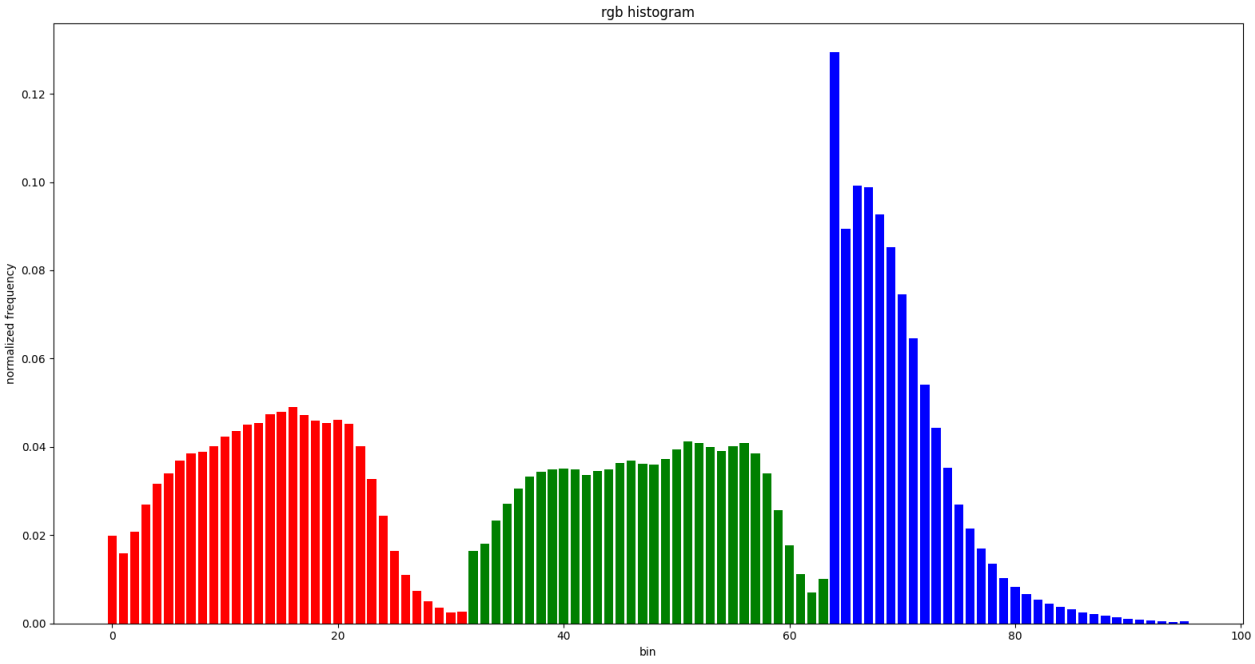
Plotting function was made:

```
def plotHistogram(histogram, title, mode):

    if mode == "gray":
        bins = np.linspace(0,31,32)

        plt.bar(bins,histogram,color='gray')

        plt.title(title)
        plt.xlabel('bin')
        plt.ylabel('normalized frequency')

        plt.show()

    if mode == "rgb":
        bins = np.linspace(0,95,96)

        plt.bar(bins,histogram,color=np.repeat(['red','green','blue'],32))

        plt.title(title)
        plt.xlabel('bin')
        plt.ylabel('normalized frequency')

        plt.show()
```

Main function calls to read image and plot histograms:

```
image = image.imread('forest.jpg')

grayHistogram = computeNormGrayHistogram(image)

plotHistogram(grayHistogram, 'gray histogram', 'gray')

RGBHistogram = computeNormRGBHistogram(image)

plotHistogram(RGBHistogram, 'rgb histogram', 'rgb')

```

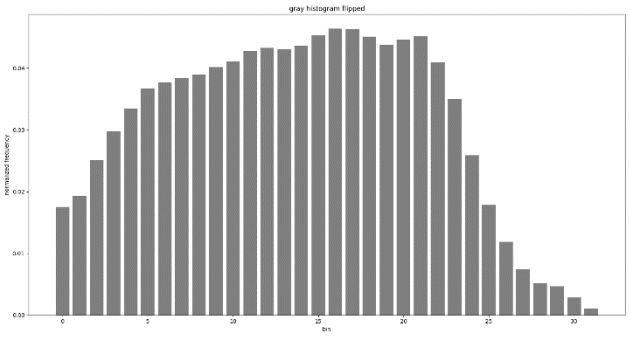

gray histogram
normalized frequency
bin
0.00
0.01
0.02
0.03
0.04
0
5
10
15
20
25
30

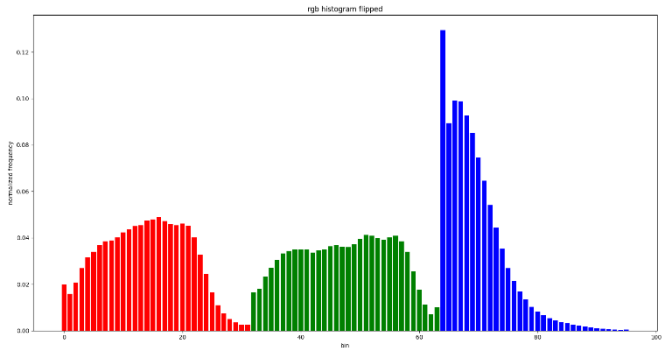

rgb histogram
normalized frequency
bin
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0
20
40
60
80
100

iv.

Main function calls to flip image and plot histograms:

```
160
161    imageFlippedHorizontally = np.fliplr(image)
162
163    grayHistogramFlipped = computeNormGrayHistogram(imageFlippedHorizontally)
164
165    plotHistogram(grayHistogramFlipped, 'gray histogram flipped', 'gray')
166
167    RGBHistogramFlipped = computeNormRGBHistogram(imageFlippedHorizontally)
168
169    plotHistogram(RGBHistogramFlipped, 'rgb histogram flipped', 'rgb')
170
171
```



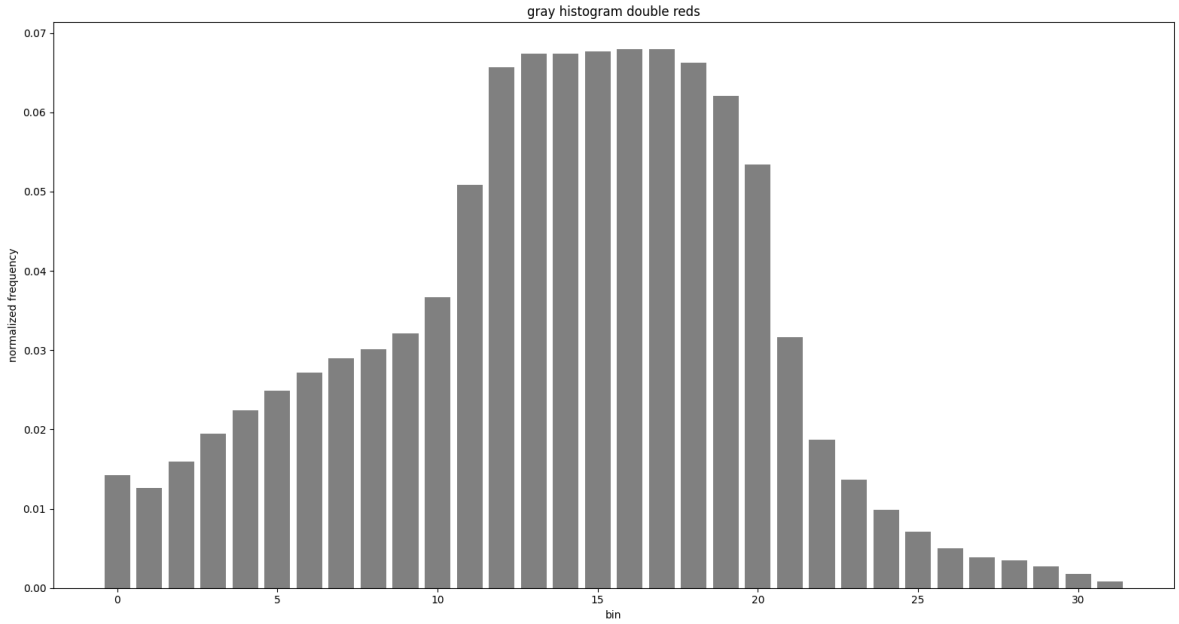


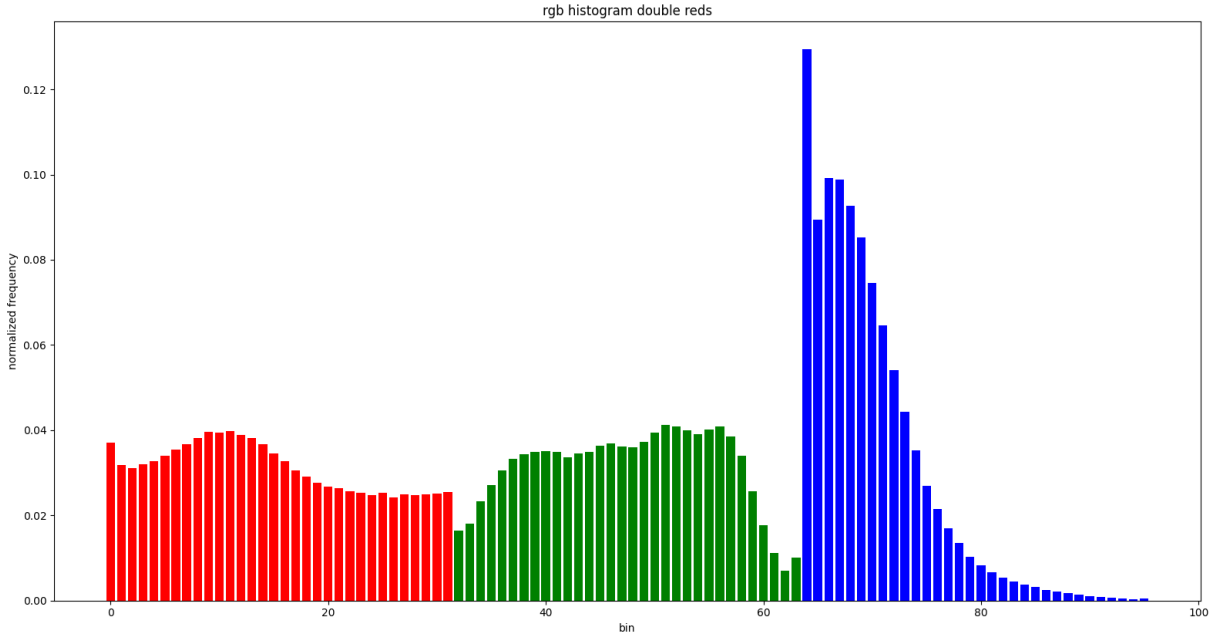
Histograms are invariant to the vertical flip.

v.

Main function calls to double the red channel, show the image, and plot the histograms:

```
75
76    imageDoubleReds = image
77    imageDoubleReds[:,:,0] = 2 * imageDoubleReds[:,:,0]
78
79    plt.imshow(imageDoubleReds)
80    plt.show()
81
82    grayHistogramDoubleReds = computeNormGrayHistogram(imageDoubleReds)
83
84    plotHistogram(grayHistogramDoubleReds, 'gray histogram double reds', 'gray')
85
86    RGBHistogramDoubleReds = computeNormRGBHistogram(imageDoubleReds)
87
88    plotHistogram(RGBHistogramDoubleReds, 'rgb histogram double reds', 'rgb')
89
```





The histograms make sense since 1. Red makes up the mid-range of luminosity, hence we see a boost in that range for the gray plot and 2. We see that the red channel's normalized histogram is more even, meaning that we are getting more amount of red across its value spectrum from 0 to 255. The reason why the normalized frequency did not stay the same is because some values after being doubled go over 255, thus clipping at 255 – this explains why we see the trough at the high range "fill up" and even out.

```python
def AHE(image, winsize): #image is grayscale to pixel values are from 0 to 1 as a float

    imagePadded = np.pad(image, pad_width=((winsize, winsize), (winsize, winsize)), mode='symmetric') #pad around symmetrically with window size which is more than enough

    outputImage = np.zeros(image.shape)

    for row in range(image.shape[0]):

        print(row) #just to keep track of the current row and see progress

        for col in range(image.shape[1]): #by iterating through the row and column, we are going through each pixel coordinate on the original image

            value = image[row][col]

            rank = 0

            x = np.linspace(col - (winsize-1)/2, col + (winsize-1)/2, winsize) #create meshgrid x y values centered around window center
            y = np.linspace(row - (winsize-1)/2, row + (winsize-1)/2, winsize)


            for i in range(winsize):
                for j in range(winsize): #now we are going through all the possible pixels within the window

                    x_padded = int(x[j] + winsize) #add offset to pixel coordinate since we have padded image by the window size all around
                    y_padded = int(y[i] + winsize)

                    sampledWindowValue = imagePadded[y_padded][x_padded]


                    if value > sampledWindowValue:
                        rank = rank + 1

            outputImage[row][col] = rank / (winsize * winsize) #since the max value is 1, we don't see it in the formula


    return outputImage
```

Main function calls to load beach.png, apply premade histogram equalization function, and apply own adaptive HE function with varying window sizes. Also each output image was saved:

```python

    image = image.imread('beach.png')

    imageHE = exposure.equalize_hist(image)


    plt.imsave("HE.png", imageHE, cmap='gray')

    outImage = AHE(image,33)

    plt.imsave("33.png", outImage, cmap='gray')

    outImage = AHE(image,65)

    plt.imsave("65.png", outImage, cmap='gray')

    outImage = AHE(image,129)

    plt.imsave("129.png", outImage, cmap='gray')

```

Original

After applying SciKit-Image's HE function

Image after applying my own AHE function:

Window size: 33

Window size: 65

Window size: 129

After applying both HE and AHE, the contrast of the image is significantly boosted – we see this especially in the texture of the ground and the house. Also, we see dark areas such as the window where the people are and underneath the house get brightened up – for AHE with window size 129, we can actually make out their faces, and even notice the fourth person who was barely seen in the original image.

HE in this case does not greatly distort the original image to the point where it is not aesthetically pleasing anymore. The sky is clipped greatly, but other than that, the brightness boost is subtle. It still fails in brightening up the window area where the people subjects are.

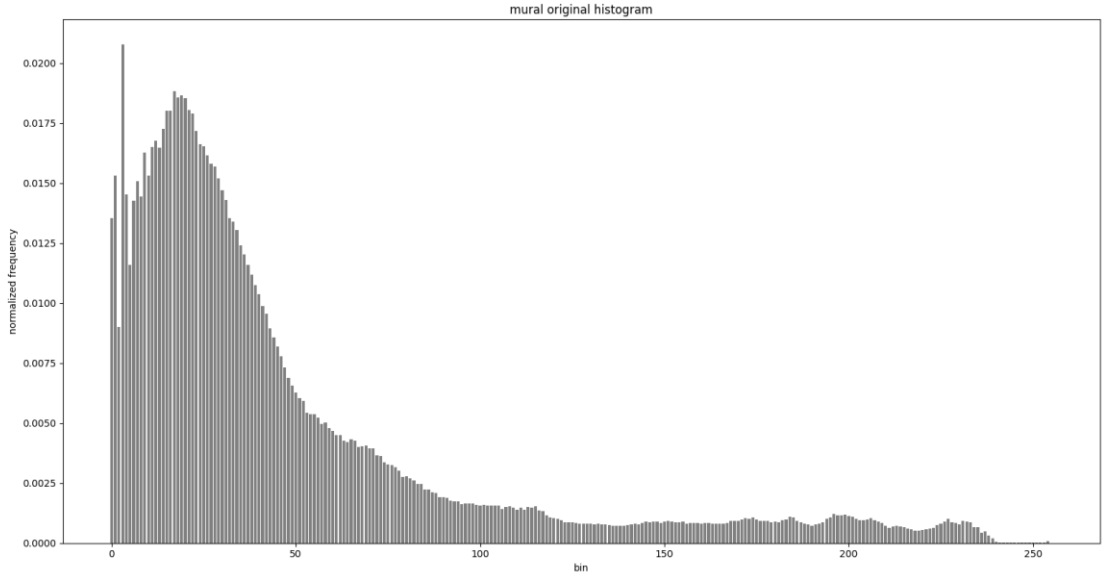
AHE performs best with the window size of 129, or ideally bigger, since it does not create over contrasted areas due to the window size being small, thus creating large-gapped quantization of pixel intensities. It still suffers in clipping of the faces, and the sky is also clipped to darkness because of how even it is in the original image – the algorithm automatically turns this area of all white (pixel value of 1) to 0 since the rank is always 0. Overall, AHE does a good job at boosting the texture of the house and the ground, giving the image more even luminosity range.

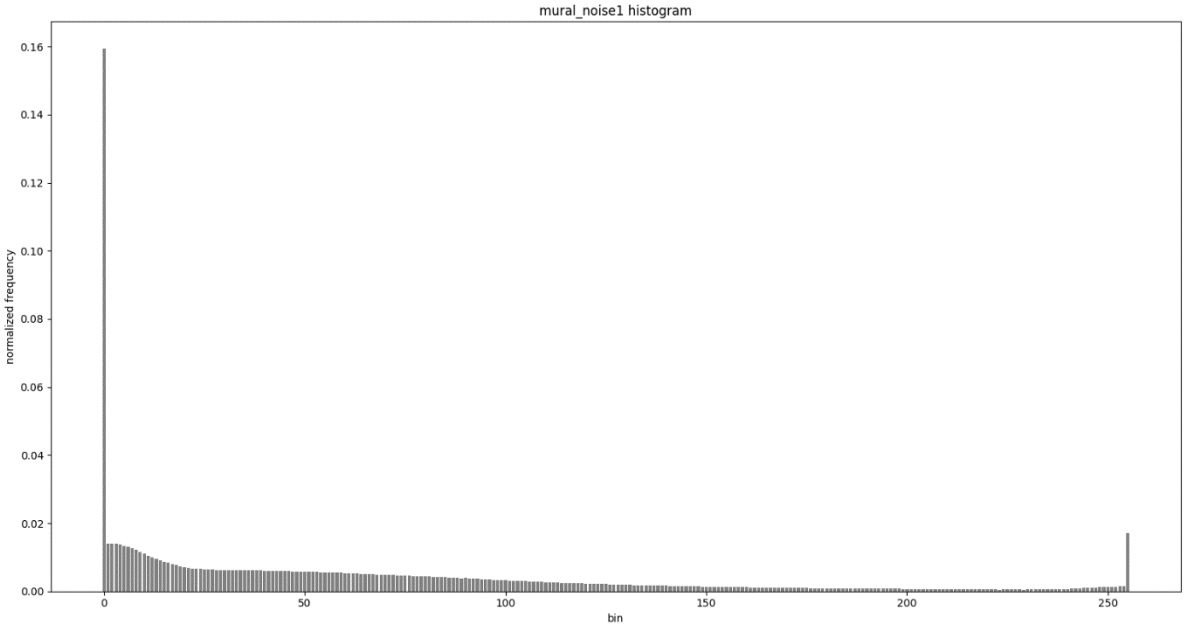
Although in this case AHE performs better than HE in the beach image, images where there are large bright areas of even value can cause AHE to inadvertently darken those areas unnaturally. Moreover, images with already good contrast to begin with can get over contrasted by AHE, especially with smaller window sizes. Those images are better with the subtle equalization that HE offers to get more luminosity range.

2. Filtering

1. The mural can be found at Price Center

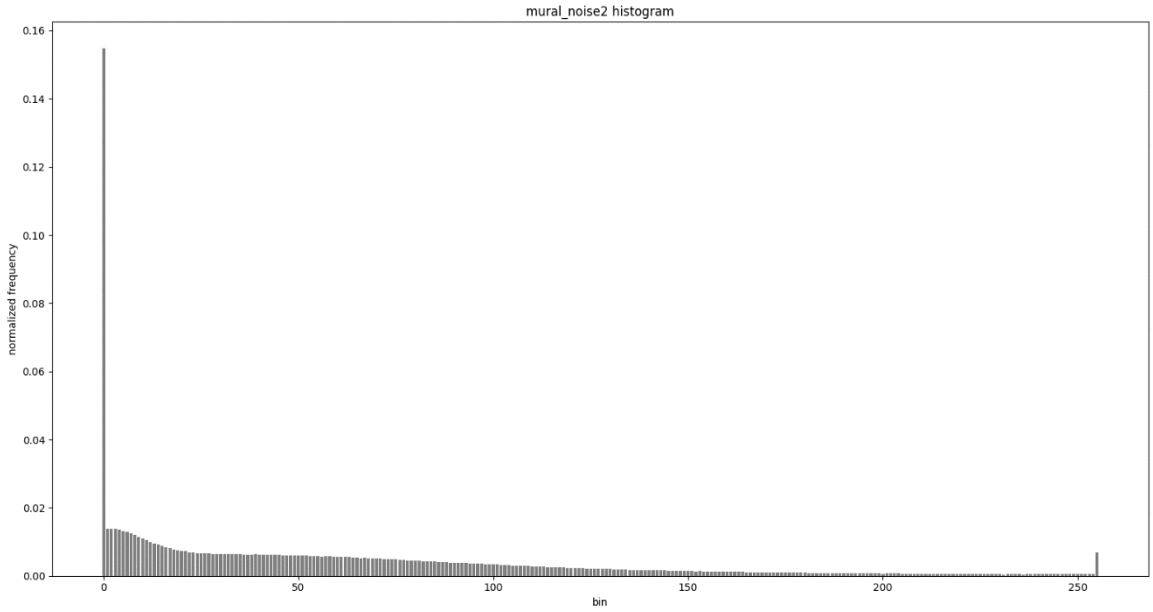
2.





There is a disproportionate occurrence of 0 (black) and 255 (white) pixels, compared to the histogram of the original image, implying the image is corrupted with salt and pepper noise. Moreover, we see higher frequencies at the high range beside from 255, pointing to the presence of some Gaussian noise with a mean near that range.

3.



Like mural_noise1.jpg, there is disproportionate occurrence of both white and black pixels. However, we do not see any perturbance for the other values. Thus, there is only salt and pepper noise.

4.

Both the mean and median filter implementations reused a good deal of the AHE function due to the sliding kernel functionality. Moreover, computation was improved by replacing the two nested for loops to iterate over the pixels of the kernel with numpy slicing. The images were padded symmetrically.

```python
def meanFilter(inputImage, kernelSize): #input image is RGB

    imagePadded = np.pad(inputImage, pad_width=((kernelSize, kernelSize), (kernelSize, kernelSize)), mode='symmetric') #pad around symmetrically with window size which is more than enough

    outputImage = np.zeros(inputImage.shape)

    for row in range(inputImage.shape[0]):


        for col in range(inputImage.shape[1]): #by iterating through the row and column, we are going through each pixel coordinate on the original image

            value = inputImage[row][col]

            rowOffset = row + kernelSize
            colOffset = col + kernelSize #center of the kernel on the padded image space


            kernelValues = imagePadded[int(rowOffset - (kernelSize-1)/2) : int(rowOffset + (kernelSize-1)/2 + 1), int(colOffset - (kernelSize-1)/2) : int(colOffset + (kernelSize-1)/2 + 1)] #extract the kernel by slicing

            kernelValues = np.sort(kernelValues.flatten())

            outputImage[row][col] = np.average(kernelValues) #new pixel value is the average of the kernel

    return outputImage
```

```python
def medianFilter(inputImage, kernelSize): #input image is RGB

    imagePadded = np.pad(inputImage, pad_width=((kernelSize, kernelSize), (kernelSize, kernelSize)), mode='symmetric') #pad around symmetrically with window size which is more than enough

    outputImage = np.zeros(inputImage.shape)

    for row in range(inputImage.shape[0]):


        for col in range(inputImage.shape[1]): #by iterating through the row and column, we are going through each pixel coordinate on the original image

            value = inputImage[row][col]

            rowOffset = row + kernelSize
            colOffset = col + kernelSize #center of the kernel on the padded image space


            kernelValues = imagePadded[int(rowOffset - (kernelSize-1)/2) : int(rowOffset + (kernelSize-1)/2 + 1), int(colOffset - (kernelSize-1)/2) : int(colOffset + (kernelSize-1)/2 + 1)] #extract the kernel by slicing

            kernelValues = np.sort(kernelValues.flatten()) #sort in ascending order to take middle value as median

            outputImage[row][col] = kernelValues[int(np.floor((kernelSize*kernelSize)/2))] #extract the middle value of the kernel values to get median


    return outputImage
```

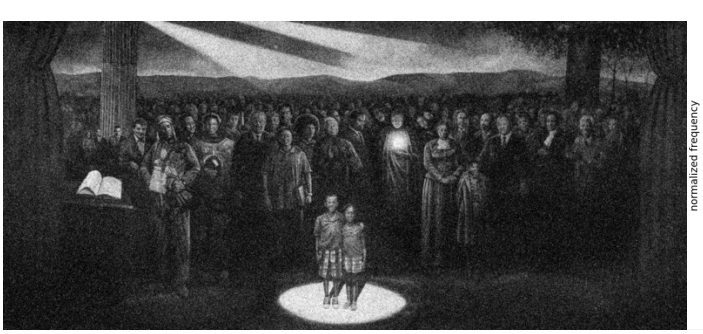
The main script is as follows, and performs four experiments on both noisy mural images. It saves the filtered images, and logs the computation times:

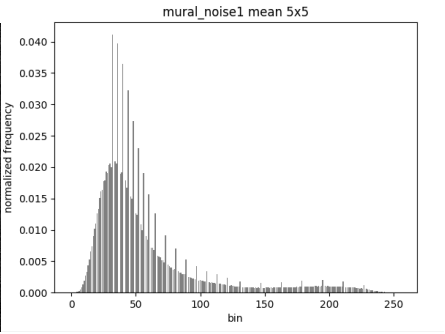
```python
def runFilters(imagePath,name):

    image_ = cv2.imread(imagePath)
    imageGray = cv2.cvtColor(image_,cv2.COLOR_BGR2GRAY)

    start = time.time()
    imageFiltered = meanFilter(imageGray,5)
    plt.imsave(name + "mean5.png", imageFiltered, cmap='gray')
    end = time.time()
    print(imagePath + " 5x5 mean filter took " + str(end - start) + " seconds")

    start = time.time()
    imageFiltered = meanFilter(imageGray,21)
    plt.imsave(name + "mean21.png", imageFiltered, cmap='gray')
    end = time.time()
    print(imagePath + " 21x21 mean filter took " + str(end - start) + " seconds")

    start = time.time()
    imageFiltered = medianFilter(imageGray,5)
    plt.imsave(name + "median5.png", imageFiltered, cmap='gray')
    end = time.time()
    print(imagePath + " 5x5 median filter took " + str(end - start) + " seconds")

    start = time.time()
    imageFiltered = medianFilter(imageGray,21)
    plt.imsave(name + "median21.png", imageFiltered, cmap='gray')
    end = time.time()
    print(imagePath + " 21x21 median filter took " + str(end - start) + " seconds")


runFilters("mural_noise1.jpg", "mural_noise1")
runFilters("mural_noise2.jpg", "mural_noise2")
```

```
mural_noise1.jpg 5x5 mean filter took 42.638566970825195 seconds
mural_noise1.jpg 21x21 mean filter took 66.27328395843506 seconds
mural_noise1.jpg 5x5 median filter took 17.45603084564209 seconds
mural_noise1.jpg 21x21 median filter took 41.865397930145264 seconds
mural_noise2.jpg 5x5 mean filter took 42.253971338272095 seconds
mural_noise2.jpg 21x21 mean filter took 66.2686026096344 seconds
mural_noise2.jpg 5x5 median filter took 17.400970458984375 seconds
mural_noise2.jpg 21x21 median filter took 41.68757152557373 seconds
```

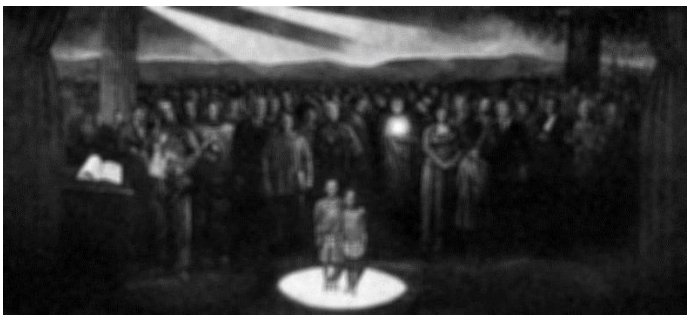
Mural_noise1

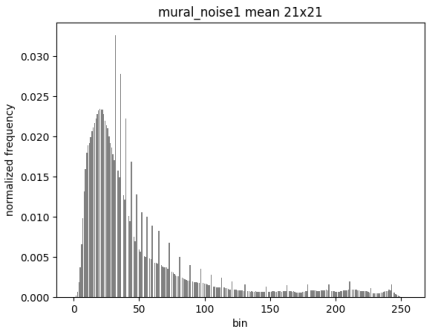
Mean 5x5



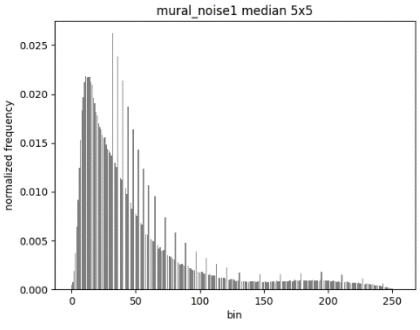


Mean 21x21

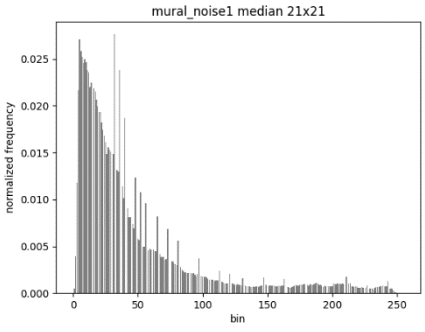




Median 5x5

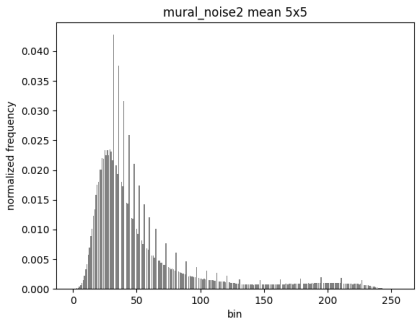


Median 21x21

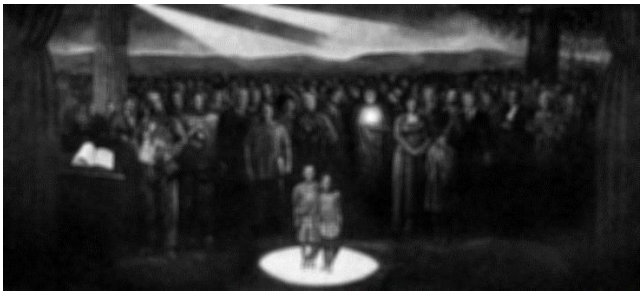


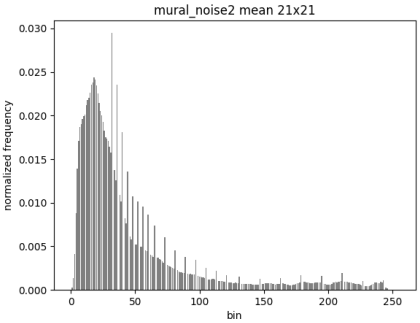
Mural_noise2

Mean 5x5

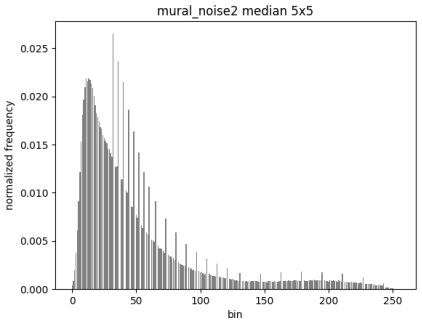


Mean 21x21

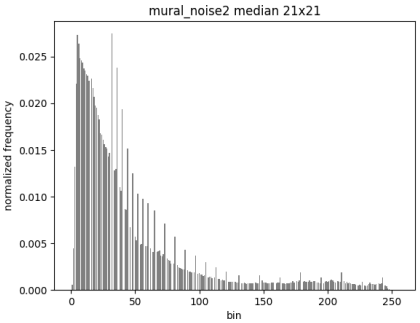




Median 5x5



Median 21x21



5.

From the log, 5x5 median filter performs the fastest, followed by 21x21 media, 5x5 mean, and 21x21 mean.

6.

I would use median 21x21 filter because the salt and pepper noise seem to go away, with the side effects of the whole mural being more contrasted, which aesthetically looks better anyways, and the details of the people and objects fading – the image is unsharpened. The mean 21x21 filter also does a good and similar job, though does not have as strong of a contrasting effect.

7.

I would use median 21x21 filter for the same reasons in 6, though I can picture using the 5x5 median filter for both since the noise adds an artist "grain" effect to it.

8.

$$\sum_{i=0}^{height-1} \sum_{j=0}^{width-1} (f(i,j) - i(i,j))^2$$

This expression assumes that the i and j's are indexing the corresponding absolute coordinates of the original image when the sliding window is over a certain portion i.e. i and j need not be "offset" for template image, but need to for original image depending on where the top left corner of the template is.

This expression is evaluating the square error of the template vs. the original. A match would be minimizing this mean squared error.

9.

We can minimize the whole expression by minimizing what is being summed:

$$(f(i,j) - i(i,j))^2 = f^2 - 2fi + i^2$$

So to minimize the original expression, we want to maximize *2fi*

Code for both 10 and 11:

```
template = cv2.imread("template.jpg",0)
mural = cv2.imread("mural.jpg",0)
result = cv2.matchTemplate(mural,template,cv2.TM_CCORR_NORMED)
minVal, maxVal, minLoc, maxLoc = cv2.minMaxLoc(result)
mural = cv2.rectangle(mural, maxLoc - np.array([50,50]), maxLoc + np.array([50,50]),(255,255,255),3)
plt.imshow(mural,cmap="gray")
plt.show()

template = cv2.imread("template.jpg",0)
mural = cv2.imread("mural.jpg",0)
result = cv2.matchTemplate(mural,template,cv2.TM_CCORR)
minVal, maxVal, minLoc, maxLoc = cv2.minMaxLoc(result)
mural = cv2.rectangle(mural, maxLoc - np.array([50,50]), maxLoc + np.array([50,50]),(255,255,255),3)
plt.imshow(mural,cmap="gray")
plt.show()
```

10.

The matching function thinks that the template matches where the candle is. This is a bright spot on the image. Since we are trying to maximize 2fi, the ideal multiplier with the template is the highest value, which is 255. Thus, when the sliding template passes by that area, it will get multiplied by the all 255 values of the candle, outputting the highest match values there.

11.

Now the matching function is actually working. Since it is normalized to the intensity value of both the template and the windowed area of the original image, this scales back the output match value to be relative to the intensities of the window. Since values can only go from 0 to 255, really bright areas of the original image will cause the match value to clip and 255-out. By normalizing, this keeps the match value in a reasonable range, and puts more emphasis on the matching of features.

3. Canny Edge Detector

1. A GaussianSmoothing function is made to perform the sliding kernel operation over the image with the Gaussian weights, followed by normalization:

```python
def GaussianSmoothing(inputImage):

    kernel = (1/159) * np.array([[2,4,5,4,2],[4,9,12,9,4],[5,12,15,12,5],[4,9,12,9,4],[2,4,5,4,2]])

    kernelSize = kernel.shape[0]

    imagePadded = np.pad(inputImage, pad_width=((kernelSize, kernelSize), (kernelSize, kernelSize)), mode='symmetric') #pad around symmetrically with window size which is mo
    than enough

    outputImage = np.zeros(inputImage.shape)

    for row in range(inputImage.shape[0]):

        print(row)


        for col in range(inputImage.shape[1]): #by iterating through the row and column, we are going through each pixel coordinate on the original image

            value = inputImage[row][col]

            rowOffset = row + kernelSize
            colOffset = col + kernelSize #center of the kernel on the padded image space

            kernelSampledValues = imagePadded[int(rowOffset - (kernelSize-1)/2) : int(rowOffset + (kernelSize-1)/2 + 1), int(colOffset - (kernelSize-1)/2) : int(colOffset +
            (kernelSize-1)/2 + 1)] #extract the kernel by slicing

            elementwiseMultiply = np.multiply(kernel,kernelSampledValues) #multiply kernel values by Gaussian weights

            outputImage[row][col] = np.sum(elementwiseMultiply) #return the sum of the kernel values

    return outputImage
```

The main script is as follows:

```
155   lane = cv2.imread("lane.png",0)
156   laneSmooth = GaussianSmoothing(lane)
157   laneMag,laneAngle,laneComb = Sobel(laneSmooth)
158   laneMagSuppressed = NMS(laneMag,laneAngle)
159
160   plt.imshow(laneSmooth,cmap="gray")
161   plt.show()
```

It reads the lane.png image and performs the Gaussian smoothing filter and shows the results, along with other operations that will be discussed in the proceeding problems.

The output image:

The image is free of the original grid-like noise, and the edges are well-preserved, an advantage of the Gaussian filter over a mean filter.

2. A Sobel function is made:

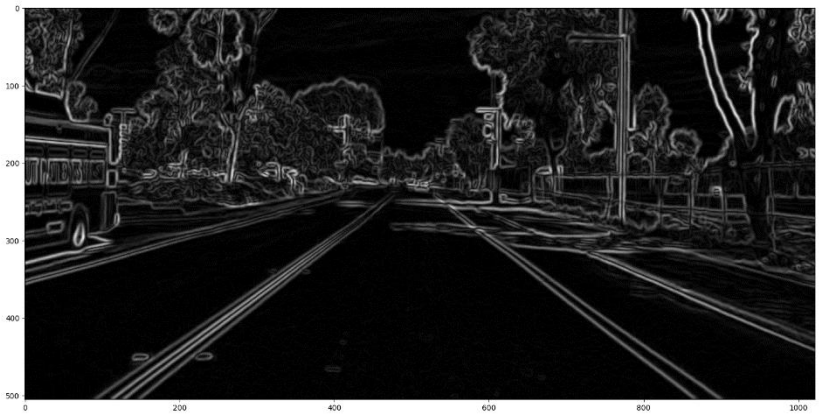
```python
def Sobel(inputImage):

    kx = np.array([[-1,0,1],[-2,0,2],[-1,0,1]])
    ky = np.array([[-1,-2,-1],[0,0,0],[1,2,1]]) #gradient kernels

    kernelSize = kx.shape[0]

    imagePadded = np.pad(inputImage, pad_width=((kernelSize, kernelSize), (kernelSize, kernelSize)), mode='symmetric') #pad around symmetrically with window size which is more than enough


    outputGradientMagnitudes = np.zeros(inputImage.shape)
    outputGradientAngles = np.zeros(inputImage.shape)
    outputCombinedImage = np.zeros(inputImage.shape)


    for row in range(inputImage.shape[0]):

        print(row)

        for col in range(inputImage.shape[1]): #by iterating through the row and column, we are going through each pixel coordinate on the original image

            value = inputImage[row][col]

            rowOffset = row + kernelSize
            colOffset = col + kernelSize #center of the kernel on the padded image space

            kernelSampledValues = imagePadded[int(rowOffset - (kernelSize-1)/2) : int(rowOffset + (kernelSize-1)/2 + 1), int(colOffset - (kernelSize-1)/2) : int(colOffset + (kernelSize-1)/2 + 1)] #extract the kernel by slicing

            elementwiseMultiplyKx = np.multiply(kx,kernelSampledValues)
            Gx = np.sum(elementwiseMultiplyKx) #multiply by kernel, then sum values to get value of gradient

            elementwiseMultiplyKy = np.multiply(ky,kernelSampledValues)
            Gy = np.sum(elementwiseMultiplyKy)

            outputGradientMagnitudes[row][col] = np.sqrt(np.square(Gx) + np.square(Gy)) #get magnitude by performing Pythagorean theorem
            outputGradientAngles[row][col] = np.arctan(Gy / Gx) #get angle in radians with arctan; this is limited to -pi/2 to pi/2

            if outputGradientMagnitudes[row][col] > 100: #to build the magnitude-gradient image, we only want to show the pixels above a certain threshold, since angle image alone is very noisy
                outputCombinedImage[row][col] = (outputGradientAngles[row][col] + (np.pi / 2)) / (np.pi) #first add pi/2 to angle so now it is always positive from 0 to pi, then normalize to 0-1 range by dividing by pi

    return outputGradientMagnitudes, outputGradientAngles, outputCombinedImage #return all three images containing different information extracted using Sobel
```

This will return three images:

- Magnitude image where each pixel value represents the magnitude of the gradient vector
- Angle image where each pixel value represents the angle of the gradient vector, limited to a range of (-pi/2, pi/2) by nature of arctan
- Magnitude-angle image that is similar to the angle image, only showing the edges more cleanly

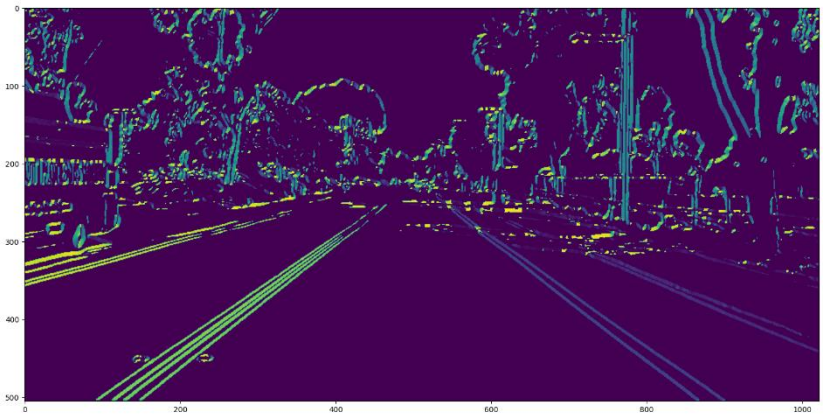
The main script is as follows to show the three images:

```
plt.imshow(laneMag,cmap="gray")
plt.show()
plt.imshow(laneAngle,cmap="gray")
plt.show()
plt.imshow(laneComb,cmap="viridis")
plt.show()
```



Magnitude

Angle



Magnitude-Angle

From the output, the Sobel operation is working by extracting the prominent edges in the original image. Though the angle image is noisy and unintuitive, the magnitude-angle image does a good job of showing the range of angles for each edge. For example, we can see that the straight lines of the road all take on a single color, implying that the angle stays consistent throughout which is expected. We can also see on the bus wheel the range of colors being cycled which is also expected for a circular shape.

3.

A non-maximum suppression function is made:

```
def NMS(inputMagnitudesImage,inputAngleImages):

    inputMagnitudesImagePadded = np.pad(inputMagnitudesImage, pad_width=((1,1), (1,1)), mode='symmetric') #pad 1 pixel around image symmetrically so we can go value compare in gradient direction along edges

    outputMagntiudeImage = np.zeros(inputAngleImages.shape)

    for row in range(inputMagnitudesImage.shape[0]):

        print(row)

        for col in range(inputMagnitudesImage.shape[1]): #by iterating through the row and column, we are going through each pixel coordinate on the original image

            rowOffset = row + 1
            colOffset = col + 1 #absolute coordinates in padded image space; basically just add 1 to both
            pixelValueTarget = inputMagnitudesImagePadded[rowOffset][colOffset]

            direction = getAngleDirection(inputAngleImages[row][col]) #get value from 0 to 3 to determine direction of gradient and which neighboring pixels to check

            if direction == 0: #NS
                pixelValue1 = inputMagnitudesImagePadded[rowOffset+1][colOffset]
                pixelValue2 = inputMagnitudesImagePadded[rowOffset-1][colOffset]
                if pixelValue1 > pixelValueTarget or pixelValue2 > pixelValueTarget: #if either of the neighboring pixels has a higher value, the target pixel cannot be a local max, so suppress it
                    outputMagntiudeImage[row][col] = 0
                else:
                    outputMagntiudeImage[row][col] = pixelValueTarget

            if direction == 1: #SE-NW
                pixelValue1 = inputMagnitudesImagePadded[rowOffset-1][colOffset+1]
                pixelValue2 = inputMagnitudesImagePadded[rowOffset+1][colOffset-1]
                if pixelValue1 > pixelValueTarget or pixelValue2 > pixelValueTarget:
                    outputMagntiudeImage[row][col] = 0
                else:
                    outputMagntiudeImage[row][col] = pixelValueTarget

            if direction == 2: #EW
                pixelValue1 = inputMagnitudesImagePadded[rowOffset][colOffset+1]
                pixelValue2 = inputMagnitudesImagePadded[rowOffset][colOffset-1]
                if pixelValue1 > pixelValueTarget or pixelValue2 > pixelValueTarget:
                    outputMagntiudeImage[row][col] = 0
                else:
                    outputMagntiudeImage[row][col] = pixelValueTarget

            if direction == 3: #NE-SW
                pixelValue1 = inputMagnitudesImagePadded[rowOffset+1][colOffset+1]
                pixelValue2 = inputMagnitudesImagePadded[rowOffset-1][colOffset-1]
                if pixelValue1 > pixelValueTarget or pixelValue2 > pixelValueTarget:
                    outputMagntiudeImage[row][col] = 0
                else:
                    outputMagntiudeImage[row][col] = pixelValueTarget

    return outputMagntiudeImage
```

The function uses a getAngleDirection function which takes in the angle output of arctan, and outputs which direction of the 8-count neighboring directions the angle belongs to:

```python
def getAngleDirection(inputAngle): #angle is output of arctan in radians, so is limited to -pi/2 and pi/2

    #return direction enums: 0: NS, 1:SE-NW, 2:EW, 3:NE-SW

    inputAngleOffset = inputAngle + np.pi/2 #add pi/2 so now angle is within 0 to pi and always positive

    rounded = np.round(inputAngleOffset / (np.pi/4)) #divide by 45 degrees and round to nearest integer to determine which direction is nearest

    if rounded == 0 or rounded == 4:
        return 0
    else:
        return rounded
```

NMS will compare the neighboring pixels, in the direction of the gradient, of each pixel in the magnitude image, and determine whether the target pixel is a local maximum in value. If not, then suppress the pixel by setting the value to 0. The result is that we get 1 pixel-wide, clean edges:

```python
plt.imshow(laneMagSuppressed,cmap="gray")
plt.show()
```

4.

Now the NMS image is thresholded with a pixel value of 110 i.e., any pixel value in the image that is lower than or equal to 110 gets set to 0:

```
164    plt.imshow((laneMagSuppressed > 110) * laneMagSuppressed,cmap="gray")
165    plt.show()
```

The result is cleaner than the NMS image because we removed the faint edges on the road and especially the trees. Plus, the threshold value picked preserved the outline of the road which is critical for autonomous driving vision.

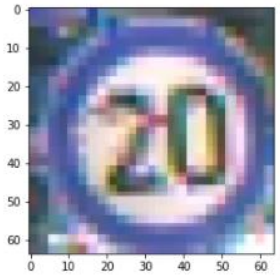
4. Convolutional Neural Network

1.

The code was modified so two types of traffic signs are extracted from the unrandomized training images and shown. The one-hot encoded labels are printed, and we can see that the first sign is of class 0, and the second is of class 26.

In [32]:
```
input_array = np.stack(list_images)
train_y = to_categorical(output)

randomize = np.arange(len(input_array))
np.random.shuffle(randomize)

print(train_y.shape)

plt.figure()
plt.imshow(input_array[42,:])
print(train_y[42])
plt.figure()
plt.imshow(input_array[2536,:])
print(train_y[2536])
x = input_array[randomize]
y = train_y[randomize]
```

```
(4300, 43)
[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```



3.

The interpolation argument is to specify the method use in interpolation – this is the process of estimating a pixel value between two known values. This is used in the resize function because if an image is smaller than 64x64, it will have to be upscaled, thus some pixel values will have to be estimated.

4.

From the code

```
0]: split_size = int(x.shape[0]*0.6)
    train_x, val_x = x[:split_size], x[split_size:]
    train1_y, val_y = y[:split_size], y[split_size:]

    split_size = int(val_x.shape[0]*0.5)
    val_x, test_x = val_x[:split_size], val_x[split_size:]
    val_y, test_y = val_y[:split_size], val_y[split_size:]
```

We can see first we are splitting the training images and labels, x and y, by a 0.6:0.4 ratio, where 0.6*total training images goes to the training set, and 0.4*total training images goes to the validation set.

Subsequently, we take the validation set, and split that by a 0.5:0.5 ratio, where 0.5 goes back to the validation set, and 0.5 goes to the test set.

Hence, overall we get a 0.6:0.2:0.2 ratio between training:validation:test sets.

Since the overall amount of training images is the size of x, this is 4300, so training, validation, and testing get 2580, 860, and 860 images respectively.

We shuffle the original training images to get different results in the network, since it is a gradient problem and results will differ depending on initialization. This is done prior to splitting so the randomness is consistent over all split sets, not within the sets which is less random.

5.

i.

A convolution layer performs a sliding kernel operation over each pixel location of the image, like a Gaussian blur filter, but we do not normalize after multiplying the kernel weights with the sampled kernel values and summing them up. Moreover, the weights in the kernel are not static, but are variables that are learned. Each pixel location will produce an output pixel value, producing a new image once passed through the filter.

ii.

```
Conv2D(16, (3, 3), activation='relu', input_shape=(64,64,3), padding='same'),
```

In this case, Conv2D is specifying a portion of the neural network architecture. This is the first portion that takes in the image, hence input_shape is specified to take an RGB, 3 channel, 64x64 image as a 3D array. The first argument, 16, specifies the number of convolution filters. (3,3) specifies the dimensions of the sliding kernel. Lastly, since padding is set to 'same', the image is padded all around with the edge values, so the sliding kernel can produce values even on the edge pixel positions. This will lead to no reduced size after going through a convolution layer. To sum up, this is 16 convolution filters that use a 3x3 sliding kernel; each filter pads the image so no size reduction occurs. The output after the 16 filters is still a 64x64x3 RGB image, but it goes through an activation function specified as 'relu'. This is the rectified linear unit, which takes in each scalar value of the image, and outputs a scalar value based on the ReLU function to add nonlinearity. In essence, any values on the output of the convolution filters that are less than 0 get zeroed out.

iii.

Maxpooling uses a sliding kernel operation over a 2D array, and for each kernel will take the largest value and set that as the output pixel for the target pixel location. Typically, this will be done without padding, so depending on the kernel size, the edge pixels of the original array can not be the center of the kernel. The output of maxpooling will be a downsampling of the original array i.e., the output array will be smaller in dimension.

iv.

A dense layer is where the nodes of a given layer are directly connected to the nodes of the next. Moreover, each node of the next layer will be affected by every node of the previous, attenuated or boosted by a weight and bias. Visually, this looks like a "dense" rats nest of wired connections that sort of represent how neurons are connected. This is different from a convolution layer because there is no sliding kernel operation happening. The weights and biases to be learned here are not of the kernel, which is usually a small amount, but instead of all of the connections between the nodes, which is usually a large amount.

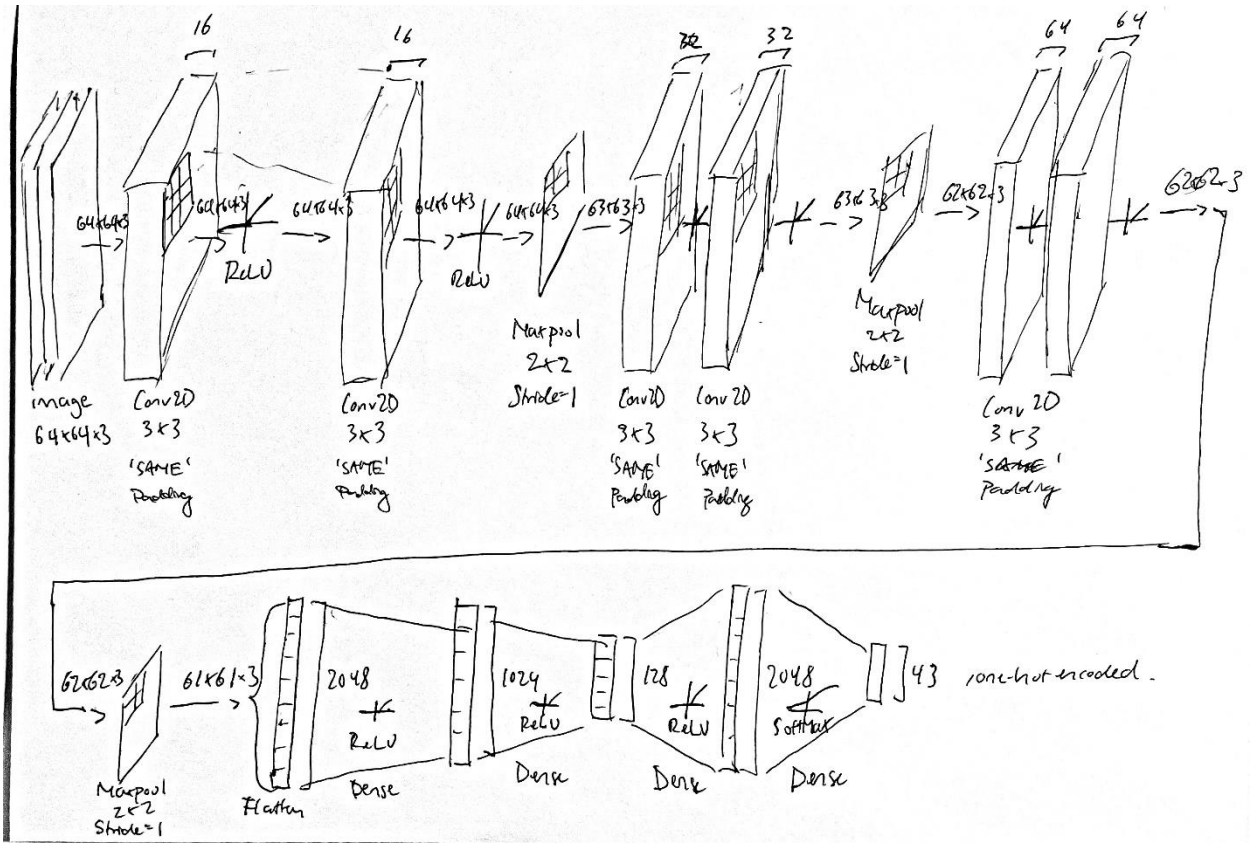
v.

An epoch is one cycle of the neural network through the total training set. For example, if the training set has 10 images, and the network has done 5 epochs, then the network has seen the same 10 images 5 times, or has seen 50 images in total.

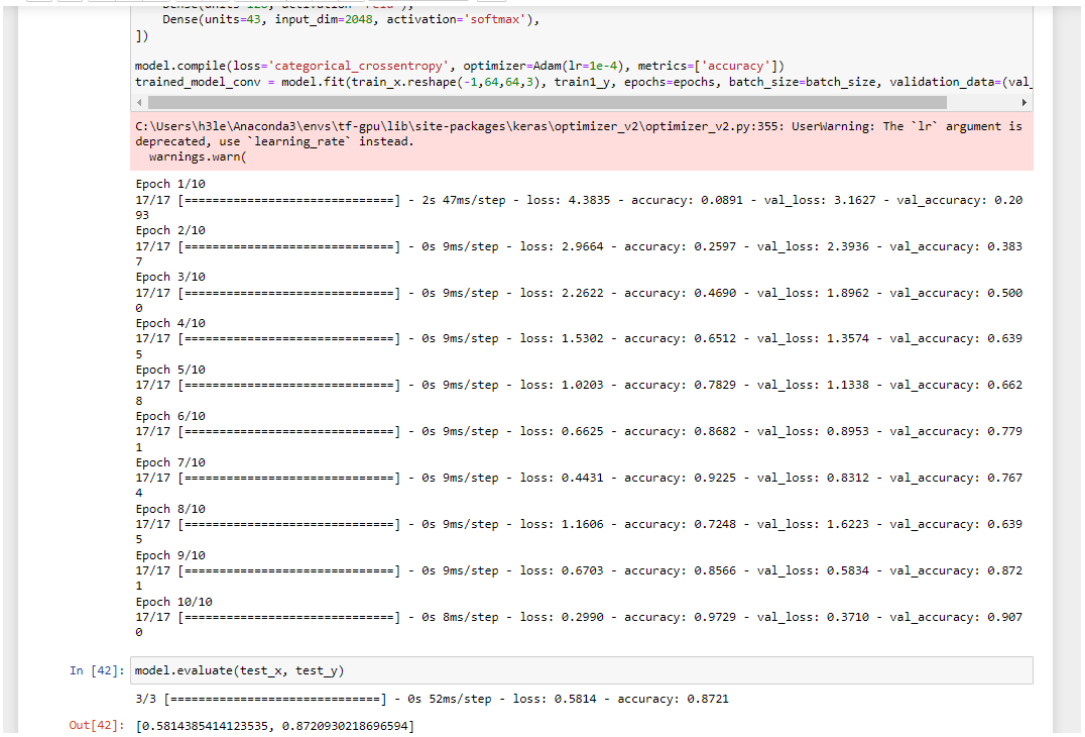
vi.

We are using the categorical cross-entropy loss function because it is appropriate for multi-class classification where the output label is a one-hot encoded array. Our problem is a classification problem of 43 classes, so it works for us.

vii.



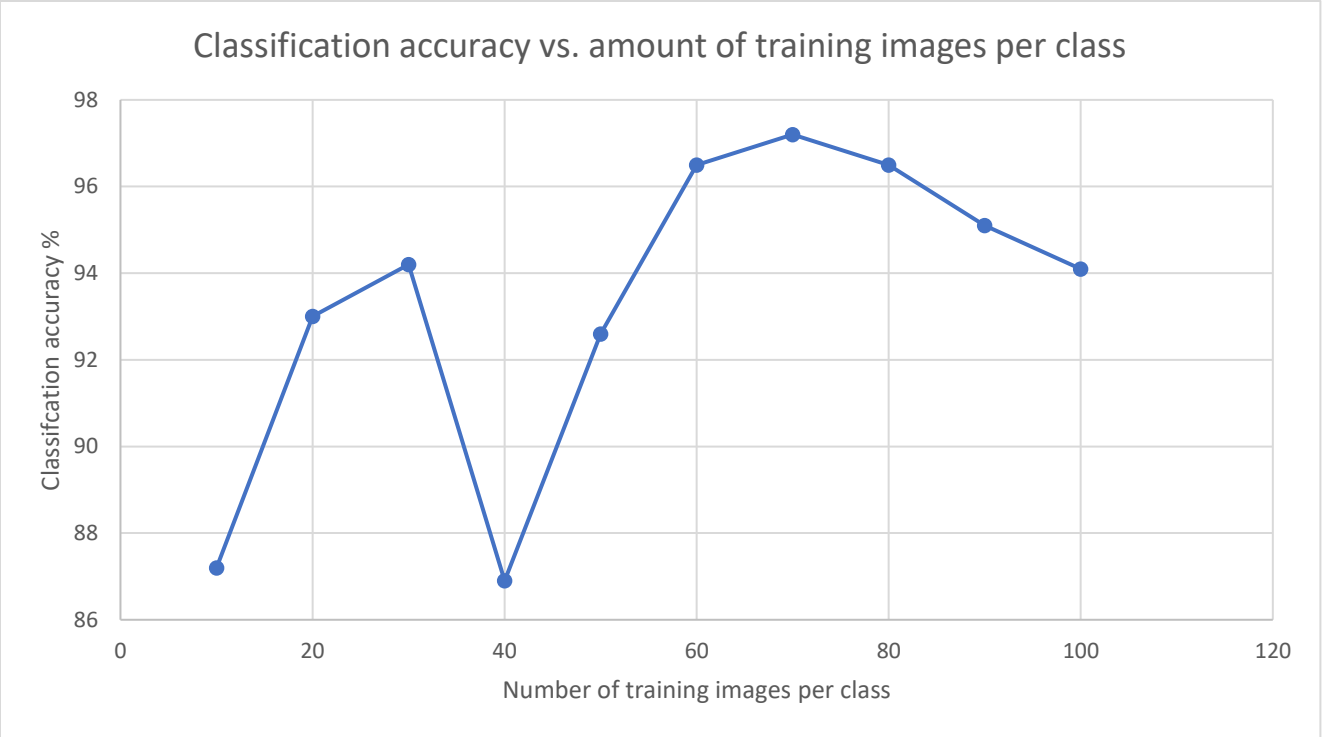
7.



```
    Dense(units=43, input_dim=2048, activation='softmax'),
])

model.compile(loss='categorical_crossentropy', optimizer=Adam(lr=1e-4), metrics=['accuracy'])
trained_model_conv = model.fit(train_x.reshape(-1,64,64,3), train1_y, epochs=epochs, batch_size=batch_size, validation_data=(val

C:\Users\h3le\Anaconda3\envs\tf-gpu\lib\site-packages\keras\optimizer_v2\optimizer_v2.py:355: UserWarning: The `lr` argument is
deprecated, use `learning_rate` instead.
  warnings.warn(

Epoch 1/10
17/17 [==============================] - 2s 47ms/step - loss: 4.3835 - accuracy: 0.0891 - val_loss: 3.1627 - val_accuracy: 0.20
93
Epoch 2/10
17/17 [==============================] - 0s 9ms/step - loss: 2.9664 - accuracy: 0.2597 - val_loss: 2.3936 - val_accuracy: 0.383
7
Epoch 3/10
17/17 [==============================] - 0s 9ms/step - loss: 2.2622 - accuracy: 0.4690 - val_loss: 1.8962 - val_accuracy: 0.500
0
Epoch 4/10
17/17 [==============================] - 0s 9ms/step - loss: 1.5302 - accuracy: 0.6512 - val_loss: 1.3574 - val_accuracy: 0.639
5
Epoch 5/10
17/17 [==============================] - 0s 9ms/step - loss: 1.0203 - accuracy: 0.7829 - val_loss: 1.1338 - val_accuracy: 0.662
8
Epoch 6/10
17/17 [==============================] - 0s 9ms/step - loss: 0.6625 - accuracy: 0.8682 - val_loss: 0.8953 - val_accuracy: 0.779
1
Epoch 7/10
17/17 [==============================] - 0s 9ms/step - loss: 0.4431 - accuracy: 0.9225 - val_loss: 0.8312 - val_accuracy: 0.767
4
Epoch 8/10
17/17 [==============================] - 0s 9ms/step - loss: 1.1606 - accuracy: 0.7248 - val_loss: 1.6223 - val_accuracy: 0.639
5
Epoch 9/10
17/17 [==============================] - 0s 9ms/step - loss: 0.6703 - accuracy: 0.8566 - val_loss: 0.5834 - val_accuracy: 0.872
1
Epoch 10/10
17/17 [==============================] - 0s 8ms/step - loss: 0.2990 - accuracy: 0.9729 - val_loss: 0.3710 - val_accuracy: 0.907
0

In [42]: model.evaluate(test_x, test_y)

         3/3 [==============================] - 0s 52ms/step - loss: 0.5814 - accuracy: 0.8721

Out[42]: [0.5814385414123535, 0.8720930218696594]
```

The two values reported are 0.58, and 0.87. The first is the total value after passing in all the testing images and labels through the categorical cross-entropy loss function. This numerically describes how “wrong” the networks prediction is. The second is the accuracy rate; this describes the rate at which the network is getting predictions right – in this case, it predicted the class of 87% of the testing traffic signs correctly.

i.



The number of training images was changed by changing the index break condition:

```
for row in csv_file
    if i < 80:
        img_path =
```

The general trend is by adding more training images, accuracy does improve, but plateaus, and sometimes performance may drop. This is because of the stochastic nature of the shuffling, so it may get "bad" training data on occasion. Moreover, it may overfit to the data.

ii.

Dropout is the process of setting the weights, and hence the output, of nodes of a layer to 0. When this is done randomly to random amounts of a layer, it has the overall effect of preventing the network from overfitting to data – this is because no one or set of nodes will take responsibility for fitting the data strictly. In Keras, Dropout(0.2) when coded before a layer will drop out the nodes of that layer at a rate of 0.2, meaning each training iteration will set 20% of the nodes at random to be 0.

Dropouts are added to the network as such:

```
In [108]: epochs = 10
batch_size = 16

input_shape = Input(shape=(32, 32,3))

model = Sequential([
    Conv2D(16, (3, 3), activation='relu', input_shape=(64,64,3), padding='same'),
    Conv2D(16, (3, 3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.2),

    Conv2D(32, (3, 3), activation='relu', padding='same'),

    Conv2D(32, (3, 3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.2),

    Conv2D(64, (3, 3), activation='relu', padding='same'),

    Conv2D(64, (3, 3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.2),

    Flatten(),
    Dense(units=2048, activation='relu'),
    Dropout(0.2),
    Dense(units=1024, activation='relu'),
    Dropout(0.2),
    Dense(units=128, activation='relu'),
    Dropout(0.2),
    Dense(units=43, input_dim=2048, activation='softmax'),
])

model.compile(loss='categorical_crossentropy', optimizer=Adam(lr=1e-4), metrics=['accuracy'])
trained_model_conv = model.fit(train_x.reshape(-1,64,64,3), train1_y, epochs=epochs, batch_size=batch_size, validation_data=(val
```

The evaluation yields a **loss of 0.526, and an accuracy of 89.7%.** This is worse performance than without dropout before. However, it was observed that the accuracy during training did not climb as fast – this is explained by the network not trying to overfit to the training data. Perhaps accuracy can be improved if the number of epochs is increased:

```
0.9837
Epoch 47/50
162/162 [==============================] - 1s 9ms/step - loss: 0.0348 - accuracy: 0.9903 - val_loss: 0.0929 - val_accuracy:
0.9814
Epoch 48/50
162/162 [==============================] - 1s 9ms/step - loss: 0.0320 - accuracy: 0.9919 - val_loss: 0.1515 - val_accuracy:
0.9686
Epoch 49/50
162/162 [==============================] - 1s 9ms/step - loss: 0.0486 - accuracy: 0.9907 - val_loss: 0.0937 - val_accuracy:
0.9849
Epoch 50/50
162/162 [==============================] - 1s 9ms/step - loss: 0.0278 - accuracy: 0.9915 - val_loss: 0.1128 - val_accuracy:
0.9756

In [110]: model.evaluate(test_x, test_y)

          27/27 [==============================] - 0s 13ms/step - loss: 0.1025 - accuracy: 0.9837

Out[110]: [0.10246266424655914, 0.9837209582328796]
```

After training for 50 epochs, the accuracy shot up to **98%,** confirming my hypothesis.

iii.

Batch normalization is added to the network:

In [116]:
```
epochs = 10
batch_size = 16

input_shape = Input(shape=(32, 32,3))

model = Sequential([
    Conv2D(16, (3, 3), activation='relu', input_shape=(64,64,3), padding='same'),
    BatchNormalization(),
    Conv2D(16, (3, 3), activation='relu', padding='same'),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.2),

    Conv2D(32, (3, 3), activation='relu', padding='same'),
    BatchNormalization(),

    Conv2D(32, (3, 3), activation='relu', padding='same'),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.2),

    Conv2D(64, (3, 3), activation='relu', padding='same'),
    BatchNormalization(),

    Conv2D(64, (3, 3), activation='relu', padding='same'),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.2),

    Flatten(),
    Dense(units=2048, activation='relu'),
    Dropout(0.2),
    Dense(units=1024, activation='relu'),
    Dropout(0.2),
    Dense(units=128, activation='relu'),
    Dropout(0.2),
    Dense(units=43, input_dim=2048, activation='softmax'),
])

model.compile(loss='categorical_crossentropy', optimizer=Adam(lr=1e-4), metrics=['accuracy'])
trained_model_conv = model.fit(train_x.reshape(-1,64,64,3), train1_y, epochs=epochs, batch_size=batch_size, validation_data=(val_
```

Reverting to 10 epochs, we get **a loss of 0.116, and accuracy of 96.2%,** a major improvement.

Batch normalization effectively transforms the data from the output of a layer so that it matches a certain distribution. By default, it is a mean of 0, and a variance of 1. It also has weights that can be learned, and these weights are parameters to change the output distribution. By making the output of each convolution layer more like each other in terms of distribution, the network does not have to put in much effort in the form of training epochs to try to learn weights to fit the differences in distribution.