

## ▼ Computer Vision and OpenCV

[OpenCV](#) is considered to be *de facto* standard for image processing. It contains a lot of useful algorithms, implemented in C++. You can call OpenCV from Python as well.

In this notebooks, we will give you some examples for using OpenCV. For more details, you can visit [Learn OpenCV](#) online course.

First, let's import cv2, as well as some other useful libraries:

```
from google.colab import drive
drive.mount('/content/drive')
```

Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mou



```
import cv2
import matplotlib.pyplot as plt
import numpy as np
```

```
def display_images(l,titles=None,fontsize=12):
    n=len(l)
    fig,ax = plt.subplots(1,n)
    for i,im in enumerate(l):
        ax[i].imshow(im)
        ax[i].axis('off')
        if titles is not None:
            ax[i].set_title(titles[i],fontsize=fontsize)
    fig.set_size_inches(fig.get_size_inches()*n)
    plt.tight_layout()
    plt.show()
```

## ▼ Loading Images

Images in Python can be conveniently represented by NumPy arrays. For example, grayscale image with size of 320x200 pixels would be stored in 200x320 array, and color image of the same dimension would have shape of 200x320x3 (for 3 color channels).

Let's start by loading an image:

```
%cd SD-AI/Lecture-1/
%ls
```

```
[Errno 2] No such file or directory: 'SD-AI/Lecture-1/'  
/content  
body.mp4  drive/  mickey.jpg  sample\_data/
```

```
im = cv2.imread('mickey.jpg')  
print(im)  
print(im.shape)  
plt.imshow(im)
```

```

[[[110 110 110]
  [112 112 112]
  [107 107 107]
  ...
  [ 85  85  85]
  [ 86  86  86]
  [ 87  87  87]]

[[107 107 107]
 [107 107 107]
 [105 105 105]
  ...
  [ 85  85  85]
  [ 85  85  85]
  [ 86  86  86]]

[[106 106 106]
 [104 104 104]
 [105 105 105]
  ...
  [ 84  84  84]
  _

```

As you can see, it is an image of braille text. Since we are not very interested in the actual color, we can convert it to black-and-white:

```

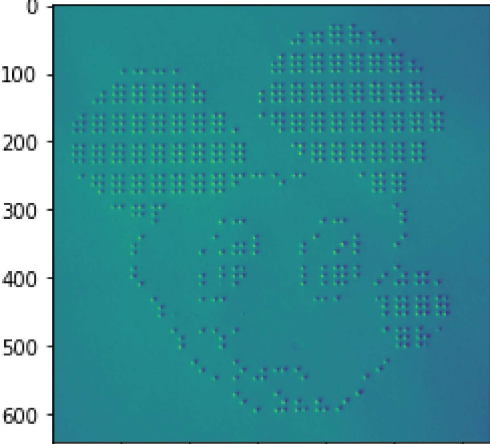
bw_im = cv2.cvtColor(im,cv2.COLOR_BGR2GRAY)
print(bw_im.shape)
plt.imshow(bw_im)

```

```

(640, 640)
<matplotlib.image.AxesImage at 0x7f5098388710>

```



```

[ 85  85  85 111

```

## ▼ Braille Image Processing

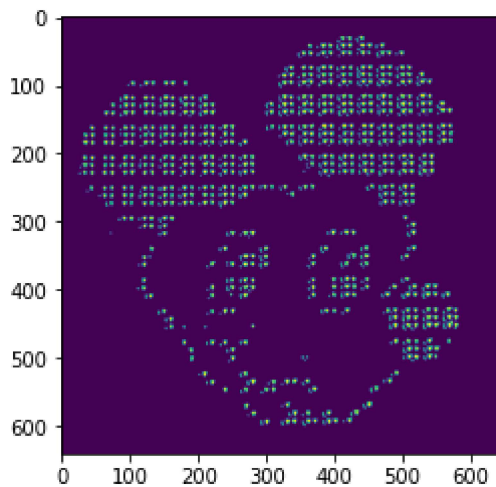
If we want to apply image classification to recognize the text, we need to cut out individual symbols to make them similar to MNIST images that we have seen before. This can be done using [object](#)

[detection](#) technique which we will discuss later, but also we can try to use pure computer vision for that. A good description of how computer vision can be used for character separation can be found [in this blog post](#) - we will only focus on some computer vision techniques here.

First, let's try to enhance the image a little bit. We can use the idea of **thresholding** (well described [in this OpenCV article](#)):

```
im = cv2.blur(bw_im,(3,3))
im = cv2.adaptiveThreshold(im, 255, cv2.ADAPTIVE_THRESH_MEAN_C,
                           cv2.THRESH_BINARY_INV, 5, 4)
im = cv2.medianBlur(im, 3)
_,im = cv2.threshold(im, 0, 255, cv2.THRESH_OTSU)
im = cv2.GaussianBlur(im, (3,3), 0)
_,im = cv2.threshold(im, 0, 255, cv2.THRESH_OTSU)
plt.imshow(im)
```

<matplotlib.image.AxesImage at 0x7f50982fc850>



To work with images, we need to "extract" individual dots, i.e. convert the images to a set of coordinates of individual dots. We can do that using **feature extraction** techniques, such as SIFT, SURF or [ORB](#):

```
orb = cv2.ORB_create(5000)
f,d = orb.detectAndCompute(im,None)
print(f"First 5 points: { [f[i].pt for i in range(5)]}")
```

First 5 points: [(469.0, 92.0), (356.0, 125.0), (359.0, 127.0), (558.0, 443.0), (568.0,



Let's plot all points to make sure we got things right:

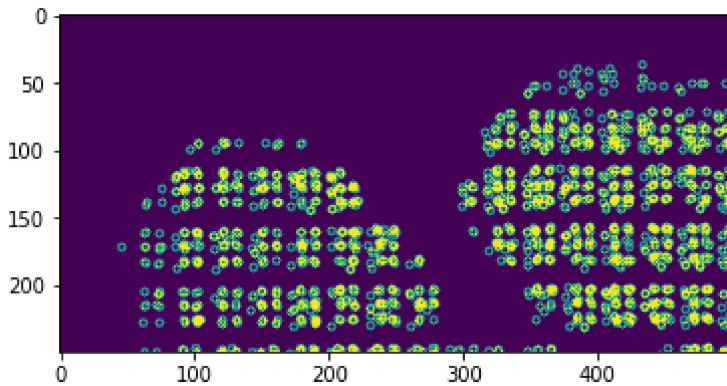
```
def plot_dots(dots):
    img = np.zeros((250,500))
```

```

for x in dots:
    cv2.circle(img,(int(x[0]),int(x[1])),3,(255,0,0))
plt.imshow(img)

pts = [x.pt for x in f]
plot_dots(pts)

```



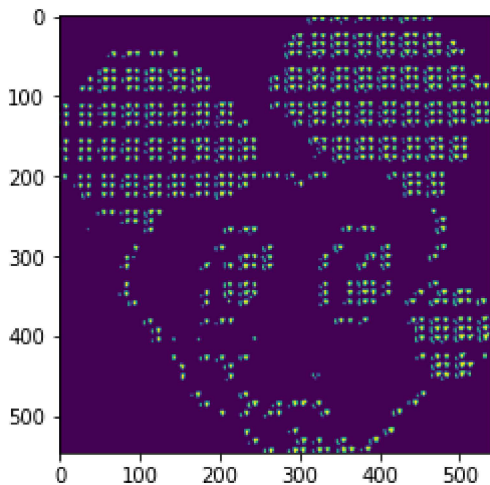
To separate individual characters, we need to know the bounding box of the whole text. To find it out, we can just compute min and max coordinates:

```

min_x, min_y, max_x, max_y = [int(f([z[i] for z in pts])) for f in (min,max) for i in (0,1)]
min_y+=13
plt.imshow(im[min_y:max_y,min_x:max_x])

```

<matplotlib.image.AxesImage at 0x7f5098339850>



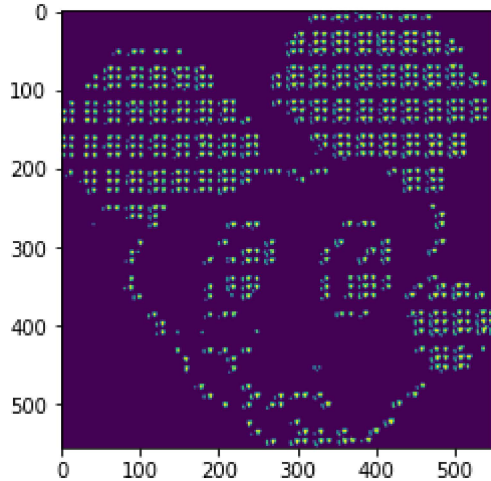
Also, this text can be partially rotated, and to make it perfectly aligned we need to do so-called **perspective transformation**. We will take a rectangle defined by points  $(x_{min}, y_{min})$ ,  $(x_{min}, y_{max})$ ,  $(x_{max}, y_{min})$ ,  $(x_{max}, y_{max})$  and align it with new image with proportional dimensions:

```

off = 5
src_pts = np.array([(min_x-off,min_y-off),(min_x-off,max_y+off),
                    (max_x+off,min_y-off),(max_x+off,max_y+off)])
w = int(max_x-min_x+off*2)
h = int(max_y-min_y+off*2)
dst_pts = np.array([(0,0),(0,h),(w,0),(w,h)])
ho,m = cv2.findHomography(src_pts,dst_pts)
trim = cv2.warpPerspective(im,ho,(w,h))
plt.imshow(trim)

```

<matplotlib.image.AxesImage at 0x7f5098a5da50>



After we get this well-aligned image, it should be relatively easy to slice it into pieces:

```

char_h = 50
char_w = 50
def slice(img):
    dy,dx = img.shape
    y = 0
    while y+char_h<dy:
        x=0
        while x+char_w<dx:
            # Skip empty lines
            if np.max(img[y:y+char_h,x:x+char_w])>0:
                yield img[y:y+char_h,x:x+char_w]
            x+=char_w
        y+=char_h

sliced = list(slice(trim))
display_images(sliced)

```



You have seen that quite a lot of tasks can be done using pure image processing, without any artificial intelligence. If we can use computer vision techniques to make the work of a neural

network simpler - we should definitely do it, because it will allow us to solve problems with smaller number of training data.

## ▼ Motion Detection using Frame Difference

Detecting motion on video stream is a very frequent task. For example, it allows us to get alerts when something happens on a surveillance camera. If we want to understand what's happening on the camera, we can then use a neural network - but it is much cheaper to use neural network when we know that something is going on.

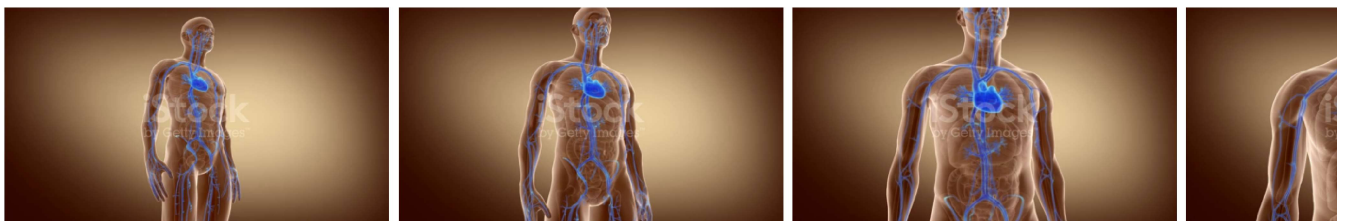
The main idea of motion detection is simple. If the camera is fixed, then frames from the camera should be pretty similar to each other. Since frames are represented as arrays, just by subtracting those arrays for two subsequent frames we will get the pixel difference, which should be low for static frames, and become higher once there is substantial motion in the image.

We will start by learning how to open a video and convert it into a sequence of frames:

```
vid = cv2.VideoCapture('body.mp4')

c = 0
frames = []
while vid.isOpened():
    ret, frame = vid.read()
    if not ret:
        break
    frames.append(frame)
    c+=1
vid.release()
print(f"Total frames: {c}")
display_images(frames[:150])
```

Total frames: 581

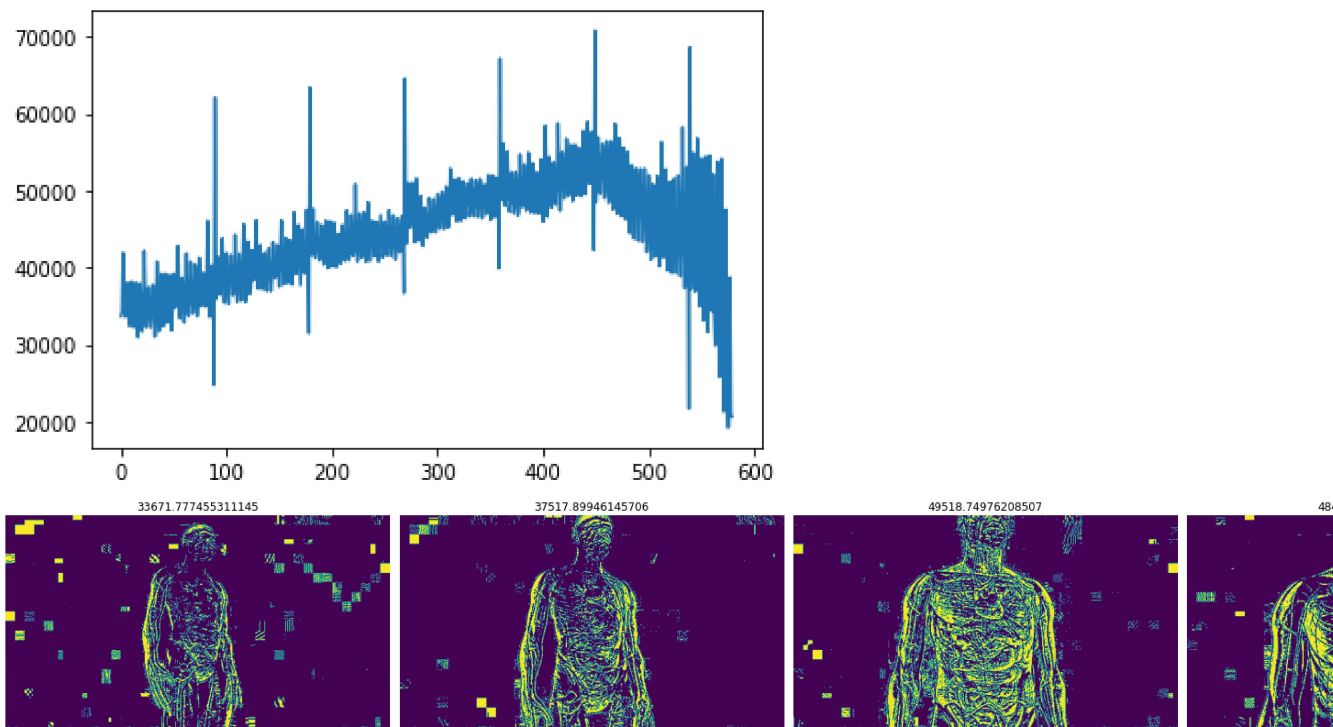


Since color is not so important for motion detection, we will convert all frames to grayscale. Then, we will compute the frame differences, and plot their norms to visually see the amount of activity going on:

```

bwframes = [cv2.cvtColor(x,cv2.COLOR_BGR2GRAY) for x in frames]
diffs = [(p2-p1) for p1,p2 in zip(bwframes[:-1],bwframes[1:])]
diff_amps = np.array([np.linalg.norm(x) for x in diffs])
plt.plot(diff_amps)
display_images(diffs[::150],titles=diff_amps[::150])

```



Suppose we want to create a report that shows what happened in front of the camera by showing the suitable image each time something happens. To do it, we probably want to find out the start and end frame of an "event", and display the middle frame. To remove some noise, we will also smooth out the curve above with moving average function:

```

def moving_average(x, w):
    return np.convolve(x, np.ones(w), 'valid') / w

```

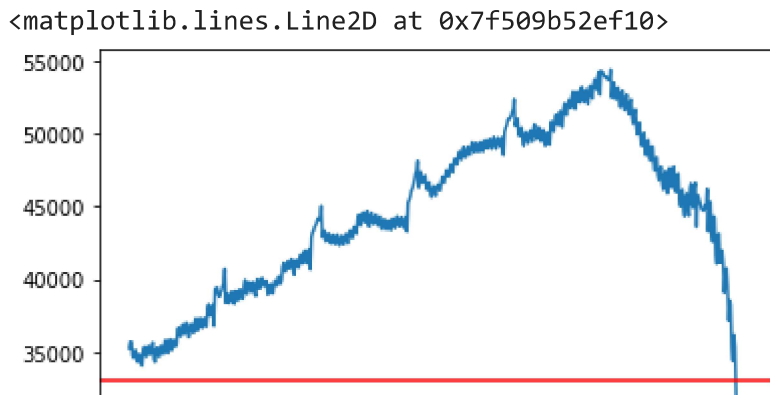
```
threshold = 33000
```

```

plt.plot(moving_average(diff_amps,10))
plt.axhline(y=threshold, color='r', linestyle='-')

```





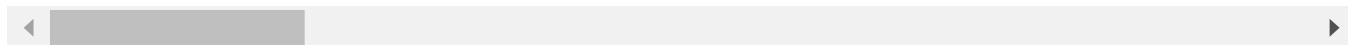
Now we can find out frames that have the amount of changes above the threshold by using `np.where`, and extract a sequence of consecutive frames that is longer than 30 frames:

```
active_frames = np.where(diff_amps>threshold)[0]
```

```
def subsequence(seq,min_length=30):
    ss = []
    for i,x in enumerate(seq[:-1]):
        ss.append(x)
        if x+1 != seq[i+1]:
            if len(ss)>min_length:
                return ss
            ss.clear()
```

```
sub = subsequence(active_frames)
print(sub)
```

```
[89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 10
```



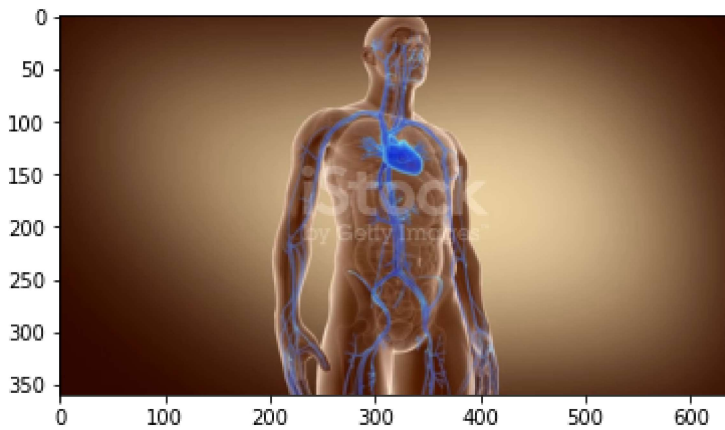
And finally we can display the image:

Rich text editor toolbar with icons for bold, italic, link, unlink, list, ordered list, indent, outdent, undo, redo, smiley, and help.



```
plt.imshow(frames[(sub[0]+sub[-1])/2])
```

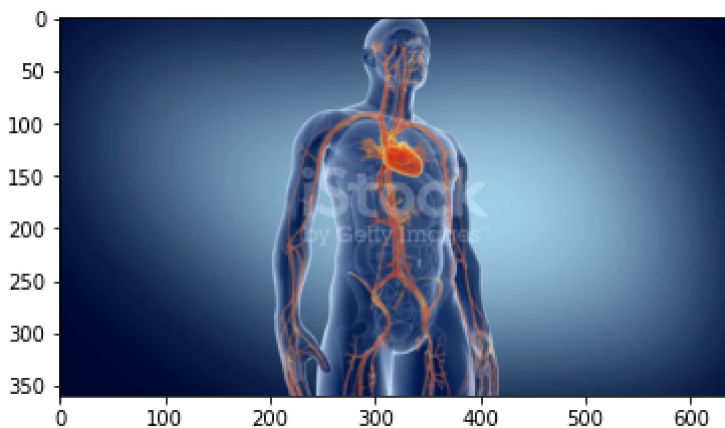
```
<matplotlib.image.AxesImage at 0x7f50998425d0>
```



You may notice that color scheme on this image does not look right! This is because OpenCV for historical reasons loads images in BGR color space, while matplotlib uses more traditional RGB color order. Most of the time, it makes sense to convert images to RGB immediately after loading them.

```
plt.imshow(cv2.cvtColor(frames[(sub[0]+sub[-1])//2], cv2.COLOR_BGR2RGB))
```

```
<matplotlib.image.AxesImage at 0x7f5099797250>
```



## ▼ Extract Motion using Optical Flow

While just comparing two consecutive frames allows us to see the amount of changes, it does not give any information on what is actually moving and where. To get that information, there is a technique called [optical flow](#):

- **Dense Optical Flow** computes the vector field that shows for each pixel where it is moving
- **Sparse Optical Flow** is based on taking some distinctive features in the image (eg. edges), and building their trajectory from frame to frame.

Read more on optical flow [in this great tutorial](#).

Let's compute dense optical flow between our frames:

```
flows = [cv2.calcOpticalFlowFarneback(f1, f2, None, 0.5, 3, 15, 3, 5, 1.2, 0)
         for f1,f2 in zip(bwframes[:-1],bwframes[1:])]
flows[0].shape

(360, 640, 2)
```

As you can see, for each frame the flow has the dimension of the frame, and 2 channels, corresponding to x and y components of optical flow vector.

Displaying optical flow in 2D is a bit challenging, but we can use one clever idea. If we convert optical flow to polar coordinates, then we will get two components for each pixel: *direction* and *intensity*. We can represent intensity by the pixel intensity, and direction by different colors. We will create an image in [HSV \(Hue-Saturation-Value\) color space](#), where hue will be defined by direction, value - by intensity, and saturation will be 255.

```
def flow_to_hsv(flow):
    hsvImg = np.zeros((flow.shape[0],flow.shape[1],3),dtype=np.uint8)
    mag, ang = cv2.cartToPolar(flow[..., 0], flow[..., 1])
    hsvImg[..., 0] = 0.5 * ang * 180 / np.pi
    hsvImg[..., 1] = 255
    hsvImg[..., 2] = cv2.normalize(mag, None, 0, 255, cv2.NORM_MINMAX)
    return cv2.cvtColor(hsvImg, cv2.COLOR_HSV2BGR)

start = sub[0]
stop = sub[-1]
print(start,stop)

frms = [flow_to_hsv(x) for x in flows[start:stop]]
display_images(frms[::25])
```

89 177



So, in those frames greenish color corresponds to moving to the left, while blue - moving to the right.

Optical flow can be a great tool to draw conclusions about general direction of movement. For example, if you see that all pixels in a frame are moving in more or less one direction - you can conclude that there is camera movement, and try to compensate for that.

Task 1: Use another image and video and run the code again (50%)

Task 2: What did you learn from the code above. Analyze your understanding in your own words. (50%)

[Colab paid products](#) - [Cancel contracts here](#)

✓ 11s completed at 8:01 PM

