Summer of Science 2022
Computer vision

Name: Ankith R
Roll Number: 200070006
Mentor: Rwitaban Goswami

June 19, 2022

Contents

1 Introduction

Computer vision is a field of artificial intelligence (AI) that enables computers and systems to derive meaningful information from digital images, videos and other visual inputs and then take actions or make recommendations based on that information

Computer vision tasks include methods for acquiring, processing, analyzing and understanding digital images, and extraction of high-dimensional data from the real world in order to produce numerical or symbolic information, e.g. in the forms of decisions

1.1 How does computer vision works?

Computer vision needs lots of data. It runs analyses of data over and over until it discerns distinctions and ultimately recognize images. For example, to train a computer to recognize automobile tires, it needs to be fed vast quantities of tire images and tire-related items to learn the differences and recognize a tire, especially one with no defects

A Convolutional neural network helps the model to look by breaking images down into pixels that are given tags or labels. It uses the labels to perform convolutions and make predictions about what it is seeing.The neural network runs convolutions and checks the accuracy of its predictions in a series of iterations until the predictions start to come true

Much like a human making out an image at a distance, a CNN first discerns hard edges and simple shapes, then fills in information as it runs iterations of its predictions. A CNN is used to understand single images. A recurrent neural network (RNN) is used in a similar way for video applications to help computers understand how pictures in a series of frames are related to one another. CNN considers only the current input while RNN considers current inputs as well as previous inputs. It can memorize previous inputs due to its internal memory

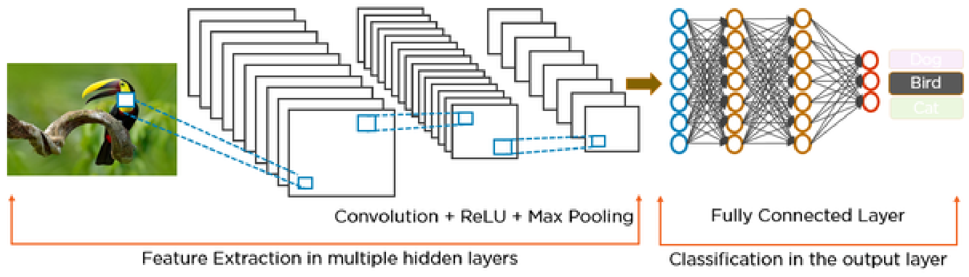
Dog
Bird
Cat
Convolution + ReLU + Max Pooling
Fully Connected Layer
Feature Extraction in multiple hidden layers
Classification in the output layer

Figure 1: Example of Convolutional Neural Network

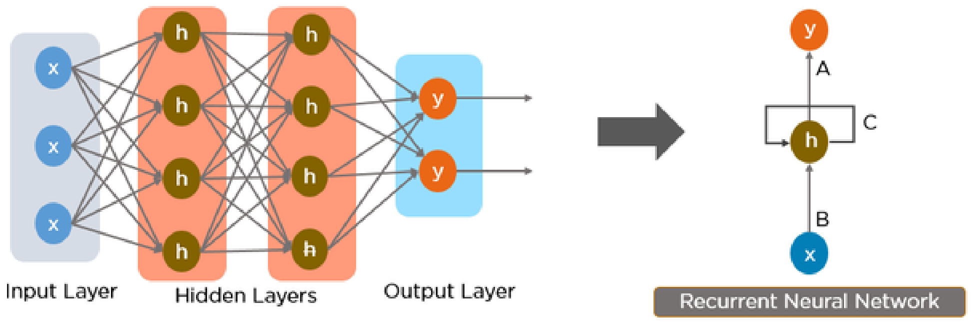

Figure 2: Example of Recurrent Neural Network

1.2 Applications of Computer Vision

- Automatic inspection in manufacturing applications
- Navigation in an autonomus car or mobile robots
- Modeling objects or enviornments, e.g in medical image analysis and topogaphical modelling
- Detecting events in visual surveillance or people counting
- Organizing information for indexing databases of images and image sequences and many more

2 Basic concepts in Computer Vision

2.1 Pinhole Camera Model

The pinhole camera model describes the mathematical relationship between the coordinates of a point in three-dimensional space and its projection onto the image plane of an ideal pinhole camera, where the camera aperture is described as a point and no lenses are used to focus light

The pin-hole camera model (or sometimes projective camera model) is a widely used camera model in computer vision. It is simple and accurate enough for most applications. In the pin-hole camera model, light passes through a single point, the camera center, C, before it is projected onto an image plane. The image formed will be the inverted as compared to original

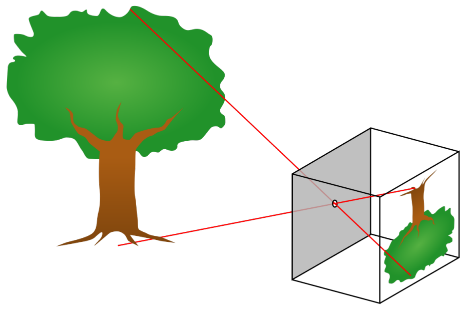

Figure 3: Pinhole Camera Model

With a pin-hole camera, a 3D point X is projected to an image point x (both expressed in homogeneous coordinates) as

$$\lambda x = PX$$

Here the 3x4 matrix P is called the camera matrix (or projection matrix). Note that the 3D point $\mathbf{X}$ has four elements in homogeneous coordinates, $\mathbf{X} = [X, Y, Z, W]$. The scalar λ is the inverse depth of the 3D point and is needed if we want all coordinates to be homogeneous with the last value normalized to one

2.1.1 Camera Matrix

The camera matrix can be decomposed as

$$P = [R|t],$$

where R is a rotation matrix describing the orientation of the camera, t a 3D translation vector describing the position of the camera center, and the intrinsic calibration matrix K describing the projection properties of the camera.

The calibration matrix depends only on the camera properties and is in a general form written as

$$\begin{bmatrix} \alpha f & s & \mathrm{C}_x \\ 0 & f & \mathrm{C}_y \\ 0 & 0 & 1 \end{bmatrix}$$

The focal length, f, is the distance between the image plane and the camera center. The skew, s, is only used if the pixel array in the sensor is skewed and can in most cases safely be set to zero.The aspect ratio, α is used for non-square pixel elements. It is often safe to assume $\alpha = 1$. With this assumption the matrix becomes

$$\begin{bmatrix} f & 0 & \mathrm{C}_x \\ 0 & f & \mathrm{C}_y \\ 0 & 0 & 1 \end{bmatrix}$$

Besides the focal length, the only remaining parameters are the coordinates of the optical center (sometimes called the principal point), the image point $c = [c_x, c_y]$ where the optical axis intersects the image plane. Since this is usually in the center of the image and image coordinates are measured from the top left corner, these values are often well approximated with half the width and height of the image

2.1.2 Computing the camera center

Given a camera projection matrix, P, it is useful to be able to compute the camera's position in space. The camera center, $\mathbf{C}$ is a 3D point with the property P$\mathbf{C} = 0$. For a camera with $P = [R|t]$, this gives

$$K[R|t]\mathbf{C} = KR\mathbf{C} + kt = 0, \tag{1}$$

and the camera center can be computed as,

$$\mathbf{C} = -R^T t \tag{2}$$

Note that the camera center is independent of the intrinsic calibration K, as expected.

2.2 Camera Calibaration

Calibrating a camera means determining the internal camera parameters, in our case the matrix K. It is possible to extend this camera model to include radial distortion and other artifacts if your application needs precise measurements. For most applications however, the simple model in equation (in last page) is good enough. The standard way to calibrate cameras is to take lots of pictures of a flat checkerboard pattern

2.2.1 A simple calibaration method

Here we will look at a simple calibration method. Since most of the parameters can be set using basic assumptions (square straight pixels, optical center at the center of the image) the tricky part is getting the focal length right.

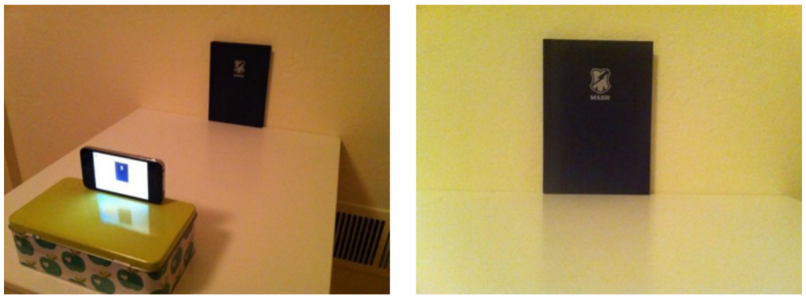

Figure 4: Setup (left) and Calibaration object (right)

Here's what to do:

- Measure the sides of your rectangular calibration object. Let's call these dX and dY
- Place the camera and the calibration object on a flat surface so that the camera back and calibration object are parallel and the object is roughly in the center of the camera's view
- Measure the distance from the camera to the calibration object. Let's call this dZ
- Take a picture and check that the setup is straight, meaning that the sides of the calibration object align with the rows and columns of the image.
- Measure the width and height of the object in pixels. Let's call these dx and dy

Now using the similar triangle properties the following relation gives the focal lengths,

$$f_x = \frac{dx}{dX} dZ, f_y = \frac{dy}{dY} dZ \tag{3}$$

2.3 Homographies

A homography is a 2D projective transformation that maps points in one plane to an other. In our case the planes are images or planar surfaces in 3D. Homographies have many practical uses such as registering images, rectifying images, texture warping and creating panoramas. In essence a homography H maps 2D points (in homogeneous coordinates) according to

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & h_9 \end{bmatrix} \begin{bmatrix} x \\ y \\ w \end{bmatrix} \ or\ \mathbf{x'} = H\mathbf{x} \tag{4}$$

Homogeneous coordinates is a useful representation for points in image planes. Points in homogeneous coordinates are only defined up to scale so that $\mathbf{x}$ = [x,y,w] = [αx,αy,αw] = [x/w,y/w,1] all refer to the same point

As a consequence, the homography H is also only defined up to scale and has eight independent degrees of freedom. Often points are normalized with w = 1 to have a unique identification of the image coordinates x,y. The extra coordinate makes is easy to represent transformations with a single matrix

When working with points and transformations we will store the points column-wise so that a set of n points in 2 dimensions will be a 3n array in homogeneous coordinates. This format makes matrix multiplications and point transforms easier.

There are some important special cases of these projective transformations. An affine transformation

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_1 & a_2 & t_x \\ a_3 & a_4 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \textit{ or } \mathbf{x'} = \begin{bmatrix} A & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix} \mathbf{x} \tag{5}$$

preserves w = 1 and can not represent as strong deformations as a full projective transformation. The affine transformation contains an invertible matrix A and a translation vector $\mathbf{t} = [t_x, t_y]$. Affine transformations are used for example in warping.

A similarity transformation

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s\,cos(\theta) & -s\,sin(\theta) & t_x \\ s\,sin(\theta) & s\,cos(\theta) & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \textit{ or } \mathbf{x'} = \begin{bmatrix} sR & \mathbf{t} \\ \mathbf{0} & 1 \end{bmatrix} \mathbf{x} \tag{6}$$

is a rigid 2D transformation that also includes scale changes. The scalar s specifies scaling, R is a rotation of an angle θ and $\mathbf{t} = [t_x, t_y]$ is again is a translation. With s = 1 distances are preserved and it is then a rigid transformation. Similarity transformations are used for example in image registration

2.3.1 The Direct Linear Transformation Algorithm

Homographies can be computed directly from corresponding points in two images (or planes). As mentioned earlier, a full projective transformation has eight degrees of freedom. Each point correspondence gives two equations, one each for the x and y coordinates, and therefore four point correspondences are needed to compute H
The direct linear transformation (DLT) is an algorithm for computing H given four or more correspondences. By rewriting the equation for mapping points using H for several correspondences we get an equation like

$$\begin{bmatrix} -x_1 & -y_1 & -1 & 0 & 0 & 0 & x_1x_1' & {y_1}'_{x}1 & x_1' \\ 0 & 0 & 0 & -x_1 & -y_1 & -1 & x_1y_1' & y_1y_1' & y_1' \\ -x_2 & -y_2 & -1 & 0 & 0 & 0 & x_2x_2' & y_2x_2' & x_2' \\ 0 & 0 & 0 & -x_2 & -y_2 & -1 & x_2y_2' & y_2y_2' & y_2' \\ & . & & . & & . & & . & \\ & . & & . & & . & & . & \\ & . & & . & & . & & . & \\ & . & & . & & . & & . & \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \\ h_3 \\ h_4 \\ h_5 \\ h_6 \\ h_7 \\ h_8 \\ h_9 \end{bmatrix} = \mathbf{0} \qquad (7)$$

or A$\mathbf{h} = 0$ where A is a matrix with twice as many rows as correspondences. By stacking all corresponding points a least squares solution for H can be found using singular value decomposition (SVD)

The points are conditioned by normalizing so that they have zero mean and unit standard deviation. This is very important for numerical reasons since the stability of the algorithm is dependent of the coordinate representation. Then the matrix A is created using the point correspondences. The least squares solution is found as the last row of the matrix V of the SVD. The row is reshaped to create H. This matrix is then de-conditioned and normalized before returned

An affine transformation has six degrees of freedom and therefore three point correspondences are needed to estimate H. Affine transforms can be estimated using the DLT algorithm above by setting the last two elements equal to zero, h7 = h8 = 0.

2.4 Image Warping

Applying an affine transformation matrix H on image patches is called warping (or affine warping) and is frequently used in computer graphics but also in several computer vision algorithms. A warp can easily be performed with SciPy using the ndimage package. The command

```
transformed_im = ndimage.affine_transform(im,A,b,size)
```

transforms the image patch im with A a linear transformation and **b** a translation vector as above. The optional argument size can be used to specify the size of the output image. The default is an image with the same size as the original

3 OpenCV

OpenCV is a C++ library for real time computer vision initially developed by Intel, now maintained by Willow Garage. OpenCV is a C++ library with modules that cover many areas of computer vision.

Besides C++ (and C) there is growing support for Python as a simpler scripting language through a Python interface on top of the C++ code base. OpenCV comes with functions for reading and writing images as well as matrix operations and math libraries

The current OpenCV version (2.3.1) actually comes with two Python interfaces. The old cv module uses internal OpenCV datatypes and can be a little tricky to use from NumPy. The new cv2 module uses NumPy arrays and is much more intuitive to use

3.1 OpenCV Basics

3.1.1 Reading and writing images

This short example will load an image, print the size and convert and save the image in .png format

```
import cv2
# read image
im = cv2.imread('empire.jpg')
h,w = im.shape[:2]
print h,w
# save image
cv2.imwrite('result.png',im)
```

3.1.2 Color spaces

In OpenCV images are not stored using the conventional RGB color channels, they are stored in BGR order (the reverse order). When reading an image the default is BGR, however there are several conversions available. Color space conversion are done using the function cvtColor()

```
im = cv2.imread('empire.jpg')
# create a grayscale version
gray = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)
```

After the source image there is an OpenCV color conversion code. Some of the most useful conversion codes are

- cv2.COLOR_BGR2GRAY
- cv2.COLOR_BGR2RGB
- cv2.COLOR_GRAY2BGR

In each of these, the number of color channels for resulting images will match the conversion code (single channel for gray and three channels for RGB and BGR). The last version converts grayscale images to BGR and is useful if you want to plot or overlay colored objects on the images

3.1.3 Displaying images and results

This code reads an image from the file and creates an integral image representation

```
import cv2
# read image
im = cv2.imread('fisherman.jpg')
gray = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)
# compute integral image
intim = cv2.integral(gray)
# normalize and save
intim = (255.0*intim) / intim.max()
cv2.imwrite('result.jpg',intim)
```

3.2 Processing Video

Video with pure Python is hard. There is speed, codecs, cameras, operating systems and file formats to consider. There is currently no video library for Python. OpenCV with its Python interface is the only good option

3.2.1 Video input

Reading video from a camera is very well supported in OpenCV. A basic complete example that captures frames and shows them in an OpenCV window looks like this

```
import cv2
# setup video capture
cap = cv2.VideoCapture(0)
while True:
ret,im = cap.read()
cv2.imshow('video test',im)
key = cv2.waitKey(10)
if key == 27:break
if key == ord(' '):
cv2.imwrite('vid_result.jpg',im)
```

The capture object VideoCapture captures video from cameras or files. Here we pass an integer at initialization. This is the id of the video device, with a single camera connected this is 0. The method read() decodes and returns the next video frame. The first value is a success flag and the second the actual image array. The waitKey() function waits for a key to be pressed and quit the application if the 'esc' key (Ascii number 27) is pressed or saves the frame if the 'space' key is pressed

3.2.2 Reading video to numpy arrays

Using OpenCV it is possible to read video frames from a file and convert them to NumPy arrays. Here is an example of capturing video from a camera and storing the frames in a NumPy array

```
import cv2
# setup video capture
cap = cv2.VideoCapture(0)
frames = [ ]
# get frame, store in array
while True:
ret,im = cap.read()
cv2.imshow('video',im)
frames.append(im)
if cv2.waitKey(10) == 27:
break
frames = array(frames)
# check the sizes
print im.shape
print frames.shape
```

Each frame array is added to the end of a list until the capturing is stopped. The resulting array will have size (number of frames,height,width,3). Arrays with video data like this are useful for video processing such as computing frame differences and tracking

4 Harries corner detection

The Harris corner detection algorithm (or sometimes the Harris Stephens corner detector) is one of the simplest corner indicators available. The general idea is to locate interest points where the surrounding neighborhood shows edges in more than one direction, these are then image corners.

We define a matrix $M_I = M_I(x)$, on the points $\mathbf{x}$ in the image domain, as the positive semi-definite, symmetric matrix

$$M_I = \Delta I \Delta I^T = \begin{bmatrix} I_x \\ I_y \end{bmatrix} \begin{bmatrix} I_x & I_y \end{bmatrix} = \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

where ΔI is the image gradient containing the derivatives I_x and I_y. Because of this construction the M_I has rank one with eigenvalues $\lambda_1 = |\Delta I|^2$ and λ_2=0. We now have one matrix for each pixel in the image

Let W be a weight matrix (typically a Gaussian filter G_σ), the component-wise convolution

$$\overline{M_I} = W * M_I \tag{8}$$

gives a local averaging of M_I over the neighboring pixels. The resulting matrix $\overline{M_I}$ is sometimes called a Harris matrix. The width of W determines a region of interest around x. The idea of averaging the matrix MI over a region like this is that the eigenvalues will change depending on the local image properties. If the gradients vary in the region, the second eigenvalue of $\overline{M_I}$ will no longer be zero. If the gradients are the same, the eigenvalues will be the same as for M_I

Depending on the values of ΔI in the region, there are three cases for the eigenvalues of the Harris matrix, $\overline{M_I}$:

- If λ_1 and λ_2 are both large positive values, then there is a corner at x
- If λ_1 is large and $\lambda_2 \approx 0$, then there is an edge and the averaging of M_I over the region doesn't change the eigenvalues that much
- If $\lambda_1 \approx \lambda_2 \approx 0$ then there is nothing

To distinguish the important case from the others without actually having to compute the eigenvalues, Harris and Stephens introduced an indicator function

$$det(\overline{M_I}) - \kappa\, trace(\overline{M_I})^2 \tag{9}$$

To get rid of the weighting constant κ, it is often easier to use the quotient ([10]) as an indicator

$$det(\overline{M_I})/trace(\overline{M_I})^2 \tag{10}$$

4.1 Finding corresponding points between images

The Harris corner detector gives interest points in images but does not contain an inherent way of comparing these interest points across images to find matching corners. What we need is to add a descriptor to each point and a way to compare such descriptors

An interest point descriptor is a vector assigned to an interest point that describes the image appearance around the point. The better the descriptor, the better your correspondences will be. With point correspondence or corresponding points we mean points in different images that refer to the same object or scene point.Harris corner points are usually combined with a descriptor consisting of the graylevel values in a neighboring image patch together with normalized cross correlation for comparison

In general, correlation between two (equally sized) image patches $I_1(x)$ and $I_2(x)$ is defined as

$$c\,(I_1, I_2) = \sum_x f(I_1(x), I_2(x)) \tag{11}$$

where the function f varies depending on the correlation method. The sum is taken over all positions x in the image patches. For cross correlation $f(I_1, I_2) = I_1\, I_2$ and then $c\,(I_1, I_2) = I_1.I_2$ with $\cdot$ denoting the scalar product.The larger the value of $c\,(I_1, I_2)$, the more similar the patches I_1 and I_2 are. Normalized cross correlation is a variant of cross correlation defined as

$$ncc\,(I_1, I_2) = \frac{1}{n-1} \sum_x \frac{(I_1(x) - \mu_1)}{\sigma_1} . \frac{(I_2(x) - \mu_2)}{\sigma_2} \tag{12}$$

where n: number of pixels in patch, μ: mean intensity and σ: standard deviation in patch, by this the method becomes robust to changes in image brightness

5 Canny edge detection

The Canny edge detector is an edge detection operator that uses a multi-stage algorithm to detect a wide range of edges in images. It was developed by John F. Canny in 1986. Canny also produced a computational theory of edge detection explaining why the technique works

The general criteria for edge detection include:

1. Detection of edge with low error rate, which means that the detection should accurately catch as many edges shown in the image as possible

2. The edge point detected from the operator should accurately localize on the center of the edge

3. A given edge in the image should only be marked once, and where possible, image noise should not create false edges

To satisfy these requirements Canny used the calculus of variations – a technique which finds the function which optimizes a given functional. The optimal function in Canny's detector is described by the sum of four exponential terms, but it can be approximated by the first derivative of a Gaussian

The process of Canny edge detection algorithm can be broken down to five different steps:

1. Apply Gaussian filter to smooth the image in order to remove the noise

2. Find the intensity gradients of the image

3. Apply gradient magnitude thresholding or lower bound cut-off suppression to get rid of spurious response to edge detection

4. Apply double threshold to determine potential edges

5. Track edge by hysteresis: Finalize the detection of edges by suppressing all the other edges that are weak and not connected to strong edges

5.1 Gaussian filter

Since all edge detection results are easily affected by the noise in the image, it is essential to filter out the noise to prevent false detection caused by it. To smooth the image, a Gaussian filter kernel is convolved with the image. This step will slightly smooth the image to reduce the effects of obvious noise on the edge detector. The equation for a Gaussian filter kernel of size (2k+1)×(2k+1) is given by:

$$H_{ij} = \frac{1}{2\pi\sigma^2} \exp(\frac{-(i-(k+1))^2 - (j-(k+1))^2}{2\sigma^2}); \; 1 \leq i, j \leq (2k+1)$$

It is important to understand that the selection of the size of the Gaussian kernel will affect the performance of the detector. The larger the size is, the lower the detector's sensitivity to noise. Additionally, the localization error to detect the edge will slightly increase with the increase of the Gaussian filter kernel size

5.2 Finding Intensity Gradient

An edge in an image may point in a variety of directions, so the Canny algorithm uses four filters to detect horizontal, vertical and diagonal edges in the blurred image. The edge detection operator (such as Roberts, Prewitt, or Sobel) returns a value for the first derivative in the horizontal direction (Gx) and the vertical direction (Gy). From this the edge gradient and direction can be determined:

$$G = \sqrt{\mathbf{G}_x{}^2 + \mathbf{G}_y{}^2}$$

$$\theta(x, y) = \arctan(\frac{I_y}{I_x})$$

The edge direction angle is rounded to one of four angles representing vertical, horizontal, and the two diagonals (0°, 45°, 90°, and 135°). An edge direction falling in each color region will be set to a specific angle value, for instance, in [0°, 22.5°] or [157.5°, 180°] maps to 0°

5.3 Gradient magnitude thresholding

Minimum cut-off suppression of gradient magnitudes, or lower bound thresholding, is an edge thinning technique. Lower bound cut-off suppression is applied to find the locations with the sharpest change of intensity value. The algorithm for each pixel in the gradient image is:

1. Compare the edge strength of the current pixel with the edge strength of the pixel in the positive and negative gradient directions

2. If the edge strength of the current pixel is the largest compared to the other pixels in the mask with the same direction (e.g., a pixel that is pointing in the y-direction will be compared to the pixel above and below it in the vertical axis), the value will be preserved. Otherwise, the value will be suppressed

5.4 Double thresholding

After application of non-maximum suppression, remaining edge pixels provide a more accurate representation of real edges in an image. However, some edge pixels remain that are caused by noise and color variation. To account for these spurious responses, it is essential to filter out edge pixels with a weak gradient value and preserve edge pixels with a high gradient value. This is accomplished by selecting high and low threshold values

5.5 Edge tracking by hysteresis

So far, the strong edge pixels should certainly be involved in the final edge image, as they are extracted from the true edges in the image. However, there will be some debate on the weak edge pixels, as these pixels can either be extracted from the true edge, or the noise/color variations. To achieve an accurate result, the weak edges caused by the latter reasons should be removed.
Usually a weak edge pixel caused from true edges will be connected to a strong edge pixel while noise responses are unconnected. To track the edge connection, blob analysis is applied by looking at a weak edge pixel and its 8-connected neighborhood pixels. As long as there is one strong edge pixel that is involved in the blob, that weak edge point can be identified as one that should be preserved

Reference Links

- http://programmingcomputervision.com/downloads/ProgrammingComputerVision_CCdraft.pdf
- https://www.pcmag.com/news/what-is-computer-vision
- https://en.wikipedia.org/wiki/Computer_vision
- https://www.ibm.com/in-en/topics/computer-vision
- https://medium.com/@Aj.Cheng/different-between-cnn-rnn-quote-7c224795db58
- https://en.wikipedia.org/wiki/Pinhole_camera_model
- https://en.wikipedia.org/wiki/Canny_edge_detector

Plan of Action

Week 5: Digit Recognition using MNIST dataset by building a ANN or CNN model using Pytorch library

Week 6: Building a Sudoku solver using OpenCV and deep learning, the major task is to detect the sudoku grid from the image and extract the corresponding number in blocks with high accuracy

Week 7: Understanding YOLO (you only look once) and R-CNN algorithms and doing the object detection using a pre trained YOLO model available in their official website

Week 8: Learning basics about Image stitching and panaromic stitching. Final report and completing any remaining work from previous weeks