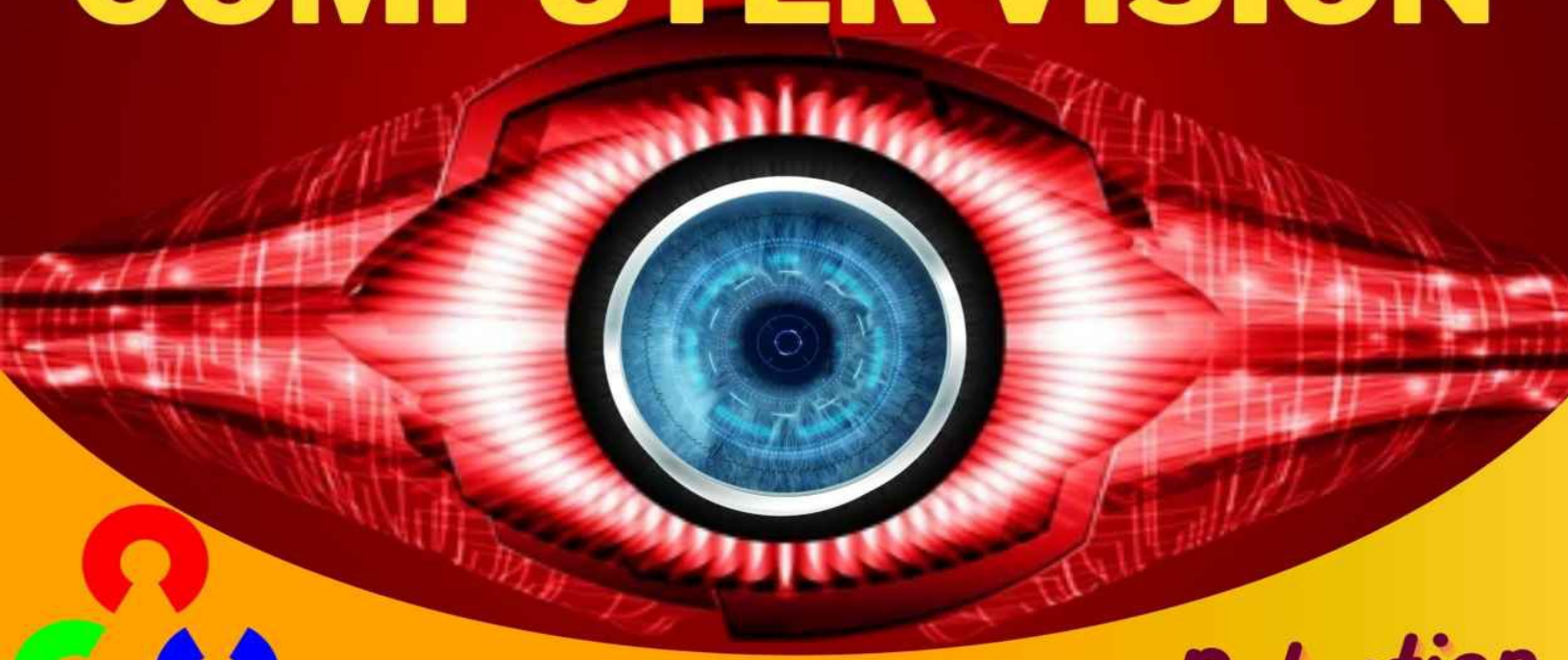
OPENCV

PYTHON

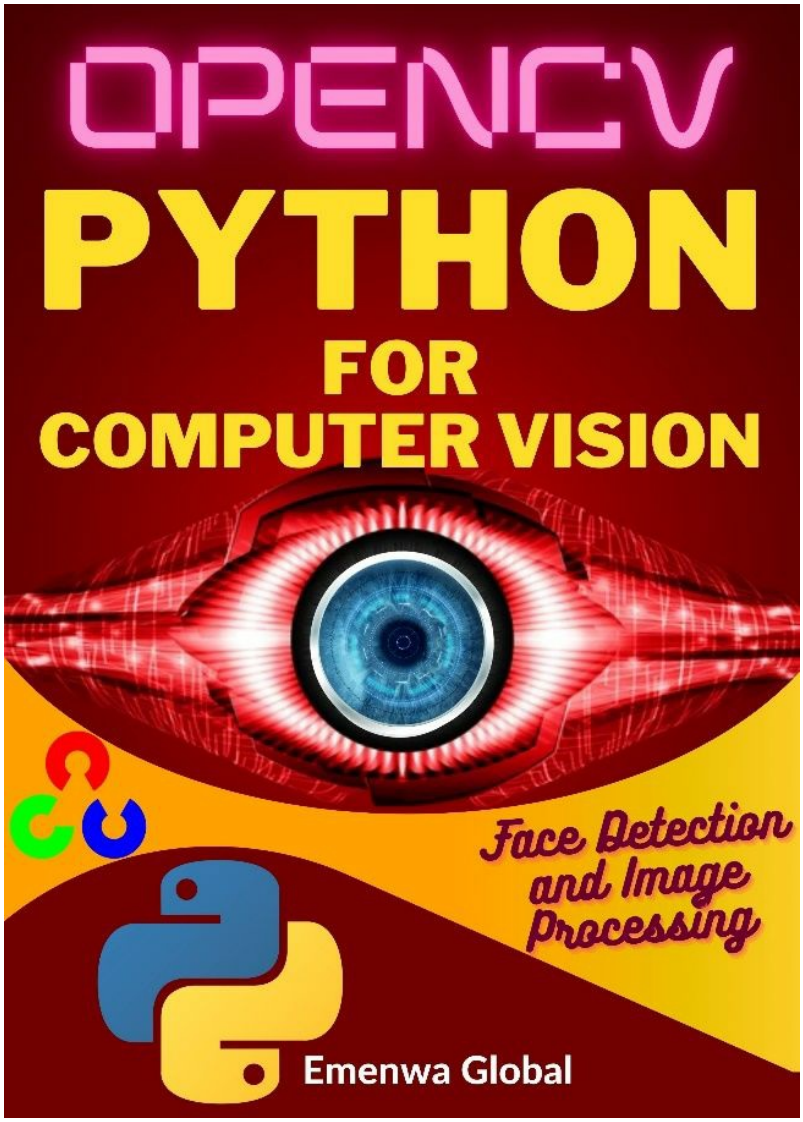
FOR COMPUTER VISION



Face Detection and Image Processing

Emenwa Global

OpenCV | Python
for Computer Vision



(Face Detection and Image Processing)

Emenwa Global

CONTENTS

INTRODUCTION

Hello, avid techy person,

I am excited to meet you. If you are reading this, you and I have one thing in common: We are fascinated by Artificial Intelligence and Machine Learning.

Computer vision is a fascinating area of computer science. Decades of research have gone into this field. Cloud computing and powerful GPUs and TPUs make image processing faster and more efficient. Cars, robots, and drones have begun to understand images and videos. The human-machine interface "computer vision" will grow in importance over the years.

Computer vision is considered the hottest field in the age of AI. It can be stressful for beginners as most people face challenges transitioning into computer vision.

Modern technology uses computer vision. In real-time, we can use OpenCV for Python. We now have more processing power to work with, thanks to powerful machines. It allows us to integrate computer vision applications into the cloud easily. Web developers do not have to reinvent the wheel.

What is OpenCV?

From what OpenCV says:

"OpenCV is released under a BSD license, which means it can be used for free in both schools and businesses. It can be used with Windows, Linux, Mac OS, iOS, and Android. It has interfaces for C++, C, Python, and Java. OpenCV was made to work well with computers and is useful for real-time applications. The library is written in optimized C/C++ to use multi-core processing. When OpenCL is turned on, it can use the hardware acceleration of the heterogeneous compute platform underneath."

OpenCV has code for more than 2500 different algorithms! It can be used for both business and school projects without cost. The library has interfaces for many other programming languages, such as Python, Java, and C++.

What can you do with OpenCV?

You can do almost any Computer Vision task you can think of with OpenCV. In real life, you have to put together a lot of blocks to get what you want. So, all you need to know to get what you wish is which modules and functions to use.

In-built Data structures and input/output

One of the best things about OpenCV is that it comes with a lot of built-in primitives for image processing and computer vision operations. If you have to start from scratch and write something, you will need to define things like an image, a point, a rectangle, and so on. Almost every computer vision algorithm needs these. All of these basic structures are already built into OpenCV. They are all in the core module. Another benefit is that these frameworks are already optimized for speed and memory, so users don't have to bother about the specifics of implementation.

The imgcodecs module is in charge of opening and saving image files. With a simple command, you can save the output image as either a jpg or a png file when you're done with it. When you work with cameras, you will have to deal with a lot of video files. There are different modules that take care of everything that has to do with putting and taking out video files. You can easily record a video from a webcam or read a video file in various formats. You can also set properties like frames per second, frame size, and so on to save a bunch of frames as a video file.

Processes for handling images

When you write a Computer Vision algorithm, you will use a lot of the same basic image processing steps over and over. The imgproc module has most of these functions. You can do things like image filtering, geometric transformations, morphological operations, drawing on images, color conversions, histograms, motion analysis, shape analysis, feature detection, and so on.

In OpenCV, we only need one line to do many of these manipulations, as you would see in this OpenCV course.

Building GUI

High-level user interface operations can be done in OpenCV. Let's say you are working on a problem and want to see what the image looks like before you move on to the next step. There are modules that can make windows that show images and/or video. There is also a function that waits until you press a key on your keyboard before moving on to the next step. There is a function that can also track what the mouse does. This is very helpful when making interactive apps. With this feature, you can draw rectangles on these input windows and then go to the next step based on the area you chose.

Video analysis

Video analysis involves figuring out how things move from one frame to the next, keeping track of different things in a video, making models for video surveillance, etc. OpenCV handles video displaying and analysis as well.

Reconstruction in 3D

Computer Vision has a lot to say about 3D reconstruction. Using the suitable algorithms, we can put together a 3D scene from a set of 2D images. OpenCV has algorithms that can figure out the 3D positions of objects by figuring out how they relate to each other in these 2D images.

Taking out the features

As we've already talked about, the human visual system tends to pick out the most essential parts of a scene so that they can be remembered later. To do the same, people started making "feature extractors" that can find these crucial parts of an image. There are many well-known algorithms that can be found and extracted with the help of a module called features2d.

Object detection

Object detection is the process of figuring out where an object is in a picture. This process doesn't care what kind of object it is. If you make a chair detector, it will tell you where the chair is in a picture. It won't tell you if the chair is red with a high back or blue with a low back. In many Computer Vision systems, figuring out where things are is an essential step. In this tutorial, we will discuss this topic and build an algorithm that can detect the faces of the characters in The Simpsons series.

This book will teach you how to build amazing computer vision applications quickly and easily. We begin by transforming images geometrically. Then we'll look at affine and projective transformations and how to apply them to photos. Applicable computer vision techniques will be covered in the following sections.

This book also includes clear Python examples for building OpenCV applications. The book begins with basic tasks like image processing, image handling, image mapping, and image detection. It also includes examples of popular OpenCV libraries.

This OpenCV guide teaches you about the different functions of OpenCV and how they are implemented.

In this course for beginners, you'll learn everything you need to know about OpenCV. You will learn everything from the basics (like reading images and videos, changing the way pictures look, and writing some text on images) to more advanced ideas (color spaces, edge detection). In the end, you will have a guide to building your own Deep Computer Vision robot that will identify and name different cartoon characters.

Dive in, techy person!

CHAPTER 1: SETTING UP OPENCV

To set up your first tool, you need Python. You are one big step ahead of many people if you already know this! I used Python 3.10 in all the tasks I did in this book. When you see this, whatever version is available is fine and will work well.

Now, I will assume that you know Python and are familiar with Python libraries' concepts. Please look for a book that teaches Python and understands it well if you aren't. To learn and use OpenCV, you must install OpenCV (obviously!), Python, and related libraries such as NumPy, OpenNI, SciPy, and SensorKinect.

There are different ways to install these tools. You can download Python here: https://www.python.org/downloads.

We will talk about installing these tools on Mac, Windows, and Ubuntu OSes.

Setting Up Windows

As we said, you will first set up your Windows computer to use Python. At least get the latest version of Python available on the website. Then, we go on with the others.

Prerequisites for installing OpenCV include Python and PIP. You must already have Python and PIP installed before OpenCV can work on your windows system. Follow these steps to see if Python is already installed on your system:

Open a command prompt window by typing cmd into the Run dialog box and pressing the Enter key. Or simply press Win + R and type in cmd.

After the command prompt window opens, run the following command:

```
python --version
```

If Python is installed, this will show in the following line:

Otherwise, go to the Python official website and install it.

How to Install Pip

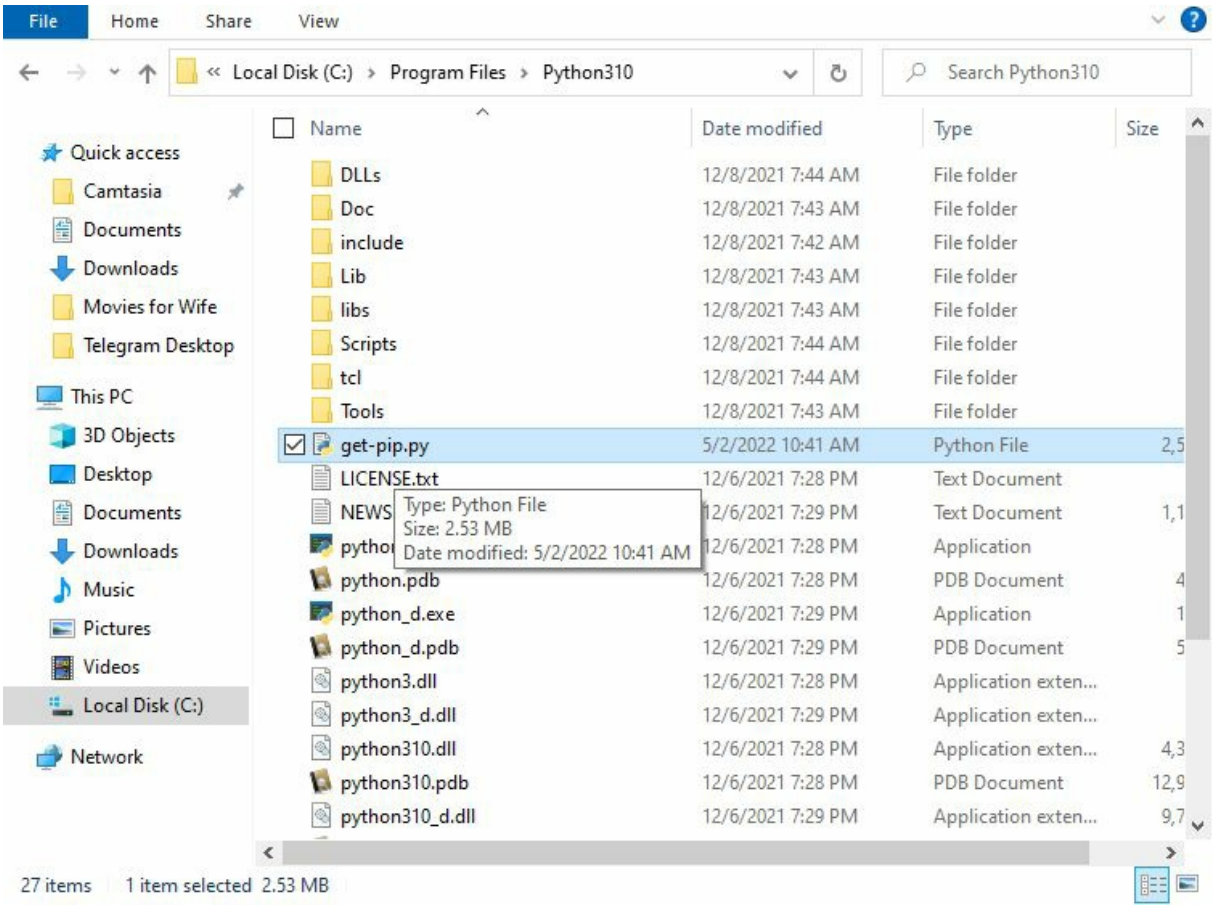
Packages written in Python can be installed and managed using PIP, a package management system. The Python Package Index is a massive "online repository" where these files are kept (PyPI).

Go to the command line and run the following command to see if PIP is already installed on your computer:

```
pip -V
```

If pip is installed, skip this next section. If not installed, follow the following steps:

Follow this link to download pip.py. After downloading the file, move it to the Python application folder.



Alternatively, you may have to use the command prompt to install pip. Run

the following command in the command prompt window:

curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py

```
C:\Users\Jide>curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
 19 2596k   19  495k    0     0   203k      0  0:00:12  0:00:02  0:00:10  203k
```

After you have installed it, type in the following line:

python get-pip.py

```
C:\Users\Jide>python get-pip.py
Defaulting to user installation because normal site-packages is not writeable
Collecting pip
  Downloading pip-22.0.4-py3-none-any.whl (2.1 MB)
     ---------------------------------------- 2.1/2.1 MB 1.1 MB/s eta 0:00:00
Collecting wheel
  Downloading wheel-0.37.1-py2.py3-none-any.whl (35 kB)
Installing collected packages: wheel, pip
```

You can type

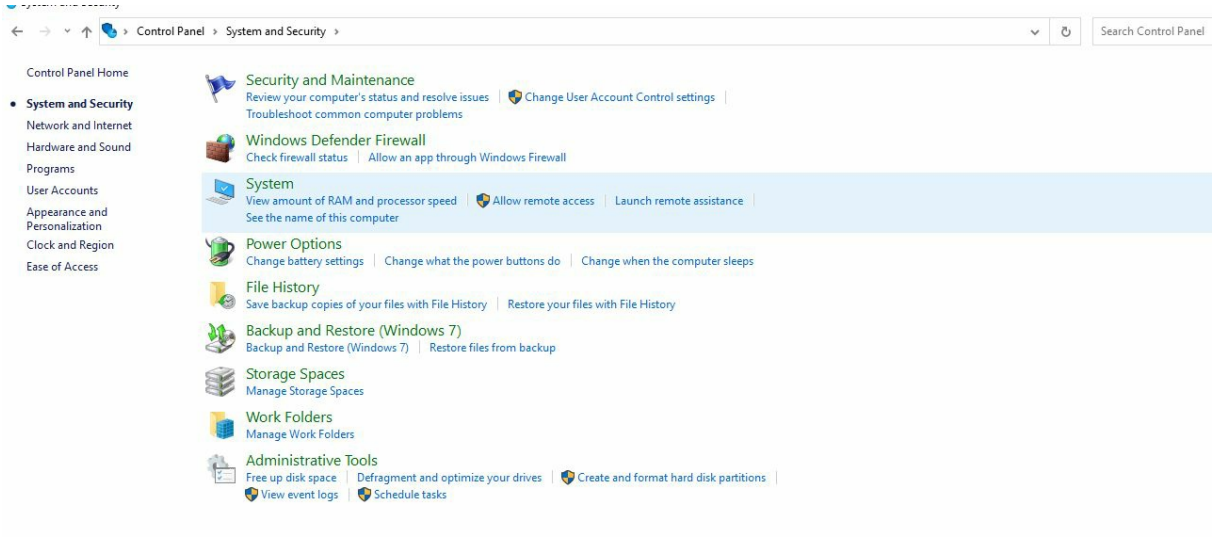
pip help

in the command line to verify your pip installation. If you get an error message, start the installation all over.

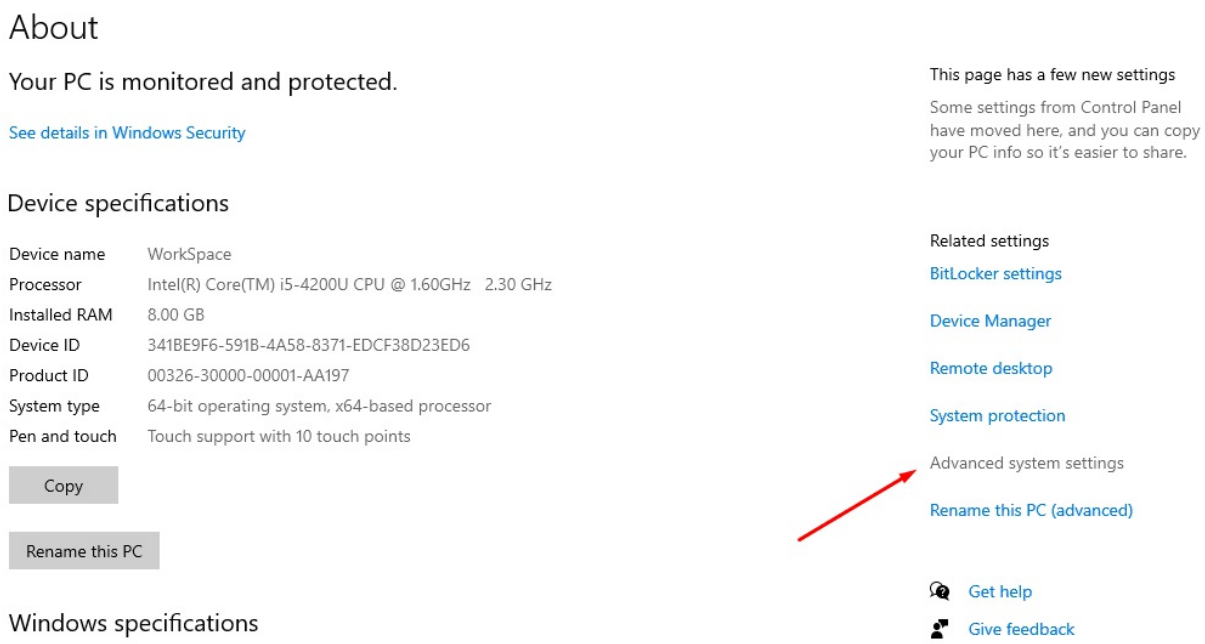
The next thing to do is to add pip to windows environment variables. To avoid the "not on PATH" error when running PIP from any location, it must be added to Windows environment variables. Do so by following the steps below:

Search for the System and Security window in the Control Panel.

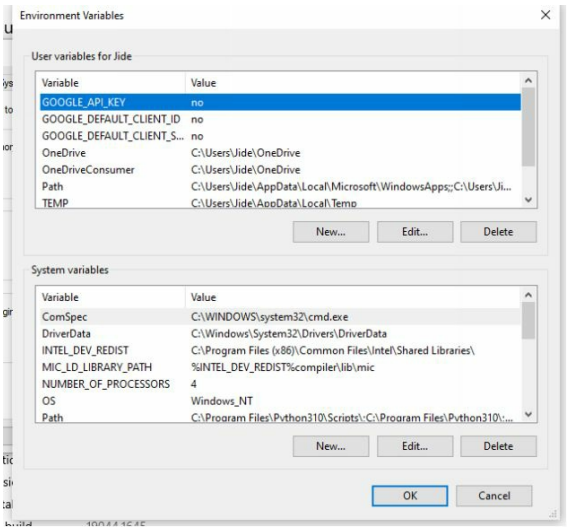
Go to the System Preferences menu.



Then, click on **Advanced system settings** by the sidebar.



Go to Environment Variables. When you've opened Environment Variables, click on the Path variable under System.



After that, click on New and specify the location of the PIP installation directory.

To save the changes, click OK.

Now that you have the pip installed, you can install the packages we need for this tutorial. Obviously, the first one is OpenCV. Run the following command:

pip install opencv-contrib-python

Sometimes, people may tell you to use another simpler line to install OpenCV. You may find something along the lines of:

opencv-python

Well, this opencv-python line is the main module of open CV. This openCV-conrib-Python line will install a package that includes everything in the main module and contribution modules the Python and OpenCV community provides. So this is why I recommend you install it, as it consists of all OpenCV functionality.

When you run that line, you wait until it installs.

```
Requirement already satisfied: numpy>=1.14.5 in c:\users\jide\appdata\roaming\python\python310\site-packages (from openc
v-contrib-python) (1.21.4)
Installing collected packages: opencv-contrib-python
Successfully installed opencv-contrib-python-4.5.5.64
```

If you notice well, once you install the OpenCV, the installation went on to install the NumPy package. Now NumPy is kind of a scientific computing package in Python that's extensively used in matrix and array manipulations, transformations, reshaping, and things like that. Now, we'll be using NumPy in some of the videos in this book. But don't worry if you've never used NumPy before. It's simple and relatively easy to get started with.

Now the next package you need to install is caer. So go ahead and type this in the command line:

```
pip install caer
```

Now, you must understand that contributors build this package to OpenCV as expert programmers love to speed up their workflow. Caer is basically a set of utility functions that will prove super useful to you in your computer vision journey. It has a ton of super useful helper functions that will help speed up your workflow.

Although we're not going to be using this for a good part of this tutorial, we'll only begin to use this in our last chapter of this course when we're building a deep computer vision model. I recommend you install it now so that you don't have to worry about the installation process later on.

Now, let us talk about installing OpenCV on Mac. For the record, I use a Windows system, and most parts of this tutorial will include screenshots from a Windows screen. However, you must understand that the differences are very trivial.

Setting Up OpenCV on Mac

Homebrew will be used to install OpenCV-Python. Mac OS X's Homebrew package manager comes in handy when setting up new libraries and applications. In order to install Homebrew, type the following command in your Terminal:

```
$ ruby -e "$(curl -fsSLhttps://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Despite the fact that OS X comes with Python preinstalled, you still need to use Homebrew to add Python to your system. Brew Python is the name given

to this version. Brew Python is the next step after installing Homebrew. Type the following command in the Terminal:

```
$ brew install python
```

As a result of this, pip will be installed as well. You can use Pip, a Python package management tool, for installing other Python packages. Let's see if the Python brew is working properly before we proceed. Type the following into your Terminal:

```
$ which python
```

/usr/local/bin/python should appear in the Terminal. This shows that we are not using the built-in system Python but rather the brewed version. We can now add the homebrew/science repository, where OpenCV is, having installed brewed Python. Run the following command in the Terminal:

```
$ brew tap homebrew/science
```

Install NumPy if it hasn't already been done so. If not, type the following command in your Terminal to install it:

```
$ pip install NumPy
```

OpenCV can now be installed. Let's get started by typing the following command in your Terminal:

```
$ brew install opencv --with-tbb --with-opengl
```

This will install OpenCV, but you can't use it just yet. Our OpenCV packages must be pointed out to Python. Let's go ahead and symlink the OpenCV files to get the job done. In your Terminal, type in the following commands and press enter:

```
$ cd /Library/Python/3.10/site-packages/
```

```
$ ln -s /usr/local/Cellar/opencv/5.5.5/lib/python2.7/site-packages/cv.py
```

```
cv.py
```

```
$ ln -s /usr/local/Cellar/opencv/4.5.5/lib/python2.7/site-packages/cv2.so
```

```
cv2.so
```

You're good to go now! Let's check to see if it's all set up. Type the following

command in the Python shell:

```
>>> import cv2
```

If you see an error message, that means the installation did not work. If you see no error, you are good to go!

Setting Up Linux

Like I said earlier, I am using a Windows computer for most of this tutorial, and everything I say about all other OSes are contributions from other experts. Now, let us walk you through the installation if you are using Linux (Ubuntu).

We must first set up a few prerequisites before we can proceed. Python and PIP must already be installed on a computer in order to use OpenCV. Make sure Python is already installed by following these steps:

Type this line in your Terminal:

```
python3.x –version
```

If Python is already installed, a message stating the version of Python that is currently available in your system will be displayed.

Using the command line, you can see if PIP is already installed on your system.

```
pip --version
```

If you don't have pip installed, Using the command line in Linux, pip can be easily installed using the following command:

```
sudo apt-get install python3-pip python-dev
```

Once pip is successfully installed, you can install OpenCV with pip. To install OpenCV, type the following command in the Terminal:

```
pip3 install opencv-python
```

To see if OpenCV is up to date, run the following commands to verify its installation:

```
python3
```

```
>>>import cv2
>>>print(cv2.__version__)
```

That's all on this first part! We have successfully set up OpenCV on your system.

In the next chapter, we'll be talking about how to read images and videos in OpenCV.

Chapter 2: Reading Images and Video

Welcome to the world. You are ready to master the weapons.

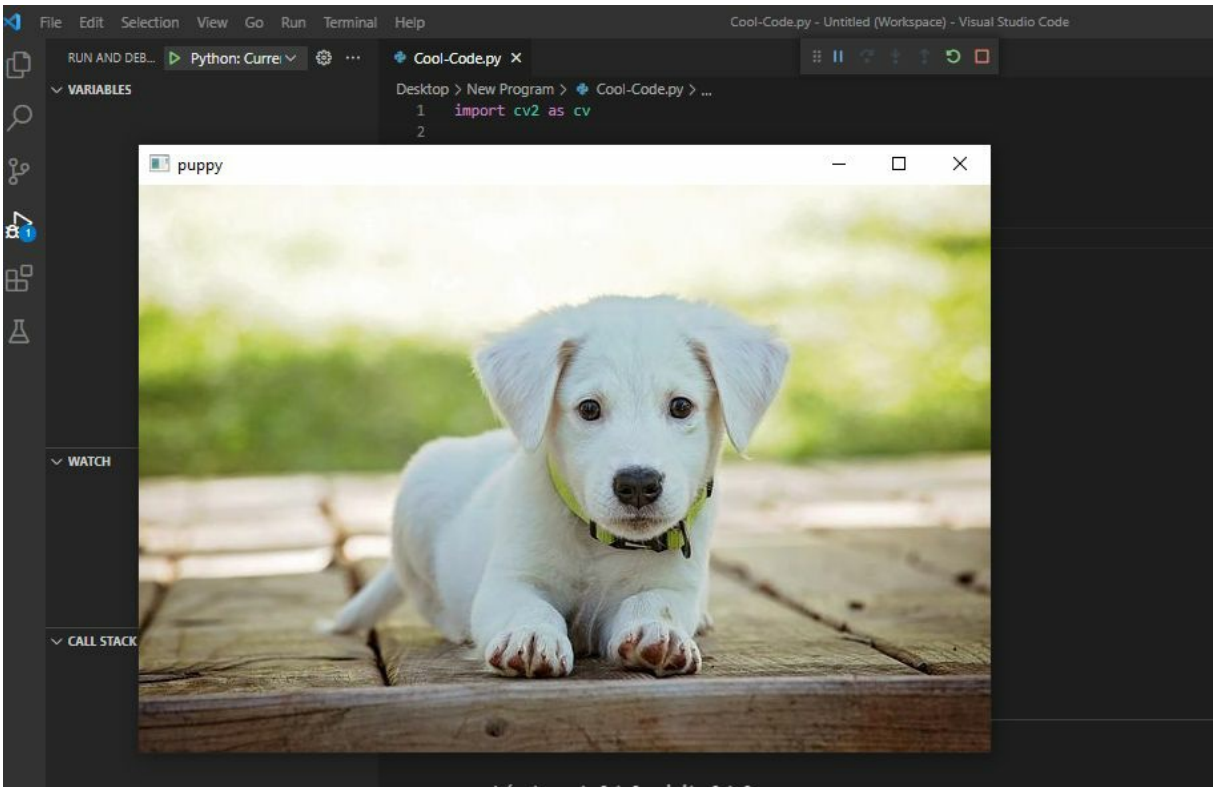
Pick any picture you have on your computer, and we will load and read it with Python OpenCV. Here is what you should do:

How do we load an image in OpenCV-Python? Open your preferred code editor (I use Visual Studio Code) and create a Python file. Name it anything you like and add .py at the end. Now create a folder on your computer and name it images or pictures. You can rename your picture into something easy to pick up. Like "fun" and move it to the new folder you just created.

Finally, add the following lines of Python code to that file:

```
import cv2 as cv


img = cv.imread('images/puppy.jpg')


cv.imshow('puppy', img)


cv.waitKey(0)
```

Save this and run the code. You will see a new window with your picture! Mine is as seen below:



This code needs to be broken down line by line. The OpenCV library is imported in the first line. Everything we'll be using in our code relies on this. The image is read and stored in a variable in the second line. A NumPy data structure is used to store the image in OpenCV.

So, let's start explaining each line. I first created a new file and called it New Program.py.

The first thing I did in the code was import cv2 as cv.

So the way we read in images in OpenCV is by making use of the cv.imread method. Now, this method basically takes in a path to an image and returns that image as a matrix of pixels.

Specifically, we're going to be trying to read this image of a puppy here. So, because I put the picture of the puppy in a folder I call images, we're going to say images/puppy.jpg. And we're going to capture this image in a variable called img.

Now you can also provide absolute paths. But since this images folder is inside my current working directory, I'm going to reference the image

relatively. Once we've read in our image, we can actually display this image by using the cv.imshow method. Now, this method basically displays the image as a new window. So the two parameters we need to pass into this method is actually the name of the window, which in this case is going to be kept, and the actual matrix of pixels to display, which in this case is IMG.

And before we actually move ahead, I added an additional line, this:

```
cv.Waitkey (0)
```

This method is basically a keyboard binding function that waits for a specific delay or time in milliseconds for a key to be pressed. So if you pass in zero, it basically waits for an infinite amount of time for a keyboard key to be pressed. I didn't worry too much about this. It's not really that important for this chapter. But we will be discussing some parts of it towards the end of this book. When I save and run the program, the image is displayed in a new window. Cool.

Again, let me explain why the last line is here. In OpenCV, the function cv.waitKey () is used to bind the keyboard. It takes a number as an argument, and that number shows how long it has been since the start of the program.

Basically, this function is used to wait for a certain amount of time until a keyboard event happens. At this point, the program stops and waits for you to press any key to keep going. If we don't give any arguments or give 0 as an argument, this function will wait forever for a keyboard event.

How OpenCV Displays Images With Colour Spaces

Most of the time, the word pixels or pixel values are used to talk about images. When it comes to color images, there are three different color channels. Because of this, single-pixel values in colored images will have more than one value. The size of these arrays can change based on the resolution and color depth. The numbers for color range from 0 to 255. Most of the time, these color channels are shown as Red, Green Blue (RGB), for example.

OpenCV makes it easy to read images. Remember that the imread function default reads images in the BGR (Blue-Green-Red) format. By adding extra flags or modules to the imread function, we can read images in different

formats:

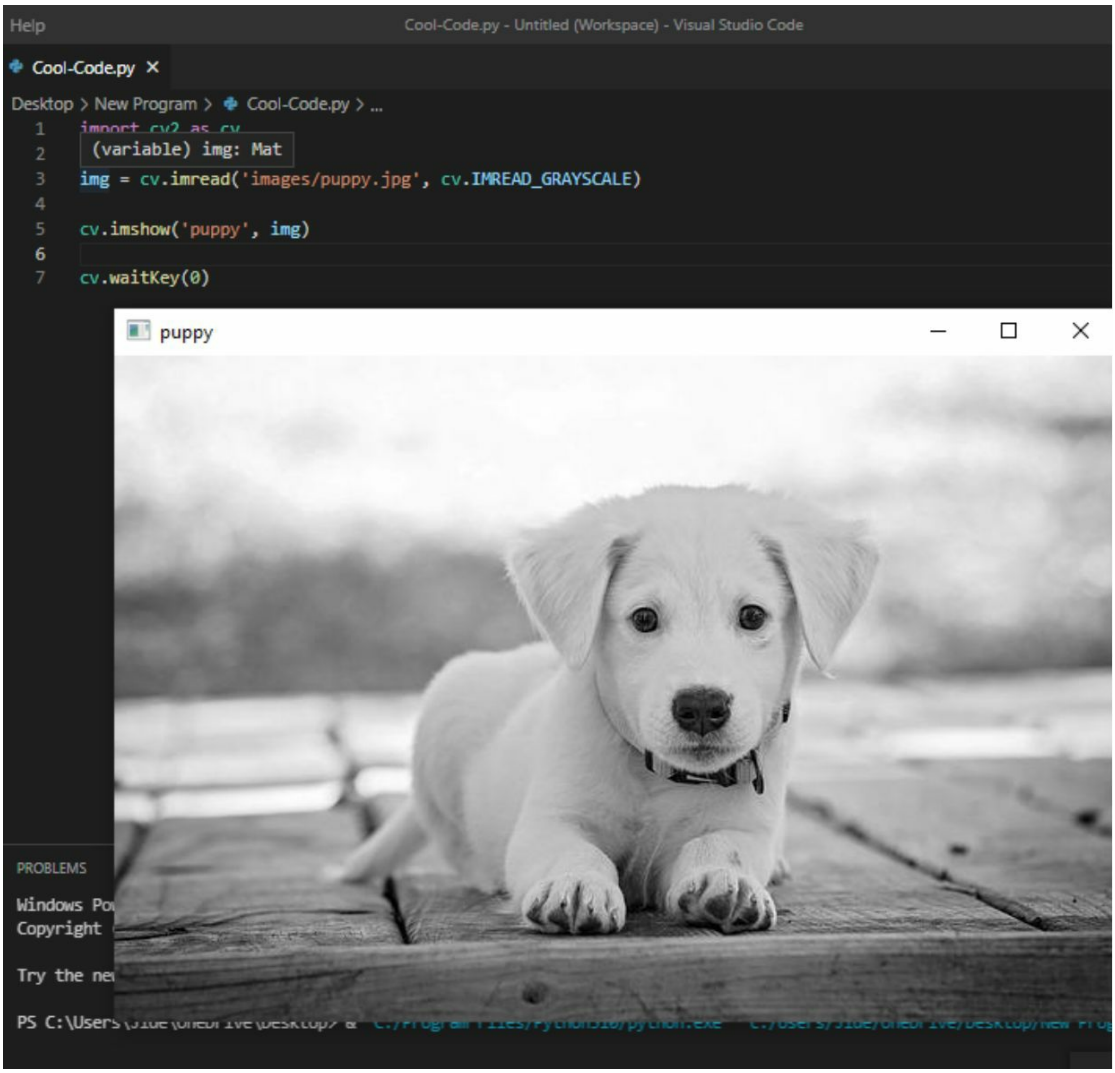
cv.IMREAD_COLOR will set the image to load in its default color.

cv.IMREAD_GRAYSCALE loads it in grayscale.

Let's try that:

```
import cv2 as cv

img = cv.imread('images/puppy.jpg', cv.IMREAD_GRAYSCALE)

cv.imshow('puppy', img)

cv.waitKey(0)
```

It will bring out the same image as grey or black and white:



Now that we have seen how to use Python OpenCV to read images, let us open a video file.

Reading Videos in OpenCV

If you have a video you want to read, do well to move it to the same folder as the image we read earlier. So I have a cute video of a dog I want to show you how I did.

So what we're going to do is we're actually going to read in this video of a dog. We read in videos by creating a capture variable and setting this equal to cv.VideoCapture. So the first line will be:

```
capture = cv.VideoCapture()
```

Now this cv.VideoCapture method either takes an integer as an argument like 0, 1, 2, 3, etc., or a path to a video file.

Now you would provide an integer argument like 0, 1, or 2 if you are using your webcam or a camera connected to your computer. That is, you are reading a video directly from your camera. In most cases, Python will reference your webcam with the integer zero. But if you have multiple cameras connected to your computer, you could reference them by using the appropriate argument. For example, 0 would reference your webcam, 1 would reference the first camera that is connected to your computer, 2 would reference the second camera, and so on. But in this chapter, we'll be actually be learning how to read an already existing video from a file path.

Now, I have the dog video saved in a folder on my Desktop as dog.mp4. That means we will provide the path.

The line will be:

```
cv.VideoCapture('video/dog.mp4')
```

```
While True:
    isTrue, frame = capture.read()
```

Now, here's where reading videos are different from reading images. In the case of reading and videos, we actually use one loop and read the video frame by frame.

That is what that While statement is saying. And the first thing we want to do inside this loop to say it is true. And the frame is equal to capture.read. Now this capture.read method reads this video frame by frame and returns the frame and a Boolean that says whether the frame was successfully read in or not.

To display this video, we can actually show individuals frame by frame. We do this by giving the command below:

```
    cv.imshow('Video', frame)
```

Now, for some way to stop the video from playing indefinitely, we will stop

the while loop with this line:

```
if cv.waitKey(20) & 0xFF== ord('d'):
        break
capture.release()
cv.destroyAllWindows()
```

So basically, just to recap, the capture variable is an instance of the video capture line we used at the beginning of the script. Inside of while loop, we grab the video frame by frame. By utilizing the captured read method, we display each frame of the video by using the cv.imshow method. And finally, for some way to break out of this while loop, we say if cv.waitKey(20) & 0xFF== ord('d'), which basically says that if the letter D is pressed, then break out of this loop and stop displaying the video.

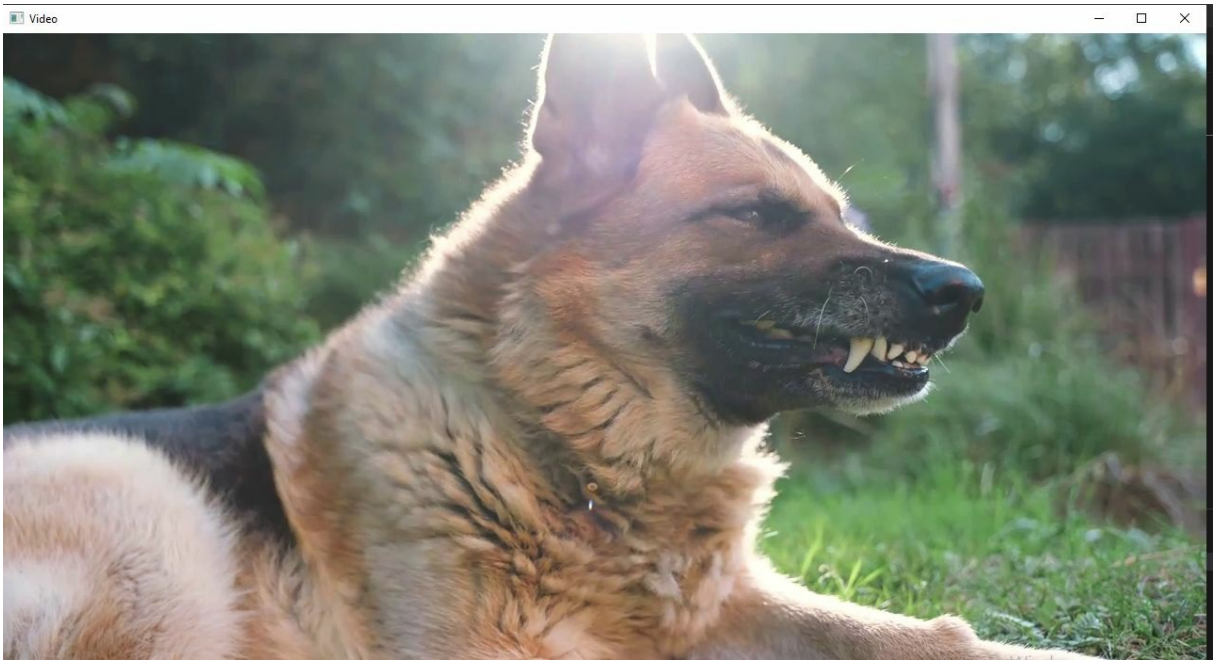
And finally, we release the capture device and destroy all the windows since we don't need them anymore.

Here is the complete script:

```
import cv2 as cv


#img = cv.imread('images/puppy.jpg', cv.IMREAD_GRAYSCALE)


#cv.imshow('puppy', img)


#reading Videos


capture = cv.VideoCapture('video/dog.mp4')
while True:


    isTrue, frame = capture.read()
```

```
    cv.imshow('Video', frame)

    if cv.waitKey(20) & 0xFF== ord('d'):
        break
capture.release()
cv.destroyAllWindows()
```

So let's save that and run. And we get a video displayed in a window like this.



If you follow the script and use it correctly, you should find that the video suddenly stops and gives an error.

More specifically, a -215 assertion failed error. If you ever get an error like this, the -215 assertion failed. This would mean in almost all cases that OpenCV could not find a media file at that particular location that you specified. Now, the reason why it happened in the video is that the video ran out of frames. OpenCV could not find any more frames after the last frame in

this video. So, it unexpectedly broke out of the while loop by itself by raising a cv2 error.

You will get the same error if we specify the wrong path to an image. For instance, we will get the same error in the first image we read if the path is wrong. This basically again says that OpenCV could not find the image or the video frame at a particular location.

So that's pretty much it. For this chapter, we talked about how to read any images in OpenCV and how to read in videos using the video capture method class. In the next chapter, we'll be talking about how to rescale and resize images and video frames in OpenCV.

Chapter 3 - Resizing and Rescaling Frames

In this chapter, we're going to be talking about how to resize and rescale images and video frames in OpenCV.

We usually resize and rescale video files and images to prevent computational strain. Large media files tend to store a lot of information, and displaying it takes up a lot of processing needs that your computer needs to assign.

So by resizing and rescaling, we're actually trying to get rid of some of that information. Rescaling video implies modifying its height and width to a particular height and width. So, let's assume that you have a very large image you want to read with Python. It may just get outside the Python window when you load that image and not display everything. That is why you may need to rescale or resize the image.

Generally, it's always best practice to downscale or change the width and height of your video files to a smaller value than the original dimensions. This is because most cameras, your webcam included, do not support going higher than their maximum capability.

For example, if a camera shoots in 720 P, chances are it's not going to be able to shoot in 1080 P or higher. So, you can use OpenCV to rescale the video or the image.

Resizing Images

The goal is to teach you how to use OpenCV to resize images. If you want to resize an image, you can do so by adjusting its width, height, or both. It's also possible to retain the original image's aspect ratio in the resized version. OpenCV provides the cv.resize() function for resizing images.

This is the syntax for the cv.resize() function:

```
cv2.resize(src, dsize[, dst[, fx[, fy[, interpolation]]]])
```

Where:

src = path of the image or video

dsize = the desired size for the new resized image

fx = [optional] the factor scale along the horizontal axis

fy = [optional] the scale factor for the vertical axis

And lastly, interpolation.

In OpenCV, there are numerous interpolation algorithms, and we will try to learn a few of them through examples. But first, what is interpolation?

Interpolation means to insert estimate values of (data or a function) between two known values between other things or parts. In OpenCV, it means you want to insert a fraction of what the video or image you are reading is. There are 4 types of interpolation in OpenCV:

1. INTER_NEAREST is an interpolation based on the nearest object in the script.

2. INTER_LINEAR is a bilinear interpolation and is used mainly by default.

3. INTER AREA – pixel area relation resampling. It may be a better method for image decimation because it eliminates moire. However, when the image is zoomed, the method is similar to INTER NEAREST.

4. INTER CUBIC – a 44-pixel neighborhood bicubic interpolation

5. INTER LANCZOS4 – a Lanczos interpolation over an 8x8 pixel area

We will do that now:

Scale percent is the percentage by which the image is to be scaled in the following example. Giving a value less than 100 reduces image quality. Using the original image's dimensions, as well as this scale percent value, we can determine the output image's width and height.

Look at the script:

```
import cv2 as cv
```

```
img = cv.imread('Desktop\images\baby.jpg', cv.IMREAD_UNCHANGED)

print('Original Dimensions : ',img.shape)

scale_percent = 60 # percent of original size
width = int(img.shape[1] * scale_percent / 100)
height = int(img.shape[0] * scale_percent / 100)
dimension = (width, height)

# resize image
resized = cv.resize(img, dimension, interpolation = cv.INTER_AREA)

print('Resized Dimensions : ',resized.shape)

cv.imshow("Resized image", resized)
cv.waitKey(0)
cv.destroyAllWindows()
```

The resize() function was used to reduce the original image's dimensions from [149 x 200 x 4] to [89, 120, 4].

Rescaling a Video

So, to rescale a video frame or an image, we can create a function called

```
def rescaleFrame(frame, scale=0.75)
```

And we can pass in the frame to be resized and scale the value, which, by default, we're going to set as 0.75.

Next, we will state the frame shape or size by height and width.

```
width = frame.shape [1] * scale

height = frame.shape [0] * scale
```

Now, remember frame.shape [1] is the width of your frame or your image and frame.shape [0] is basically the height of the image.

Since width and height are integers, we can convert these floating-point values to an integer using the int() function.

Next, we're going to create a variable called dimensions and set this equal to a tuple of width and height. So, we want to rescale a video file.

Create a folder for videos where your image folder is or anywhere you can quickly grab the path from. We will resize the video by 75%, which is the default function in OpenCV. I will first write out the complete script and then explain each line:

Let us check out what the code will be like below:

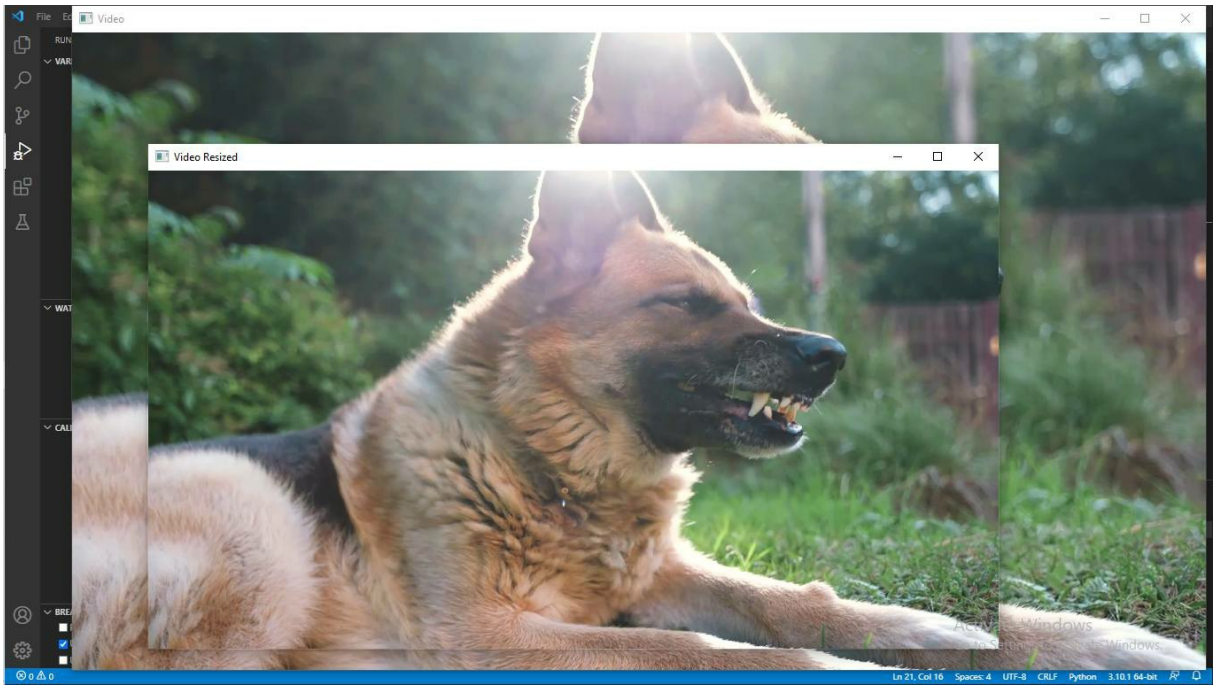
```
import cv2 as cv


def rescaleFrame (frame, scale=0.75):
    width = int(frame.shape [1] * scale)
    height = int(frame.shape [0] * scale)
    dimensions = (width,height)


    return cv.resize (frame, dimensions, interpolation=cv.INTER_AREA)


capture = cv.VideoCapture('videos/dog.mp4')
```

```
while True:
    isTrue, frame = capture.read()

    frame_resized = rescaleFrame(frame)

    cv.imshow('Video', frame)

    cv.imshow('Video Resized', frame_resized)

    if cv.waitkey(20) & 0xFF==ord('d'):
        break

capture.release()
cv.destroyAllWindows
```

While we will learn more about the cv.resize() function in upcoming chapters, for now, just note that it resizes the frame to a particular dimension. So, all the function does is take in the frame and scale that frame by a specific scalar value, which by default is 0.75.

If you run this code, you will see two different windows. One is your original video. And the second is actually a resized video with the video resized by 75%.

You can modify this percentage decrease by changing the value scale to maybe 0.2, so we rescaled it to 20%. And we get an even smaller video in a new window.



In the next part, we will draw and write on images in OpenCV.

Chapter 4 - Drawing Shapes & Putting Text on Images

In this chapter, we're going to be talking about how to draw and write on images. One of the essential things you can do with OpenCV is draw on the image. Adding lines, circles, and other geometric shapes to an image is a skill that will come in handy in the future.

When doing image analysis, you often want to draw attention to a specific part of the image. For example, you could add a rectangle around that part or an arrow to show that. Other examples could be adding a grid or a line around a figure, etc.

In this article, you'll see that these operations can be done with simple functions, each of which is designed to work with a certain type of basic geometric figure. Geometric coordinates for the figures can be specified using these functions. These basic geometric shapes, such as points, lines, circles, and rectangles, are called primitive and allow the construction of more complex shapes.

Suppose you've used other programming languages to design before. In that case, you'll find this set of commands very easy to follow and understand. If you don't know how to use these kinds of commands, don't worry. Each one will be explained in detail and in the following examples step by step.

Starting

The first thing you must do is create a new file and call it anything you like. I will call mine draw.py. (Always remember the .py)

First, we will import the packages we need. Cv2, and Numpy.

Use the following code:

```
import cv2 as cv

import numpy as np


img = cv.imread('images/baby.jpg')

cv.imshow('Baby', img)
```

```
cv.waitKey(0)
```

This is calling in the NumPy package that OpenCV had installed previously in chapter 1. What the above code is doing, as you can remember from the previous chapter, is to read or display the image of the baby from the folder in a new window.

Now there are two ways we can draw on images:

a) by actually drawing on standalone images like this image of a baby

b) or we can create a dummy image or a blank canvas to draw on.

If you want to create a blank image, you will have to use a line like this:

```
blank = np.zeros(512,512), dtype=’uint8’)
```

uint8 is basically an image, that is the data type of an image. So if you want to try and see this image, see what this image looks like. You have to type in the next code as

```
cv.imshow('Blank', blank)
```

When you save and run, you should see a blank canvas like this:

And this is basically the blank image that you can draw on. So we're going to be using that instead of drawing on the Baby image. But you can actually use anything, whether a blank canvas or a picture you have displayed on Python OpenCV.

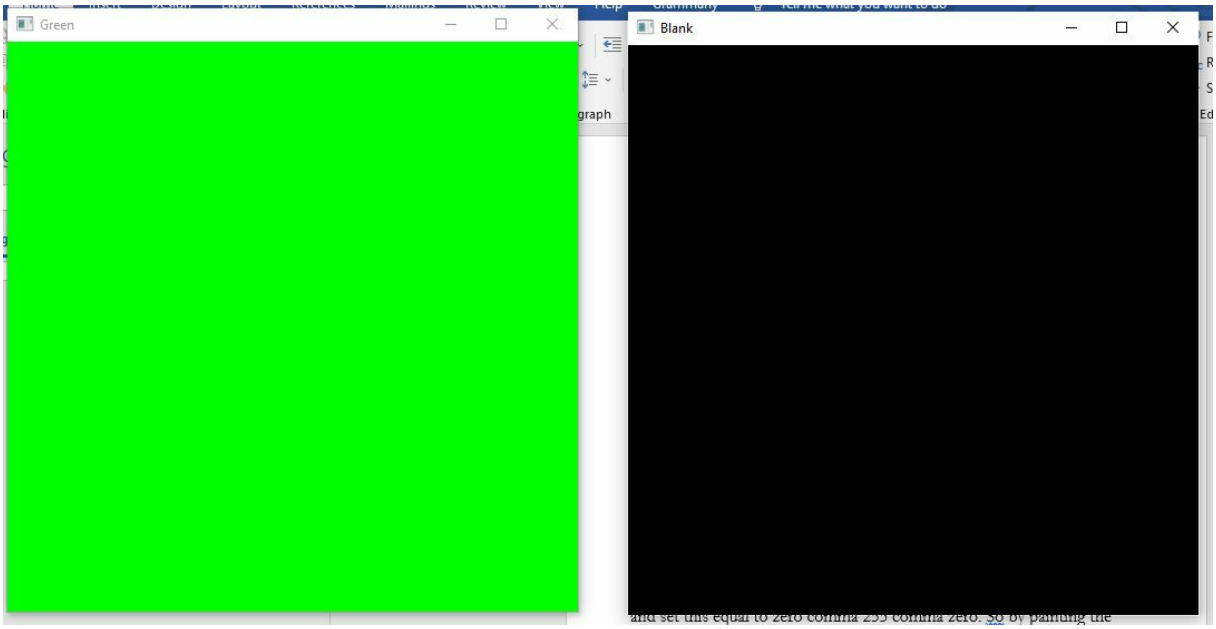
The first thing we're going to do is try to paint the image a certain color.

Using Colours

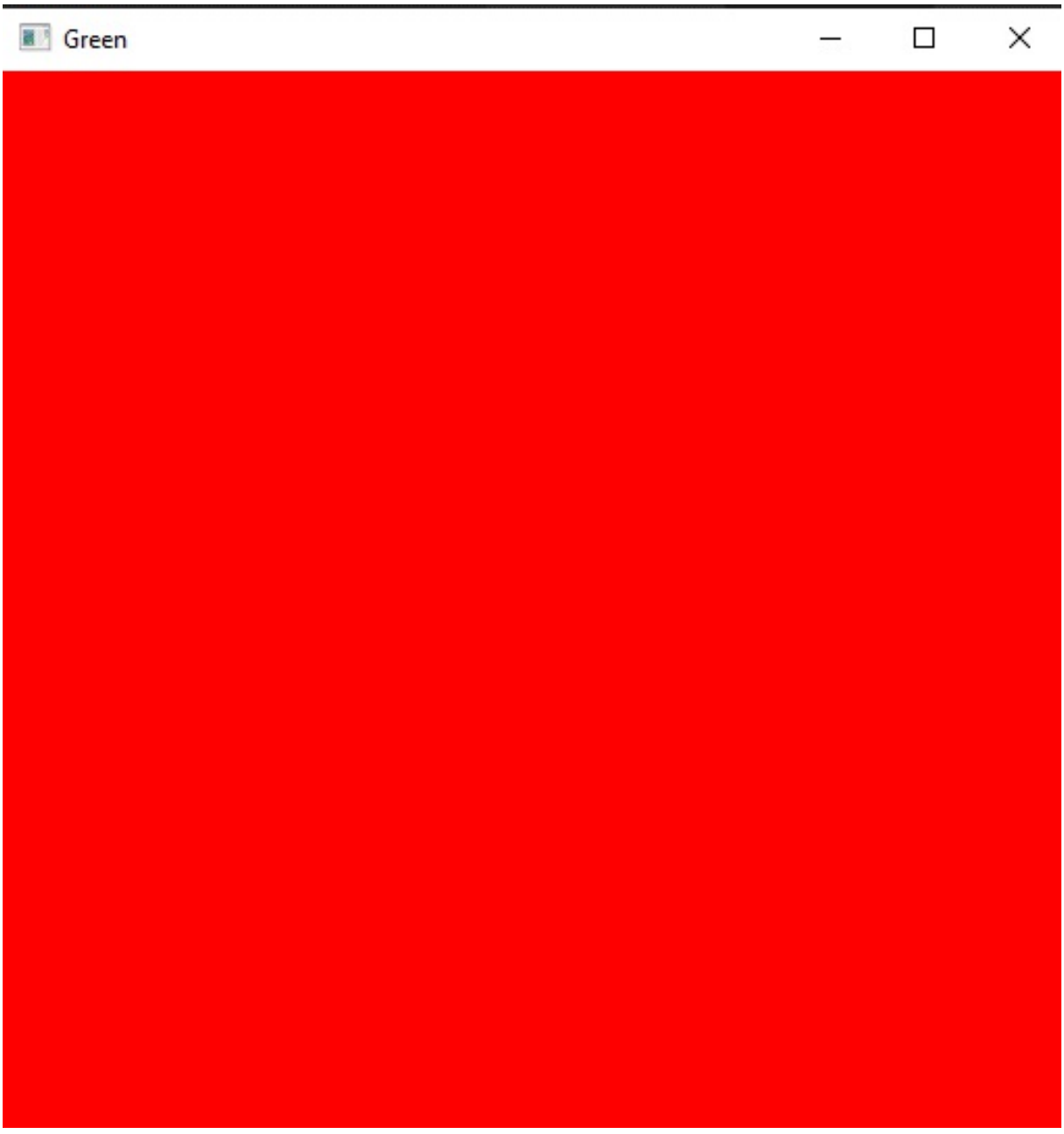
We want first to paint a color on the blank canvas. We will use the following code:

```
import cv2 as cv

import numpy as np


blank = np.zeros((512,512,3), np.uint8)


cv.imshow('Blank',blank)


blank[:] = 0,255,0

cv.imshow('Green', blank)


cv.waitKey(0)
```

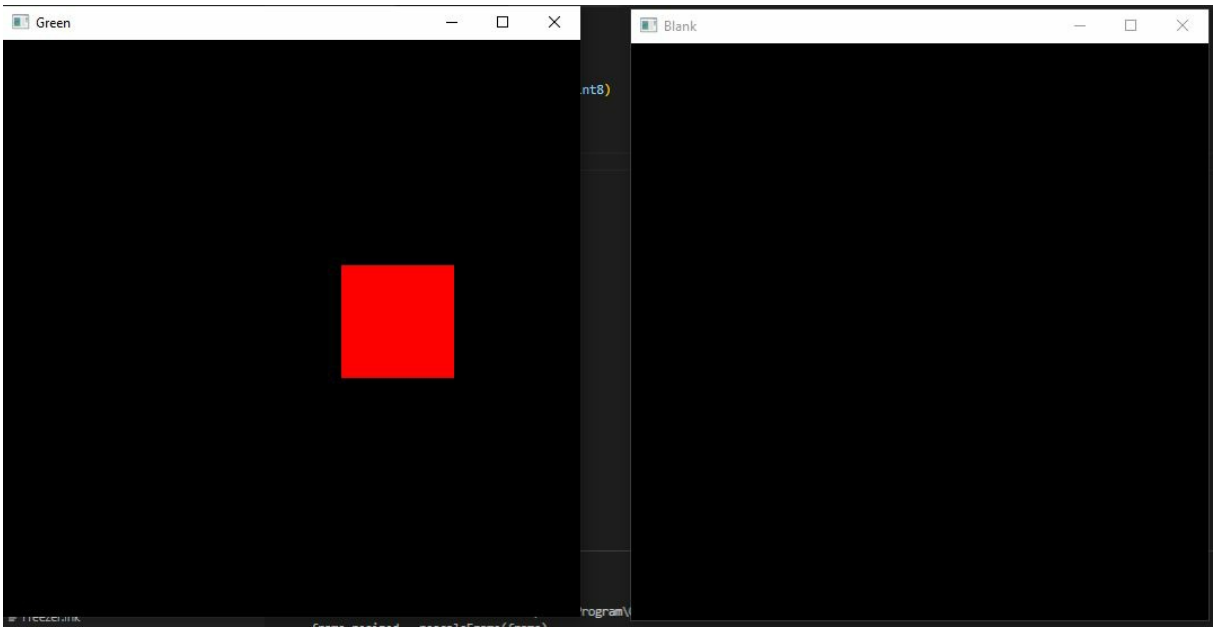
And the way we do this is by saying blank and referencing all the pixels, and setting the blank variable to 0,255,0. That will be painting the entire image green, and we can display this image by saying green in passing the blank image, save that, and run. And this is the green canvas we get.



If you want to change to red, you will change that color ref to 0,0,255.



You can also call a certain portion of the image by giving it a range of pixels. For example, check this out:



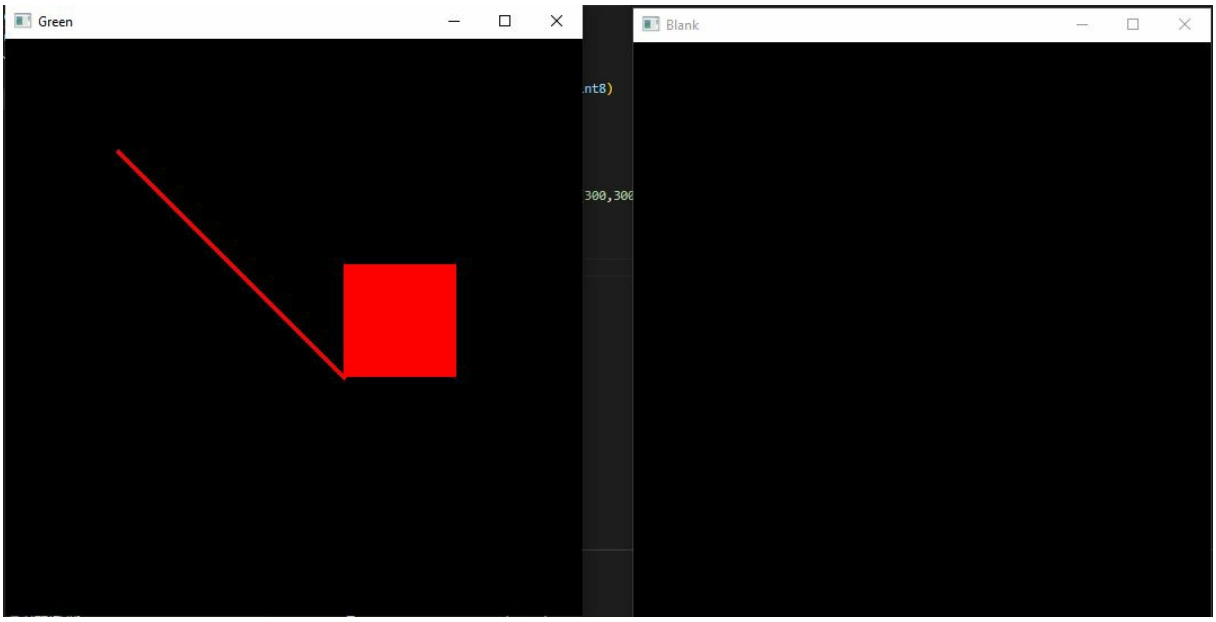
This is us saying that OpenCV should call up a range of 200 to 300. And then from 300 to 400. That gave a Red Square in this image.

Draw a line

Now, let us start drawing! If you are ready, we will first draw a line. OpenCV has a separate function for drawing. The function to draw a line is the cv.line() function. This function needs five arguments:

For example, if you want to draw a red diagonal line that is 4 pixels thick. Simply add the line below to the code we have been working on:

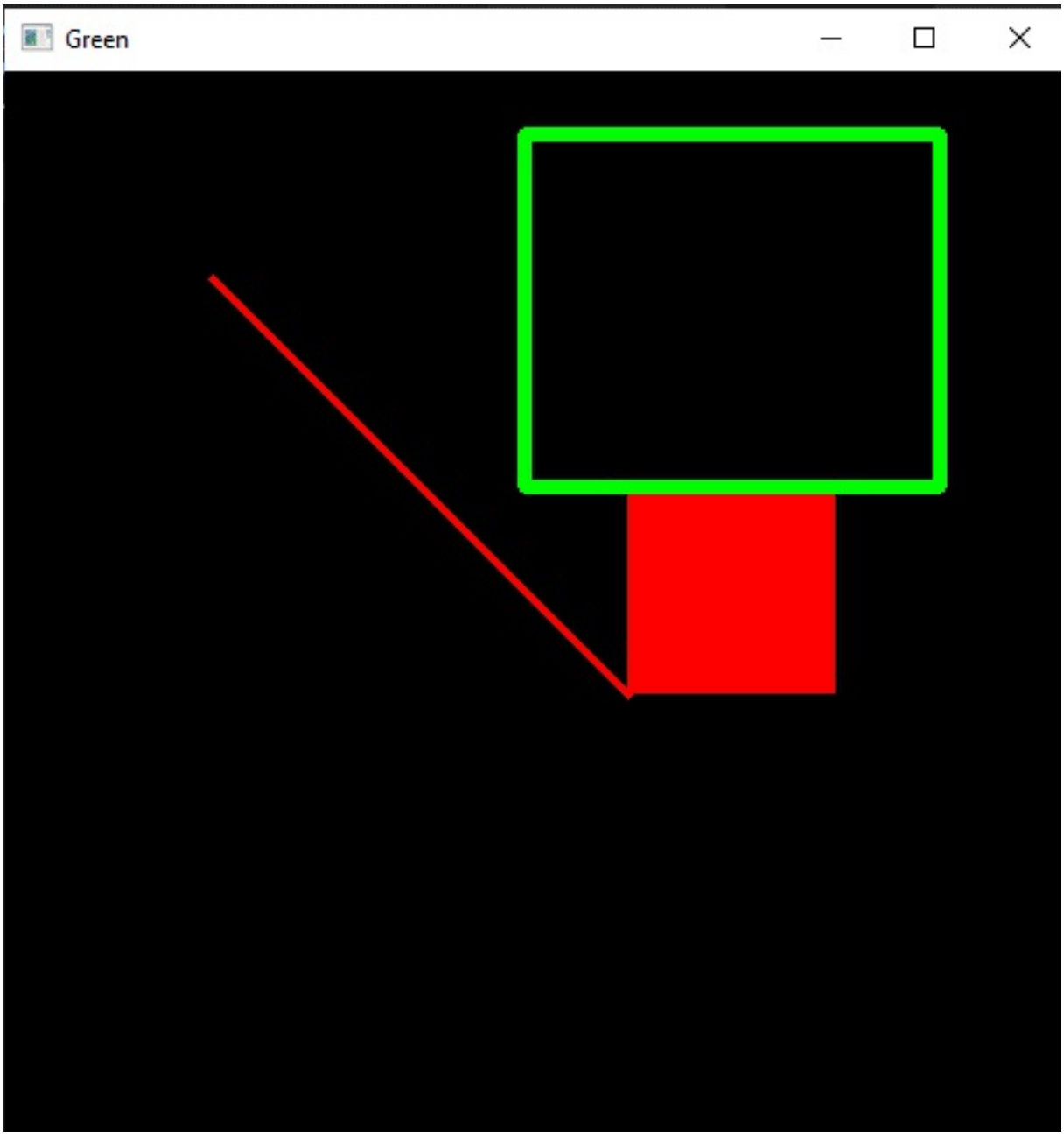
```
img = cv.line(img, (100,100), (300,300), (0,0,255),4)
```



Draw A Rectangle

The next thing to do is to draw a rectangle. And the way we do this is by using the cv.rectangle () method. This function also accepts five input parameters. To draw a four-sided shape on our screen, add the following code:

```
blank = cv.rectangle(blank, (250,30), (450,200), (0,255,0), 5)
```



This method takes in an image to draw the rectangle over, which in this case is blank. And it takes in point 1.2, color, thickness, and a line type if you'd like.

We give the rectangle the color green, which is 0,255,0.

Filling the Rectangle with Colour

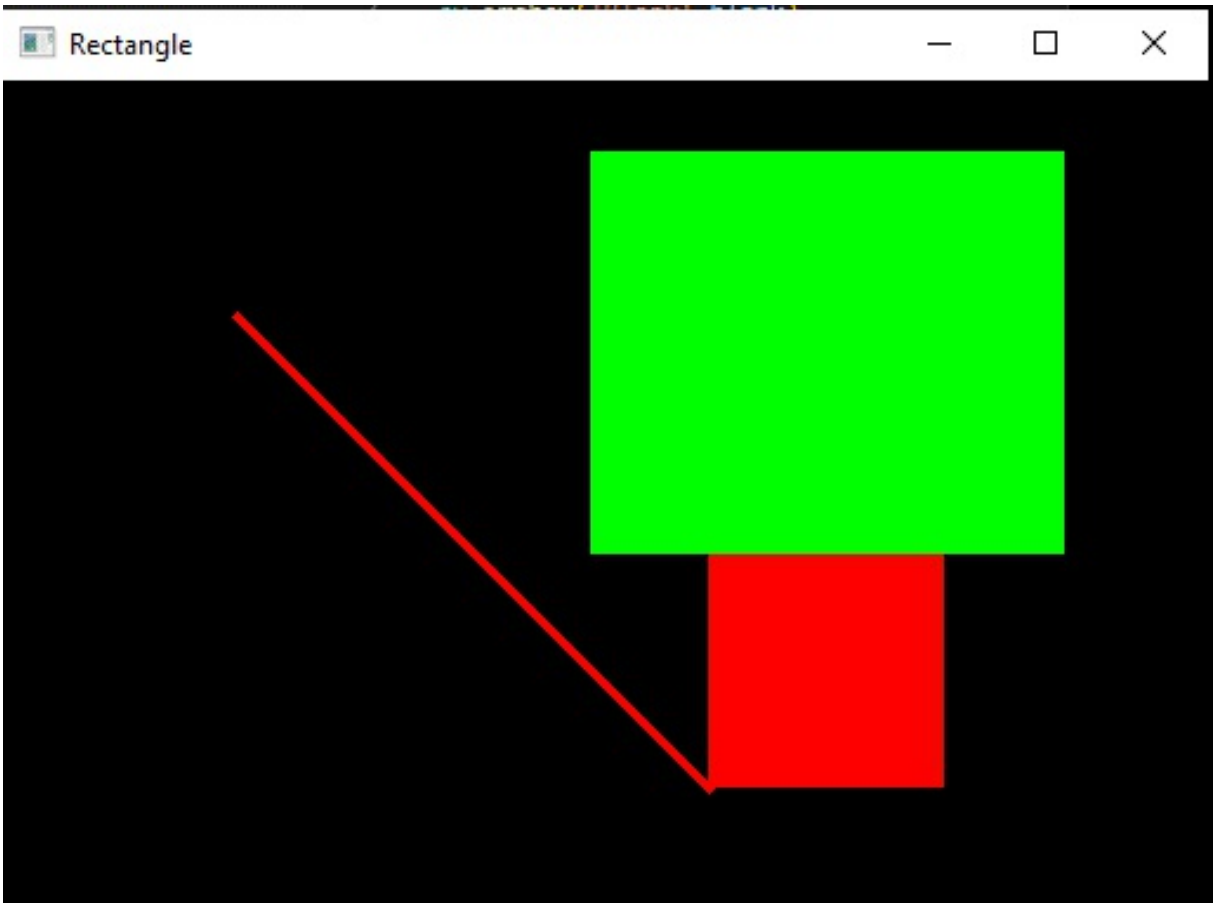
The amazing part is that we can fill the rectangle with color so that it does not

have any space.

We just added a parameter called Thickness.

```
import cv2 as cv
import numpy as np

blank = np.zeros((512,512,3), np.uint8)

cv.imshow('Blank',blank)

blank[200:300, 300:400] = 0,0,255

blank = cv.line(blank, (100,100), (300,300), (0,0,255),4)

blank = cv.rectangle(blank, (250,30), (450,200), (0,255,0), thickness = cv.FILLED)

cv.imshow('Rectangle', blank)

cv.waitKey(0)
```

Take note of the new argument passed in the blank variable. The new method cv.FILED tells OpenCV to fill the shape with the color you have specified, as you can see in the screenshot.



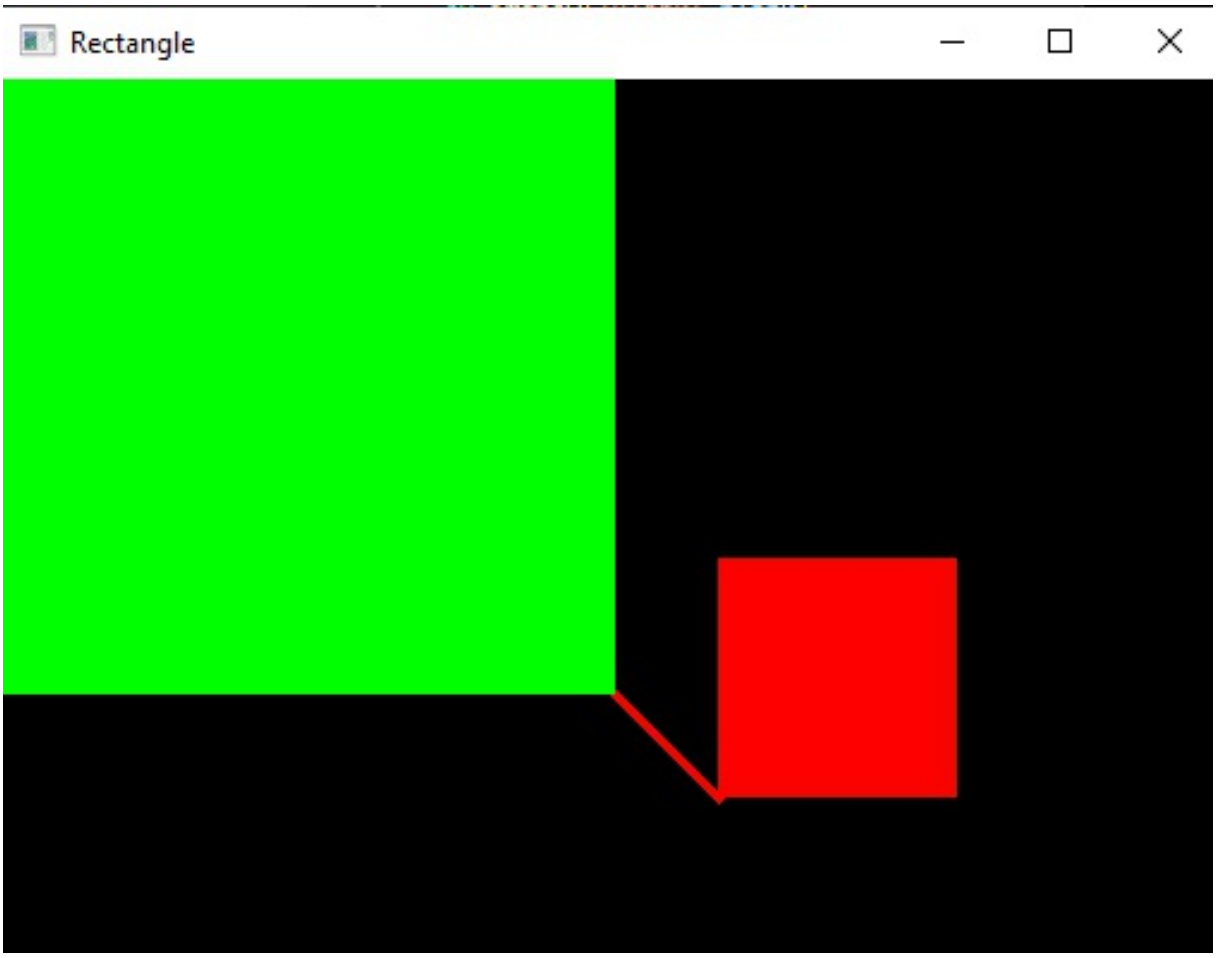
Another way to do this instead of the cv.FILLED method, you can pass the argument -1 for the thickness and get the same results.

```
blank = cv.rectangle(blank, (250,30), (450,200), (0,255,0), thickness = cv.FILLED)
```

In this line, you can see the coordinates of the rectangle as (250,30), (450,200). instead of giving it fixed values like 250, and 400, what we could do is we could say;

```
blank = cv.rectangle(blank, (0,0), (blank.shape[1]//2, blank.shape[0]//2), (0,255,0), -1)
```

We will get this result.



In this image, you can see that the script scaled the rectangle from the screen because we gave it new dimensions: half of that of the original image.

Draw a Circle

This is also fairly straightforward. We use a method called cv.circle() and pass the appropriate arguments.

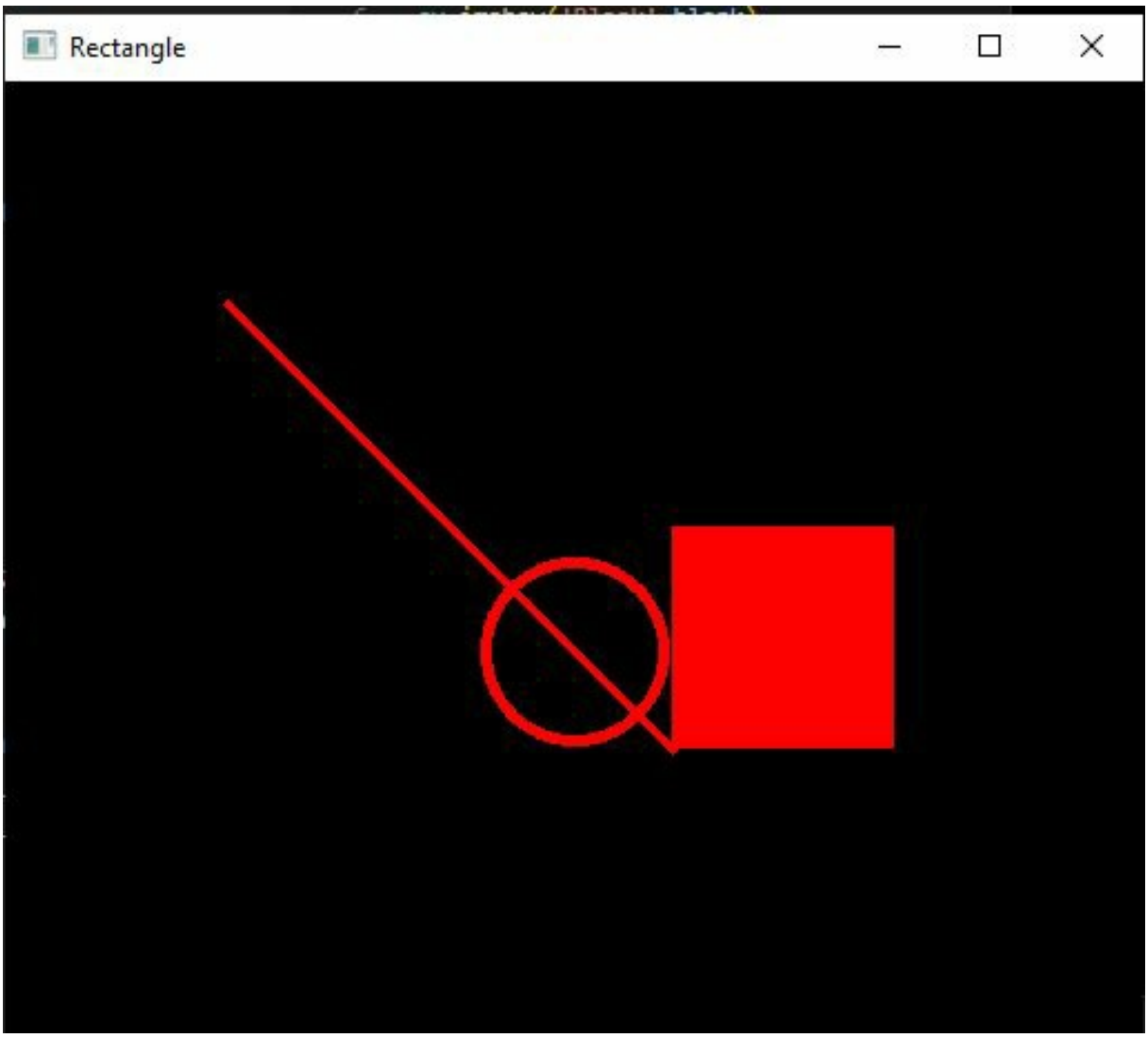
The syntax for this method is

cv.circle(img: Mat, center: Any, radius: Any, color: Any, thickness: ... = ..., lineType: ... = ..., shift: ... = ...)

That means we pass in the blank image, and give it a center, which is basically the coordinates of the center. We can set it to the midpoint of this image by using the coordinates 250, 250. You need to set the radius to like 40 pixels, giving it a color of 0,0,255. And give it a thickness of 3.

This is what the code looks like:

```
blank = cv.circle(blank, (250, 250), 40, (0,0,255), 3)
```



We can display this image, say, the circle is equal to blank. And we get a nice little circle with its center at 250, 250, and a radius of 40 pixels. Again, you can also fill in this image by giving a thickness of -1. Here, we get a nice little dot in the middle. Cool.

Write Text on Image

Use the putText() method of the Python OpenCV library to write text on an image. To draw the text with OpenCV, there is cv.putText function that

accepts a number of arguments:

Here are the arguments you must pass:

- your image or blank canvas on which to draw
- the text you want to put
- coordinates of the text
- the font you want to use
- font size
- text color
- text thickness
- line type

The fundamental ideas are the same. But before you add the actual text, you should take a quick look at the font that OpenCV gives you. This is a list of the fonts accepted in OpenCV.

FONT_HERSHEY_SIMPLEX = 0,

FONT_HERSHEY_PLAIN = 1,

FONT_HERSHEY_DUPLEX = 2,

FONT_HERSHEY_COMPLEX = 3,

FONT_HERSHEY_TRIPLEX = 4,

FONT_HERSHEY_COMPLEX_SMALL = 5,

FONT_HERSHEY_SCRIPT_SIMPLEX = 6,

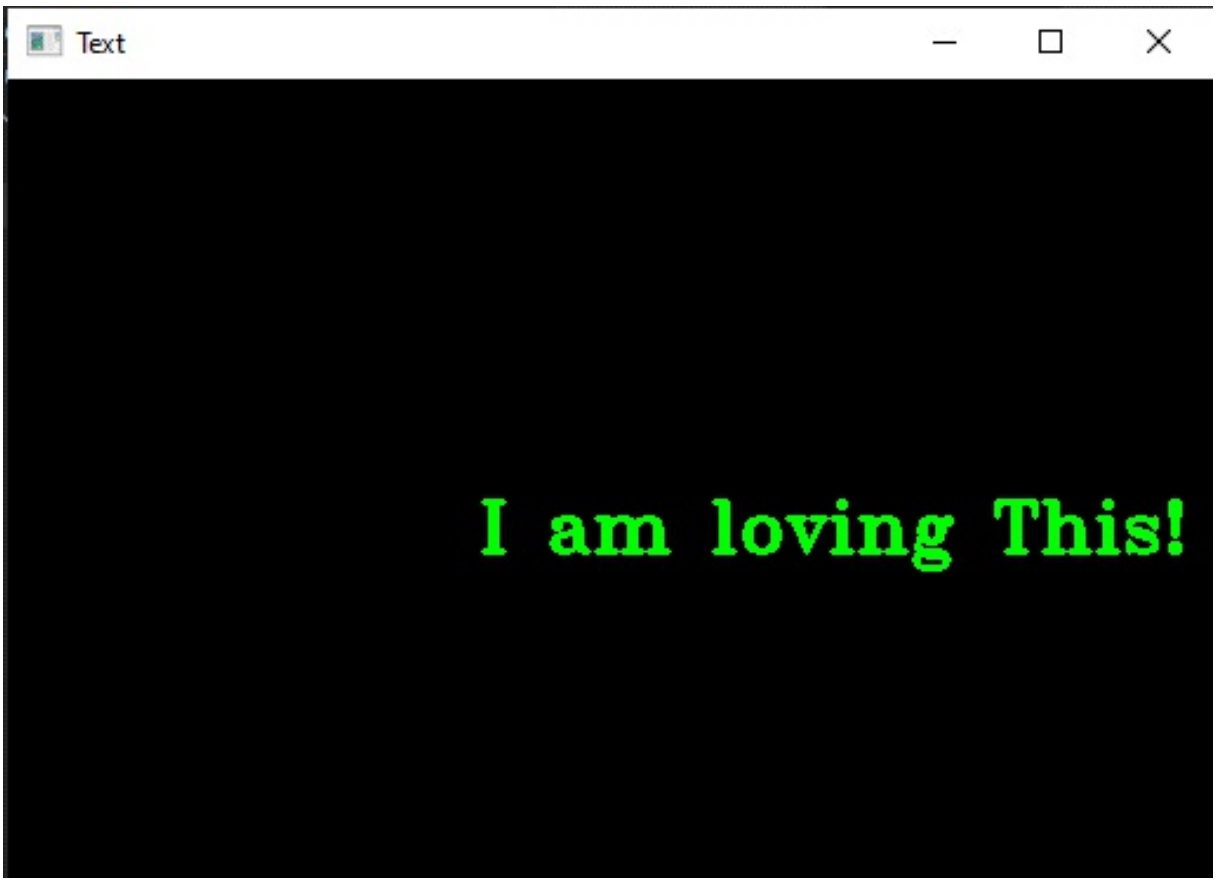
FONT_HERSHEY_SCRIPT_COMPLEX = 7,

The following example shows how to use the putText() function.

```
import cv2 as cv

import numpy as np
```

```
blank = np.zeros((512,512,3), np.uint8)


cv.putText(blank, "I am loving This!", (200, 200), cv.FONT_HERSHEY_TRIPLEX, 1.0,
(0,255,0), 2,)


cv.imshow('Text', blank)


cv.waitKey(0)
```

When you run it, you get the text window like so:



For this chapter, we talked about how to draw shapes, draw lines, rectangles,

and circles, and write text on an image. Now in the next chapter, we'll be talking about basic functions in OpenCV that you're most likely going to come across in whatever project in computer vision you end up doing.

Chapter 5 – Basic Functions You Must Use in OpenCV

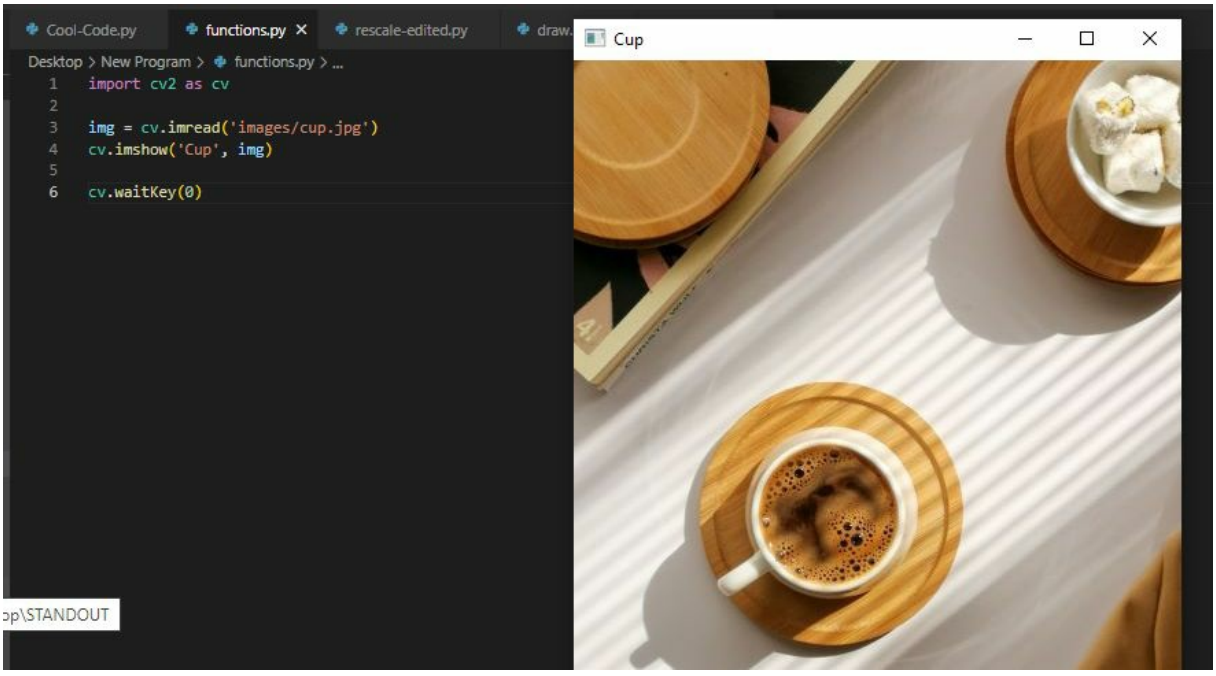
In this chapter, we will be talking about the most basic functions in OpenCV that you're going to come across in whatever computer vision project you end up building.

Converting An Image to Greyscale

So first, load an image. Let's say we have written an image, and we've displayed that image in a new window. Most images like the image of the puppy we displayed earlier are BGR images, a three-channel (blue, green, and red image).

Now there are ways in openCV to convert those BGR images to grayscale so that you only see the intensity distribution of pixels rather than the color itself.

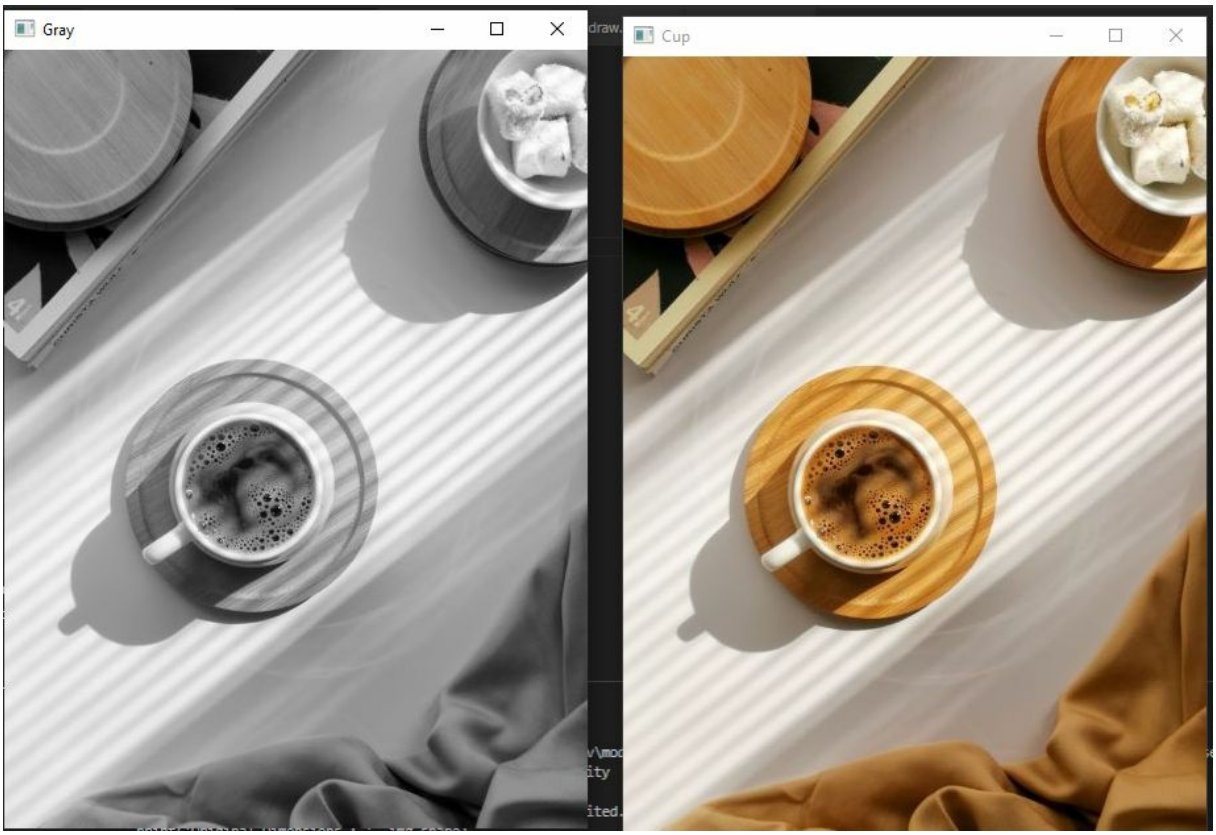
To convert an RGB image to gray scale image, OpenCV has a function called cv.cvtColor(). It comes with the argument <img, color_feature(for converting to grayscale it is cv.COLOR_BGR2GRAY)>



This window is the image display.

Now to convert it to greyscale, we will use the function. Add the following line to your script to change your image to greyscale:

```
gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)


cv.imshow('Gray', gray)
```



So, we created a gray variable and attribute the function cv.cvtColor(). We pass in the image that we want to convert from, which is img, and we specify a color code.

Now, this kind of code is cv.COLOR_BGR2GRAY, since we're converting a BGR image to a grayscale image. And we can go ahead and display this image by saying cv.imshow passing gray and passing the gray image.

You will see the original and grayscale images when you run. You can try it with as many images as you like. Nothing too fancy. We've just converted

from a BGR image to a grayscale image.

That is an essential function every Computer Vision expert must first know.

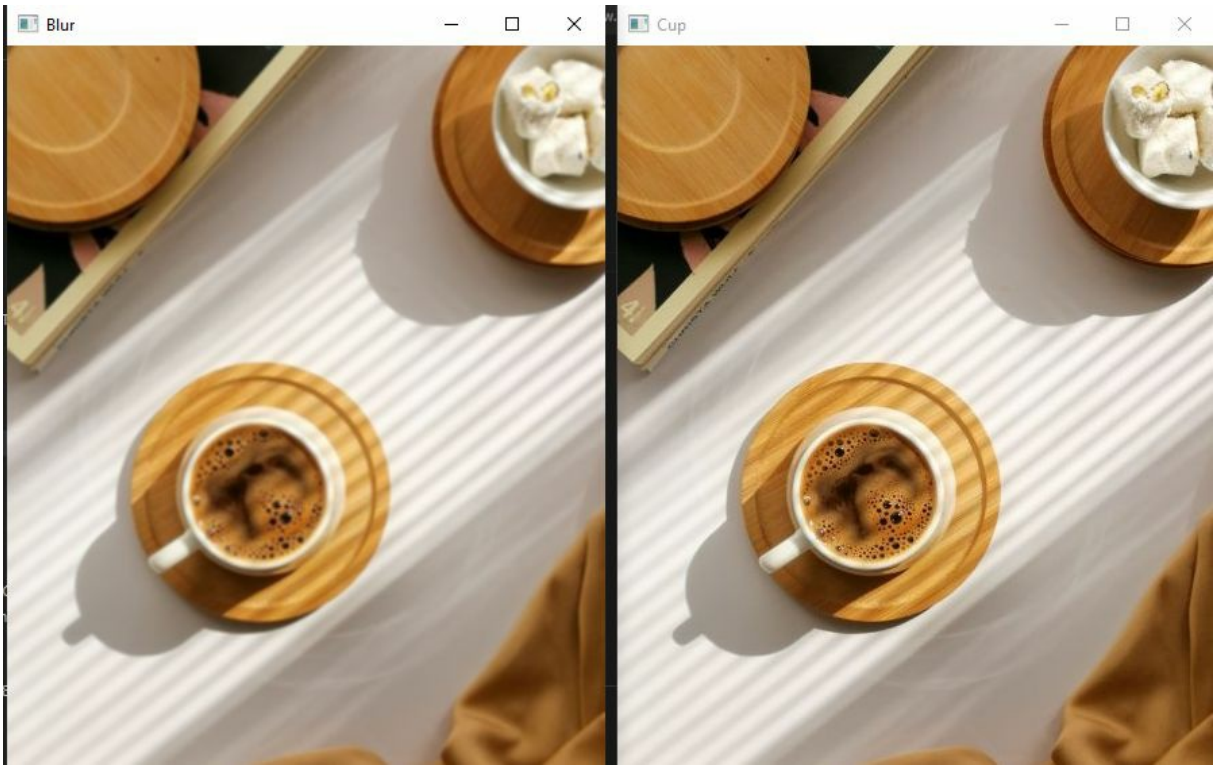
Blurring an image

Now blurring an image essentially removes some of the noise that exists in an image. For example, there may be some extra elements in an image because of bad lighting when the image was taken, or maybe some issues with the camera sensor and so on.

One of the ways we can reduce this noise is by applying a slight blur. There are way too many blurring techniques that we will get into in the advanced part of this book. But for now, we're just going to use the Gaussian Blur. So, what we're going to do is we're going to create a blurred image of the cup of coffee we displayed earlier.

The function to do this is the Gaussian filter, as I mentioned. The function is cv.GaussianBlur(). It has the following arguments, <img, kernel_size, border_type>.

Kernel size is the size of the kernel we want to apply the filter. It is actually a two-by-two tuple, which is the window size that OpenCV uses to compute the blurred image. We'll get into this in the advanced part of the book, so don't worry too much about this. Just know that this kernel size (ksize) has to be an odd number.



Now, you will be able to notice some of the differences in this image. And that is because of the blur that is applied to it. In the screenshot of the two windows, you see the coffee bubbles are pretty clear on the second image, and over on the first, they're slightly blurred.

To increase a blur in your, we can increase the kernel size from three by three to seven by seven, save that, and run. And this is the image that is way more blurred than the previous image. So that's it.

Creating Edge Cascade

Now we're going to discuss how to create an edge cascade, which is trying to find the edges that are present in the image.

In edge cascade, we will look for the edges that are in an image. There are a lot of edge cascade functions, but we use the canny edge detector, which is well-known in the world of computer vision.

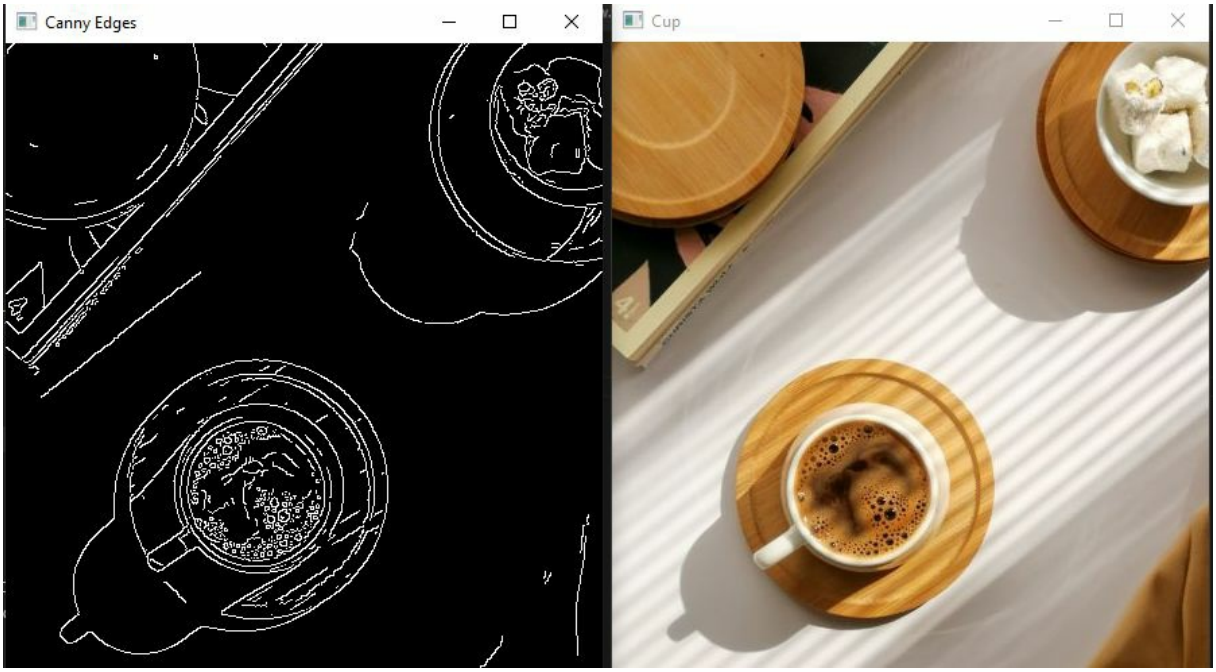
Essentially, it's a multi-step process that involves a lot of blurring and then involves a lot of grading computations and stuff like that.

So we create a variable called edge and assign it to cv.Canny(), we pass in the

image and the threshold values of 125 and 175. When we displayed the code, you can see in the screenshot below:

If you look at the image, you will see the edges that were found in this coffee image. There are not a lot of features in the picture, though.

```
edge = cv.Canny(img, 125, 175)


cv.imshow('Canny Edges', edge)
```



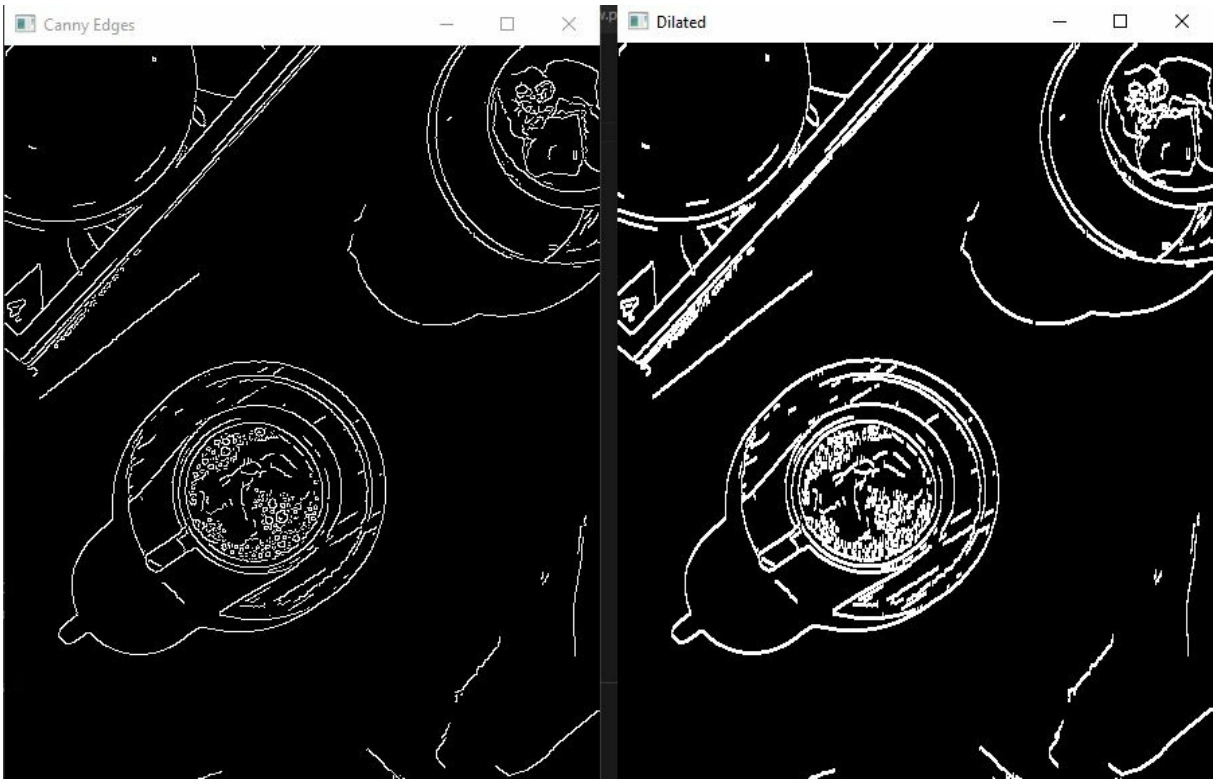
We can reduce some of these edges by essentially blurring the image. And the way we do that is instead of passing the img. We pass in the blur. Try that and run the code. And as you can see, there were far fewer edges found in the image. And this is a way you can reduce the number of edges that were found by a lot by applying a lot of blur or get rid of some of the edges by applying a slight blur.

How to Dilate an Image

Dilation is a morphological procedure that is used to make an image look better. This means you want to enhance some features of an image following a specific structuring element. Now the structuring element that we are going

to use is these edges we created before. The function we will use is called cv.dilate().

```
dilate = cv.dilate(edge, (5,5), iterations=2)


cv.imshow('Dilated', dilate)
```



Did you see the difference between the two images? Let me explain the arguments a little. We passed in the src, "blur, " and took a kernel size, which we'll specify as 5 by 5. And it also took in iterations of 2.

Now, dilation can be applied using several iterations of the time, but for now, we're just going to stick with 2.

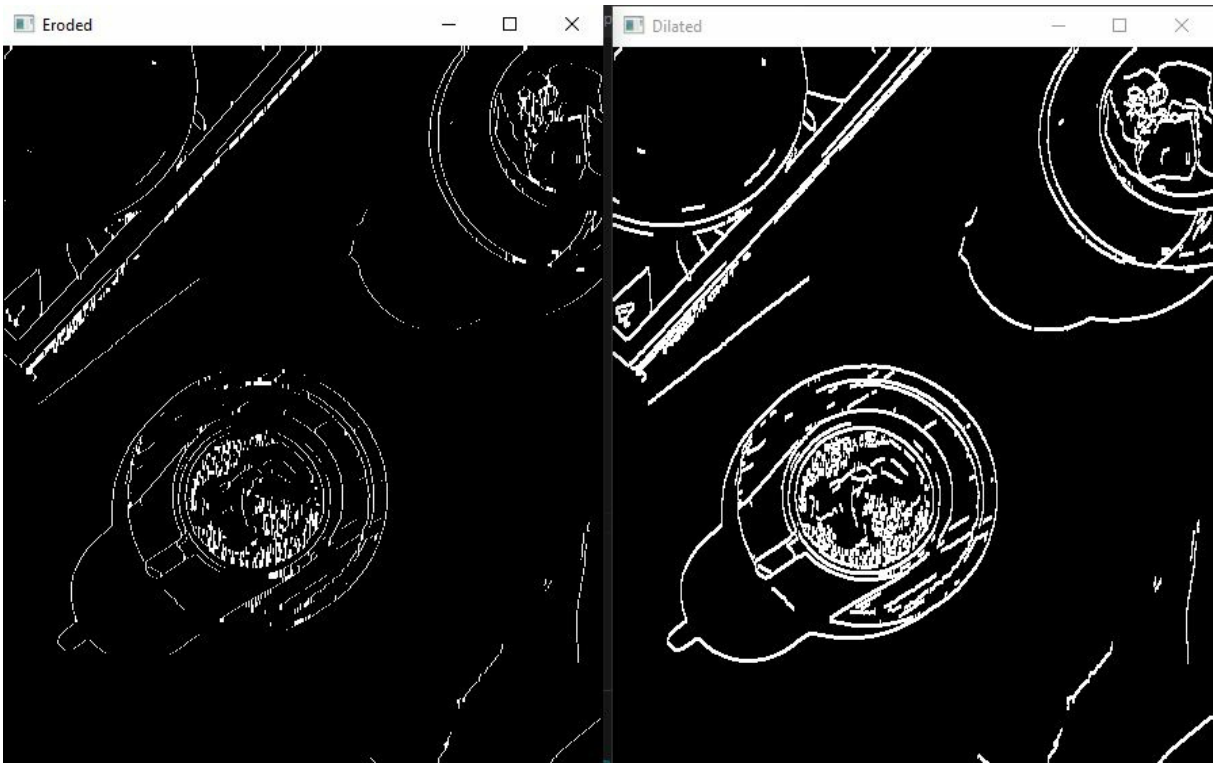
The point of the dilate is to enhance the features of the image, and you can see how cool that is.

Erosion

Erosion removes pixels on object boundaries. This is a way of eroding this dilated image to get back this structuring element. Now, it's not going to be perfect, but it will work in some cases. This operation can be done with the cv.erode() function.

The function will take in the dilated image, a kernel size of 3 by 3, and given iterations of 3.

```
eroded = cv.erode(dilate, (3,3), iterations=3)


cv.imshow('Eroded', eroded)
```



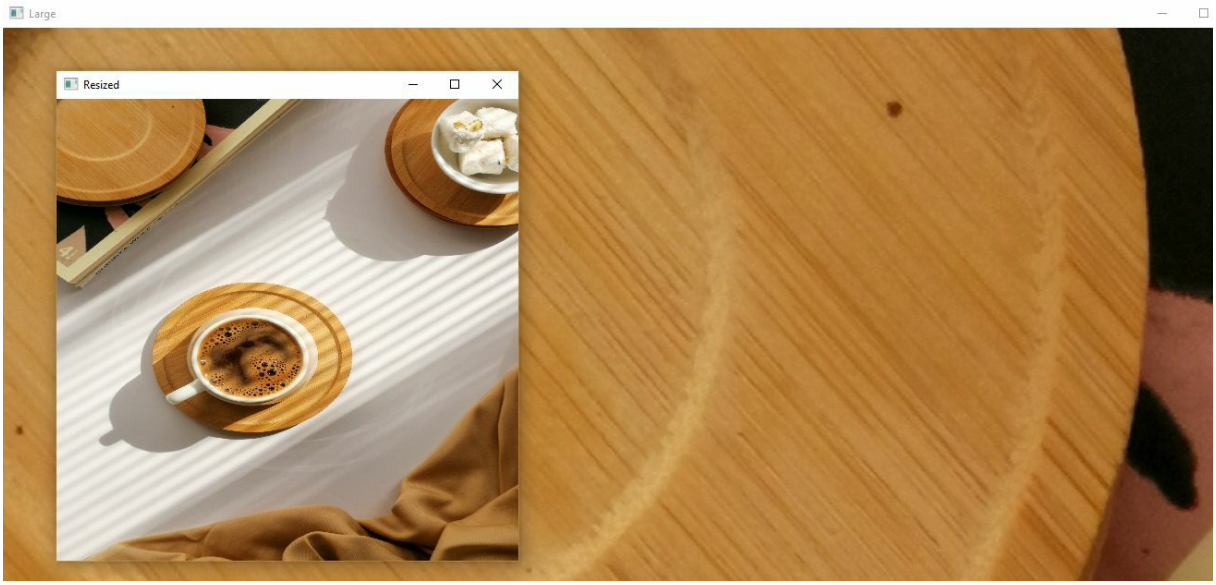
And as you see, the display image shows the eroded version of the dilated image. Now, it isn't the same as a structural element of the dilated image. But you can just about to make the features that. But you can see that there is a subtle change between the edges and the thickness of the edges between the

two.

Resize and Crop an Image

First, we come to resizing video frames and images in the previous chapter. Now, we want to talk about the special function designed to resize images, and that is the cv.resize() function.

I will load a large image and then resize it.



The large image is so large that it went out of the window screen display.

The code is as follows:

```
imglarge = cv.imread('images/cuplarge.jpg')
cv.imshow('Large', imglarge)


resized = cv.resize(imglarge, (500,500))
cv.imshow('Resized', resized)
```

The cv.resize() function takes an image to be resized, and it will take in a destination size, which we set to 500 by 500. And so this essentially takes in

this image of the coffee cup, the large size, and resizes that image to 500 by 500, ignoring the original aspect ratio.

You can see the original image and the image that was resized to 500 by 500.

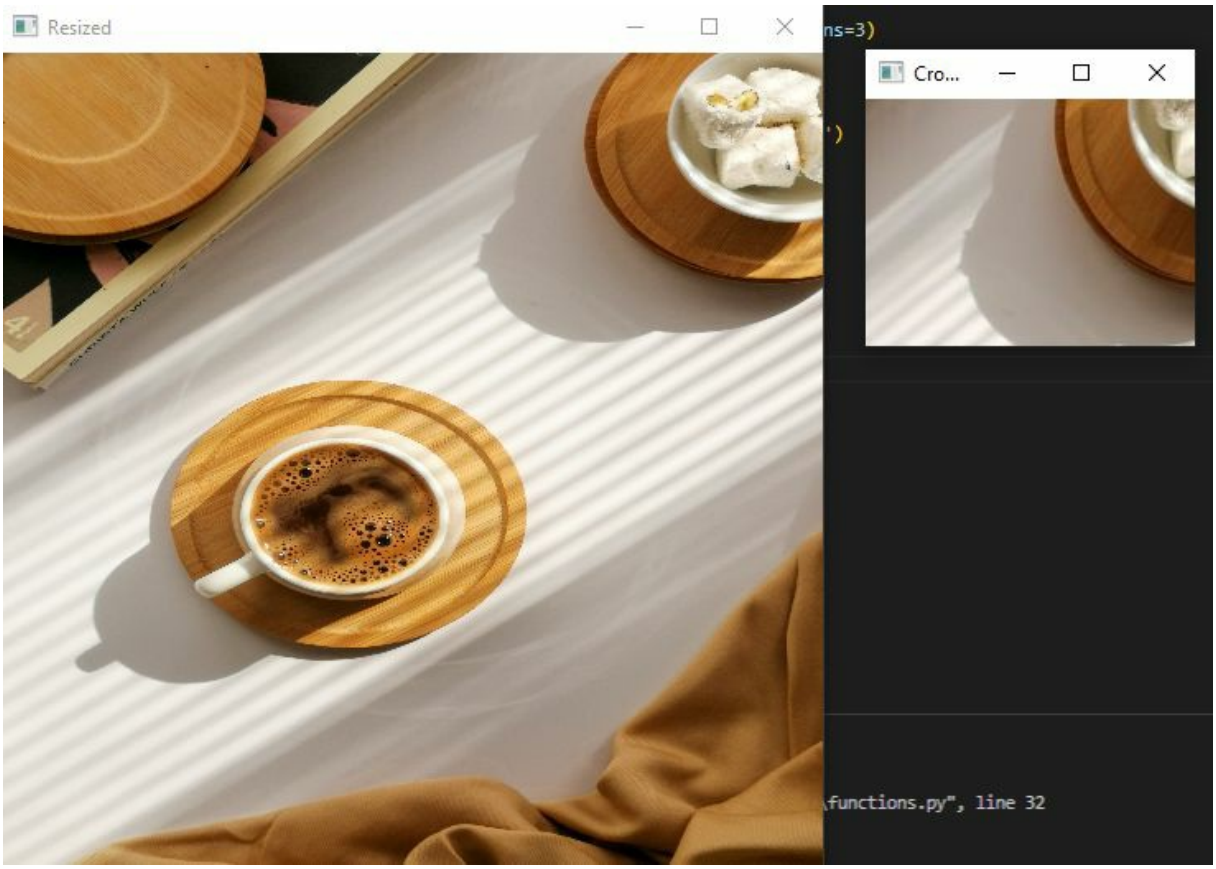
Now by default, an interpolation occurs in the background, and that is cv.INTER_AREA. This interpolation method is useful if you shrink the image to smaller dimensions than the original ones. But in some cases, if you are trying to enlarge the image and scale the image to a much larger dimension, you will probably use the INTER_LINEAR or the INTER_CUBIC. Now cubic is the slowest among them all. But the resulting image that you get is of a much higher quality than the INTER_AREA or the INTER_LINEAR.

Cropping an Image

We know that images are arrays. That is a list or stack of pixels. So, cropping means we want to slice the array. You have learned about slicing arrays in beginner Python tutorials. We do the same thing here.

We can select a portion of the image based on your pixel values. Look at the code and see what happens:

```
crop = img[50:200, 200:400]
cv.imshow('Cropped', crop)
```



We have the variable crop, and we made a list from the img variable. We selected a region from 50 to 200. And from 200 to 400. As you can see from the two images in the above screenshot, there is the original image, and you can see the portion that was cropped. So that's pretty much it.

For this chapter, we talked about the most basic functions in OpenCV. We talked about converting an image to grayscale by applying some blur by creating an edge cascade by dilating the image by eroding that dilated image by resizing an image and cropping an image using Array Slicing. In the next chapter, we're going to be talking about image transformations in open CV, that's translation, rotation, resizing, flipping, and cropping.

Chapter 6 - Image Transformations

In this section, we're going to cover basic image transformations. These are common techniques that you would likely apply to images, including translation, rotation, resizing, clipping and cropping. So let's start with the translation.

Translation

The process of translation is to move an image along the x and y axes. If you are working on a deep learning project, you can use these translations as steps to add to the data. Translation is shifting an image along the x and y-axis. So using translation, you can shift an image up, down, left, right, or with any combination of the above.

Unlike the other functions we have discussed in the previous chapter, this operation does not have a built-in function in OpenCV. To translate an image, we have to create a translating function. We're going to call this def translate and pass the arguments.

To accomplish this, we'll write a translation function that takes three parameters: an image, x, and y. When translating, we need a translation matrix. Numpy comes in handy for this. OpenCV's cv.warpAffine() function is used to perform the translation. Let's take a closer look at the code.

```
import cv2 as cv

import numpy as np


img = cv.imread('images/photo.jpg')

cv.imshow('Baloons', img)


def translate(img, x, y):
    transMat = np.float32([[1,0,x],[0,1,y]])
    dimensions = (img.shape[1], img.shape[0])
    return cv.warpAffine(img, transMat, dimensions)
```

```
translated = translate(img, -100, 100)
cv.imshow("Translated", translated)

cv.waitKey(0)
```

According to the code, this translation function we created will take in an image to translate and take an x and y arguments. x and y basically stand for the number of pixels you want to shift along the x-axis and y-axis.

So the code is to translate an image, and we need to create a translation matrix. That is why we have that line of the transMat is equal to np.float 32. And this will take in a list with two lists inside of it. And the first list we're going to say, 1,0,x, and 0,1,y.

And once we've created our translation matrix, we can essentially get the dimensions of the image saying dimensions, which is a tuple of the width and the height. Lastly, we used the cv.warpAffine() function. This function will take the image matrix to transMat, taking the dimensions. And with that data, we can essentially translate our image.

I do want to mention that if you have negative values for x, you're essentially translating the image to the left, negative y values imply shifting up positive x values implies shifting to the right. And as you guessed, positive y values shifted down.

Feel free to play around with these values as you see fit. Just know that negative x shifts to the left, negative y shoves it up, x shifts to the right and positive y values shift down.

Rotation

Rotation is precisely what it sounds like rotating an image by some angle. You understand what a rotation is—the act of rotating an image by a certain number of degrees. The pivot point for rotation in OpenCV can be any point. Here, the cv.warpAffine() function is called with the same parameters we used for translation. However, the creation of the matrix is different.

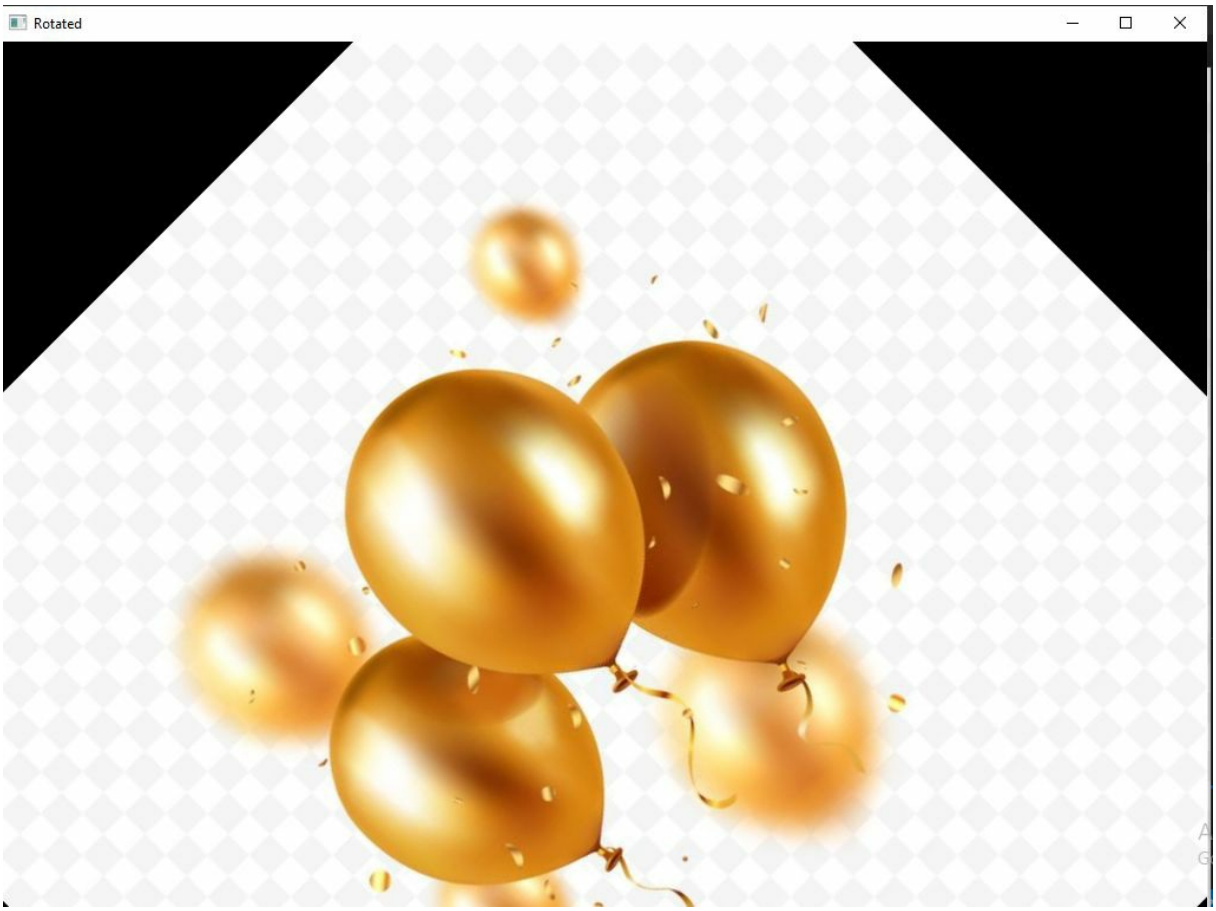
Open CV allows you to specify any rotation point that you'd like to rotate the image around.

Usually, rotation occurs with the anchor point in the center, but you could specify any arbitrary point with OpenCV. It could be any corner, it could be 10 pixels to the right, 40 pixels down, and you can shift the image around that point.

Just like there is no built-in function for translation, we also have to create the function for rotation. We create the rotating function that will take an image, an angle to rotate around and a rotation point.

Let’s see what the code looks like:

```
def rotate(img, angle, rotPoint=None):
    (height,width) = img.shape[:2]

    if rotPoint is None:
        rotPoint = (width//2, height//2)

    rotMat = cv.getRotationMatrix2D(rotPoint, angle, 1.0)
    dimensions = (width, height)
    return cv.warpAffine(img, rotMat, dimensions)

rotated = rotate(img, 45)
cv.imshow("Rotated", rotated)
```



Basically, if the rotation point is set to none, we are going to assume that we want to rotate around the center. That is why the code said rotPoint is equal to width divided by two divided by two in height divided by two.

And we created the rotation matrix as we did with the translation matrix. We passed in the center the rotation point and angle to rotate around, which is the angle and a scale value. Now we're not interested in scaling the image, so we set this to 1.0. We then set a dimensions variable equal to the width and the height. We can return the rotated image, which is a cv.warpAffine.

And then, we created a rotated image by setting this equal to rotate, and we can rotate the original image by 45 degrees. You can see the image in the screenshot above rotated by 45 degrees.

If somehow you wanted to rotate this image clockwise, just specify negative values for this angle, and it will rotate the image around rotated clockwise.

Now you can also rotate a rotated image. Take an image you have rotated and

rotate it again 45 degrees further.

When you rotate images, you will see the black background in the display. These black lines were included because if there's no image in it, if there's no part of the image in it, it's going to be black by default. So when you took this image and rotated it by 45 degrees, you essentially rotated the image but introduced the black triangles. Now, if you tried to rotate this image further by some angle, you are also trying to rotate these black triangles along with it. And you get a kind of skewed image. So far, we've covered two image transformations, translation, and rotation.

Resizing

Nothing new. We already talked about how to resize in our last chapter. We are only adding more information to what you have learned this time. Now, this is nothing too different from what we've discussed previously. But let's touch on adjusting an image by resizing.

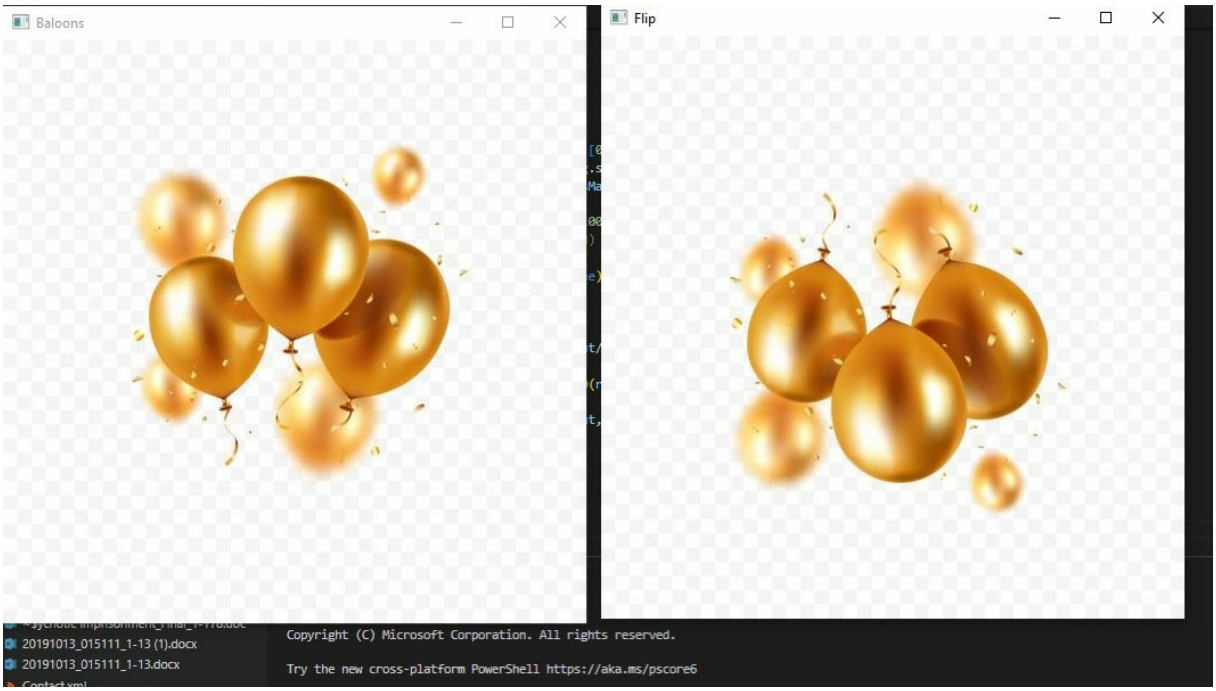
Unlike the others we have discussed in this chapter, we don't have to create a new resize function in our script. We need to create a new variable and pass in the image to resize the destination signs of maybe 500 by 500.

And by default, the interpolation is cv.INTER_AREA. You can change this to INTER_LINEAR or INTER_CUBIC. That is definitely a matter of preference, depending on whether you're enlarging or shrinking the image. You will probably go for INTER_AREA with the default if you're shrinking the image. If you're enlarging the image, you could probably use the INTER_LINEAR or the INTER_CUBIC, which is slower, but the resulting image is better with over high quality. We have discussed this before.

Flipping an image

So we don't need to define a function for this. We just need to create a variable and set this equal to cv.flip. This will take in an image and a flipped code. Now, this flip code could either be 0 or -1. Zero basically implies flipping the image vertically that is over the x-axis. 1 specifies that you want to flip the image horizontally or over the y axis, and -1 implies flipping the image both vertically as well as horizontally. So let's start off with zero

flipping it vertically.



Here is the code to do that:

```
flip = cv.flip(img, 0)
cv.imshow("Flip", flip)
```

And that is the image that was flipped vertically. You can play around with this and try out a horizontal flip. You can even display both horizontal and vertical flip images together. And if they looked like mirror images, then it was flipped horizontally. You can try to flip the image vertically and horizontally by specifying -1 as a flip code.

Chapter 6 - Contour Detection

In this chapter, we're going to be talking about how to identify contours in OpenCV. Now contours are basically the boundaries of objects, the line or curve that joins the continuous points along the boundary of an object.

Contours are valuable tools when you get into shape analysis and object detection and recognition. From a mathematical point of view, they're not the same as edges. You can get away with thinking of contours as edges for the most part. But from a mathematical point of view, contours and edges are two different things. So in this chapter, I want to introduce you to the idea of contours and how to identify them in OpenCV.

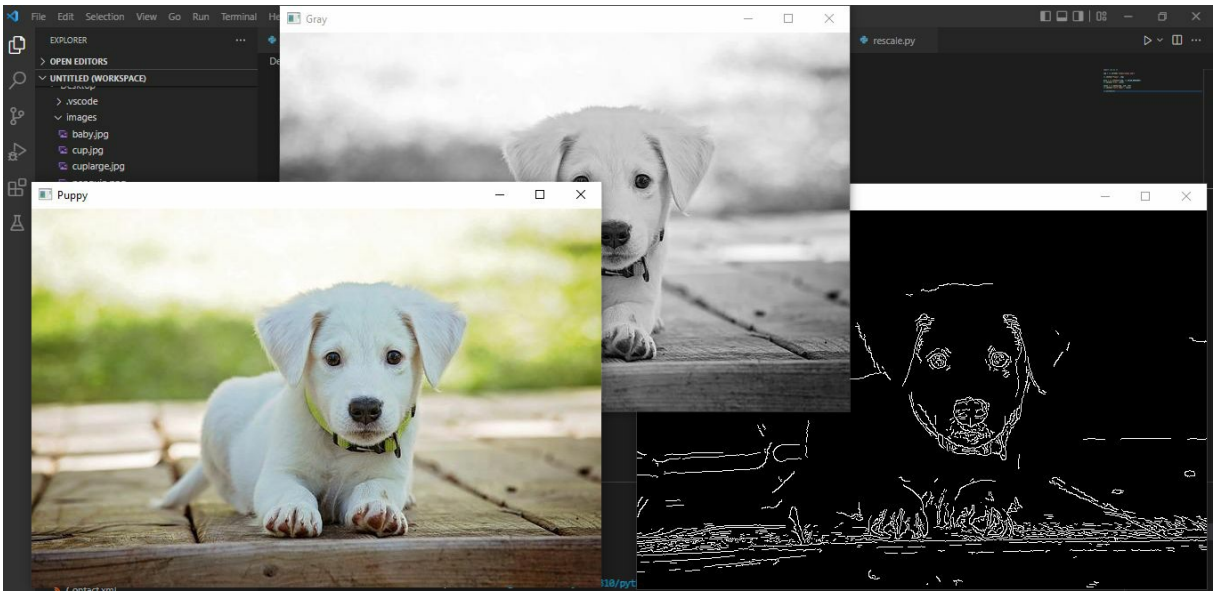
So the first thing you must do is read in your image file and display it using the cv.imshow method. The next thing you want to do is convert this image to grayscale by using the code below:

```
gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
cv.imshow('Gray', gray)
```

After this, you want to grab the edges of the image using the canny edge detector. Do you remember the canny edge detector function? Yes. cv.canny. We're going to pass in the IMG, and we're going to give it to threshold values. So 125 and 175. The code will look like this:

```
import cv2 as cv

img = cv.imread("images/puppy.jpg")

cv.imshow("Puppy", img)

gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
cv.imshow('Gray', gray)

canny = cv.Canny(img, 125, 175)
cv.imshow("Canny Edges", canny)

cv.waitKey(0)
```

When you save that and run, you will have 3 windows like so:



Now, we find the contours of this image by using the find contours method. Now, this method returns two things: contours and hierarchies.

```
contours, hierarchies = cv.findContours(canny, cv.RETR_LIST,
cv.CHAIN_APPROX_NONE)
```

Now, this single line of code can be understood simply.

And essentially, this is equal to cv.findContours. This method takes in the edges, that is, the variable canny. It also takes in a model in which to find the contents. Now, this is either cv.RETR_TREE if you want all the hierarchical contours, RETR_EXTERNAL if you want only the external contours, or RETR_LIST if you want all the contours in the image.

The next method we pass in is the cone to approximation method. For now, we're going to set this to cv.CHAIN_APPROX_NONE.

Essentially, the cv.findContours method looks at the structuring element or the edges found in the image and returns to values, the contours, which is essentially a Python list of all the coordinates of the contours that were found in the image. And hierarchies, which is really out of the scope of this course. But essentially, it refers to the hierarchical representation of contours. So, for example, if you have a rectangle, and inside the rectangle, if you have a

square, and inside of that square, you have a circle. So this hierarchy is essentially the representation that OpenCV uses to find these contours.

The cv.RETR_LIST essentially is a mode in which this findContours method returns and finds the contours. RETR_LIST essentially returns all the quantities found in the image.

The next one we have is the contour approximation method. This is basically how we want to approximate the contour. So CHAIN_APPROX_NONE does nothing. It just returns all of the contracts.

Some people prefer to use the red CHAIN APPROX SIMPLE, which essentially compresses all the quantities returned in the simple ones that make the most sense. For example, if you have a line in an image, if you use a chain approx. None, you will get all the contours and all the coordinates of the points of that line. Chain approx simple essentially takes all of those points of that line and compresses it into the two endpoints only.

Because that makes the most sense, a line is defined by only two endpoints, and we don't want all the points in between. That, in a nutshell, is what this entire function is doing.

So since contours is a list, we can essentially find the number of contours that were found by finding the length of this list. So we can print the length of the list with the print() function.

Add this code:

```
print(f'{len(contours)} contours found!')
```

The return shows 515 contours found!

We can blur the image before finding the edges. Do you remember how to blur images? Yes, the Gaussian Blur function. We can pass in the image as the image src and use the kernel size of 5 by 5.

This will significantly reduce the number of contours found by blurring the image with the five by five kernel size.

Now there is another way of finding the contours. Instead of using the canny edge detector, we can use another function in OpenCV, the threshold.

Simply add the following code instead of the canny code:

```
ret, thresh = cv.threshold(gray, 125, 255, cv.THRESH_BINARY)
```

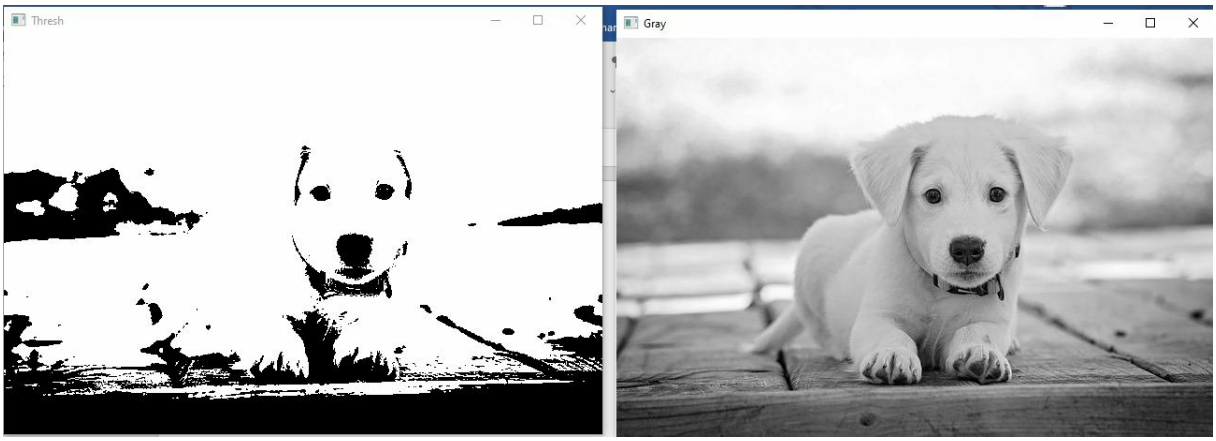
The cv.threshold function we used in this code will take in the gray image, and a threshold value of 125 and a maximum value of 255. Don't worry too much about the threshold function. For now, just know that threshold essentially looks at an image and tries to binarize that image. So if a particular pixel is below 125, if the density of that pixel is below 125, it's going to be set to zero or blank. If it is above 125, it is set to white or two by five. That's all it does. And in the findContours method, we have to replace canny with thresh, and the last part is to specify a threshold and type. That is cv.THRESH_BINARY, binary raising the image.

Save that and run. You can see the contours. 437 contours found!

We can visualize that with the print () function. Now, add the following code to display this:

```
ret, thresh = cv.threshold(gray, 125, 255, cv.THRESH_BINARY)

contours, hierarchies = cv.findContours(thresh, cv.RETR_LIST,
cv.CHAIN_APPROX_NONE)
cv.imshow('Thresh', thresh)

print(f'{len(contours)} contours found!')
```

You will be able to see the image that shows the different contours like this:



And this was the thresholded image you're using 125 as our threshold value and 255 as a maximum value, and we got this thresholded image. And when we tried to find the current use of this image, we got 437 contours.

Now don't worry too much about this thresholding business. We'll discuss this in the advanced section of this book more in-depth. You need to know that thresholding attempts to binarize an image, take an image and convert it into a binary form that is either zero or black or white. Now what's cool in open CV is that you can actually visualize the contours that were found on the image by essentially drawing over the image.

Drawing Contours on Blank Image

So, if you remember, Numpy is the library that we use to create blank screen windows. Let us create a blank screen and draw the contours directly on it to display the contours in our puppy image.

If you have not imported NumPy from the beginning of your script, it is time to do so.

After this, go ahead and create a blank variable and assign it to np.zeros. Add the following code after importing Numpy:

```
blank = np.zeros(img.shape[:2], dtype='uint8')
cv.imshow('Blank', blank)
```

If you run, you will see a blank window displayed. This screen is of the exact dimensions as our original puppy image. So, we want to draw these contours we found on that blank image so that we know what kind of contours that openCV found.

We do that by using the cv.drawContours() method. It takes in an image to draw overfill blank, and it takes in the contours, which has to be a list, which in this case is just the contours list. Take a look at the code:

```
import cv2 as cv
import numpy as np

img = cv.imread("images/puppy.jpg")
```

```
cv.imshow("Puppy", img)
```

```
gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
cv.imshow('Gray', gray)

ret, thresh = cv.threshold(gray, 125, 255, cv.THRESH_BINARY)

contours, hierarchies = cv.findContours(thresh, cv.RETR_LIST,
cv.CHAIN_APPROX_NONE)
cv.imshow('Thresh', thresh)
```

```
blank = np.zeros(img.shape, dtype='uint8')
cv.imshow('Blank', blank)

cv.drawContours(blank, contours, -1, (0,0,255), 2)

cv.imshow("Drawn Contours", blank)

# canny = cv.Canny(img, 125, 175)
# cv.imshow("Canny Edges", canny)
```

```
print(f'{len(contours)} contours found!')

cv.waitKey(0)
```

As you can see in the code above, the cv.drawsContours() method takes an account to the index, which is basically how many contours you want in the image. Since we want all of them since, we specified it as -1, give it a color, 0,0,255. And we can give it a thickness of 2.

As you can see, the Contours Drawn window is different from the contours image. OpenCV attempts to find all the contours and edges and draw them all on the blank screen we provided.

We can play around with this. For example, we can increase the thickness of the lines so that we have a crisper view. Go back to the script, set the thickness argument to 1, and run.

You can also see the difference between this method and the Canny Edge Detector tool.

Generally, I recommend that you use the Canny method first and then try to find the contours using that, rather than try to threshold the image and then find the contours on that. As we will discuss in the advanced section, this type of simple thresholding has its disadvantages. It's not the most ideal, but in some cases, in most cases, it is the most favored kind of thresholding

because it's the simplest and it does the job pretty well.

For this chapter, we talked about how to identify contours in OpenCV. But in two methods, first trying to find the edge cascades of the image using the canny edge detector, and try to find the contours using that and also trying to binarize that image using the cv.threshold and finding the contours on that.

ADVANCED SECTION

Chapter 7 - Color Spaces

We are now at the advanced section of this course, where we are going to discuss the advanced concepts in OpenCV.

So what we're going to discuss in this chapter is how to switch between color spaces in urgency. Our color spaces are basically a space of colors, a system of representing an array of pixel colors.

RGB is a color space, and grayscale is also a color space. We also have other color spaces like HSV, lamb, and many more.

BGR to Grayscale

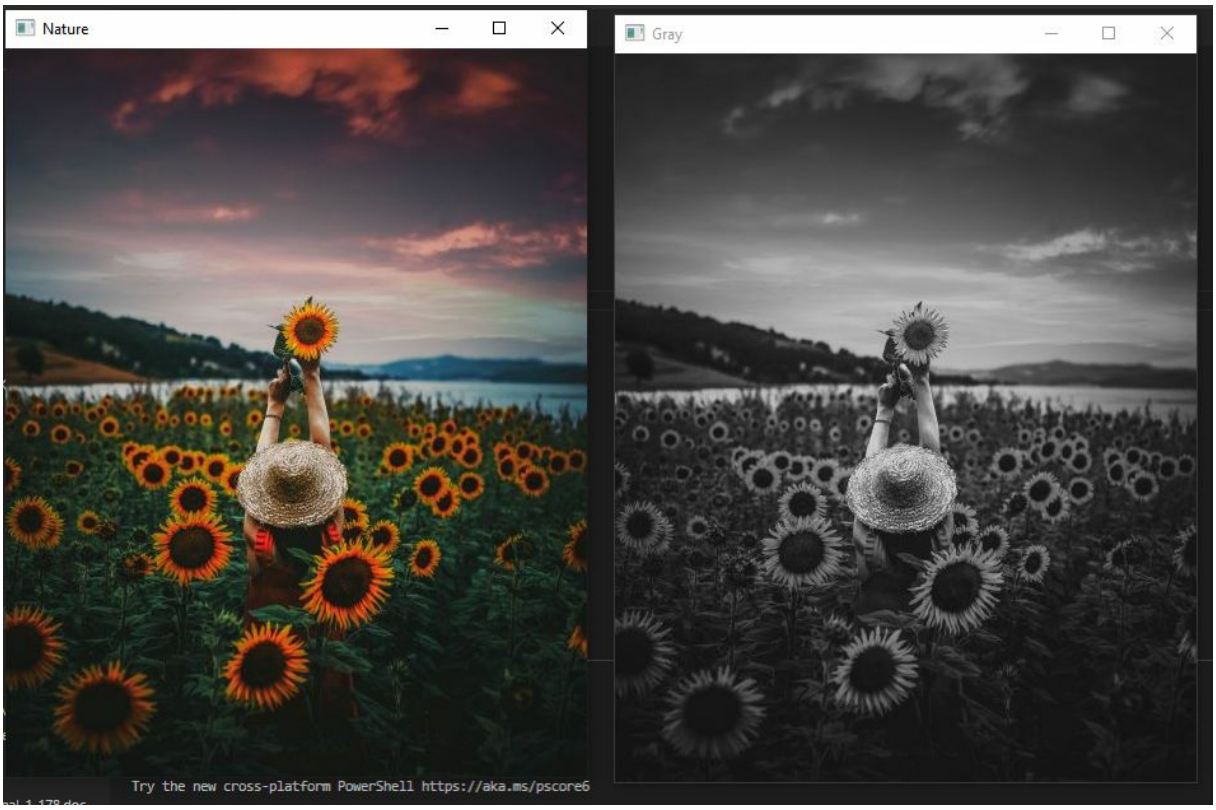
So let's start off by trying to convert an image to grayscale. When you read an image, OpenCV, by default, reads it in the BGR color space.

Now, we will convert that to grayscale.

```
import cv2 as cv
from cv2 import cvtColor

img = cv.imread ('images/colours.jpg')

cv.imshow('Nature', img)

gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
cv.imshow('Gray', gray)
```

```
cv.waitKey(0)
```

Now, look at that code. We are using the cvtColor() method to do the conversion. We pass in the image and specify a color code, which is cv.COLOR_BGR2GRAY, since we're converting from a BGR image format to a grayscale format. We display this image with the cv.imshow() function.



Now you can see the grayscale version of this BGR image.

Grayscale images show you the distribution of pixel intensities at particular locations of your image.

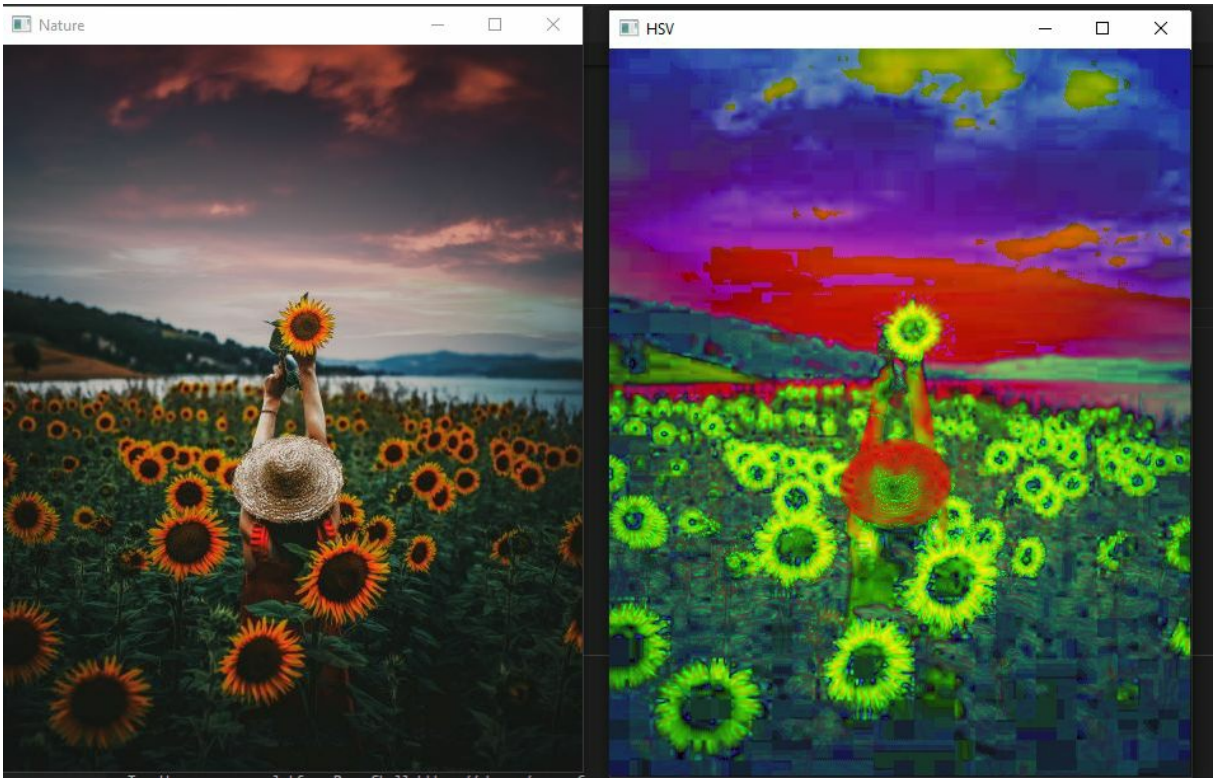
BGR to HSV

HSV is short for Hue Saturation Value. It is based on how humans think and perceives color.

COnverting to HSV is pretty easy. Just the same steps as with the Grayscale conversion, which we have already locked down. In this case, we use the cv.COLOR_BGR2HSV method. Let's look at the code in action:

```
import cv2 as cv
from cv2 import cvtColor

img = cv.imread ('images/colours.jpg')

cv.imshow('Nature', img)
```

```
#to HSV
hsv = cv.cvtColor(img, cv.COLOR_BGR2HSV)
cv.imshow('HSV', hsv)

cv.waitKey(0)
```



You can see the HSV version of this BGR image. As you can see, there is a lot of green in the image, and the skies are reddish. Now we also have another kind of color space. And that is called the LAB color space.

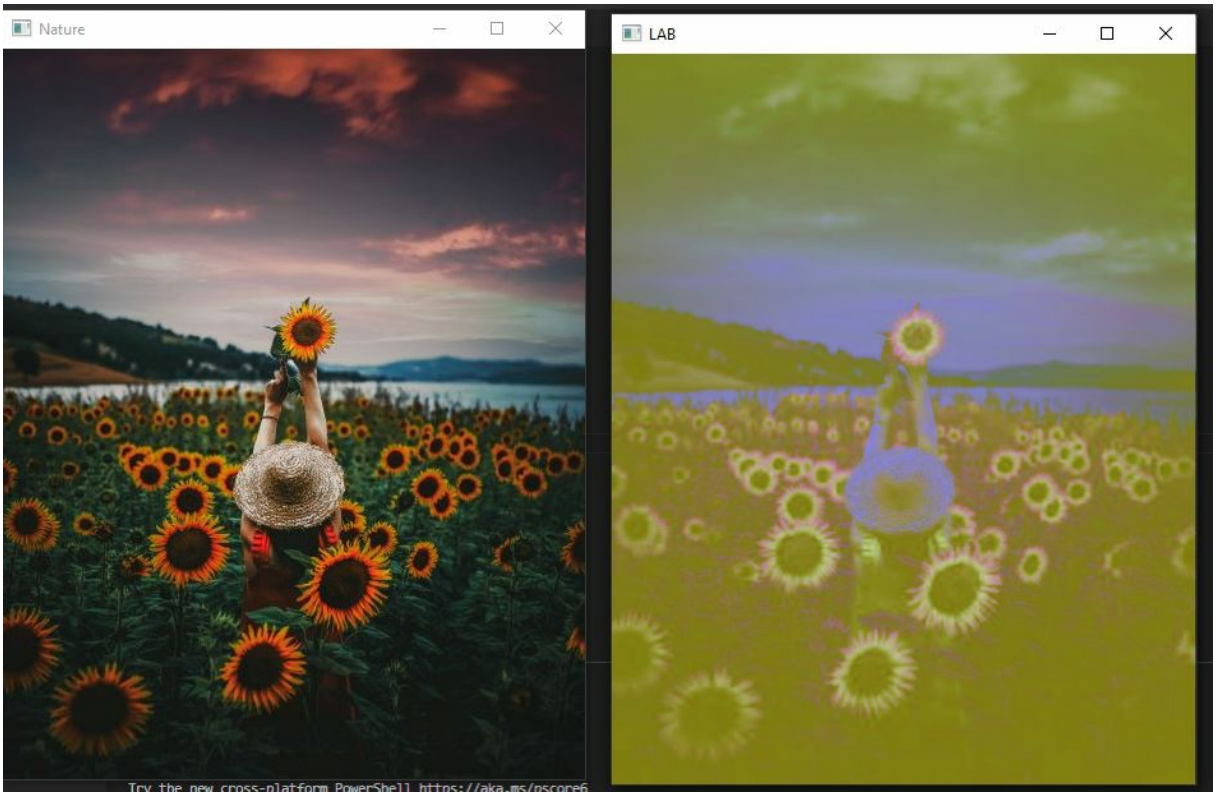
BGR to LAB

This color space is sometimes represented as L times A times B, but feel free to use whatever you want. Converting to LAB is pretty much similar to what we have been seeing so far in the script:

```
import cv2 as cv
from cv2 import cvtColor

img = cv.imread ('images/colours.jpg')
```

```
cv.imshow('Nature', img)

#to l*a*b
lab = cv.cvtColor(img, cv.COLOR_BGR2LAB)
cv.imshow('LAB', lab)
```

```
cv.waitKey(0)
```



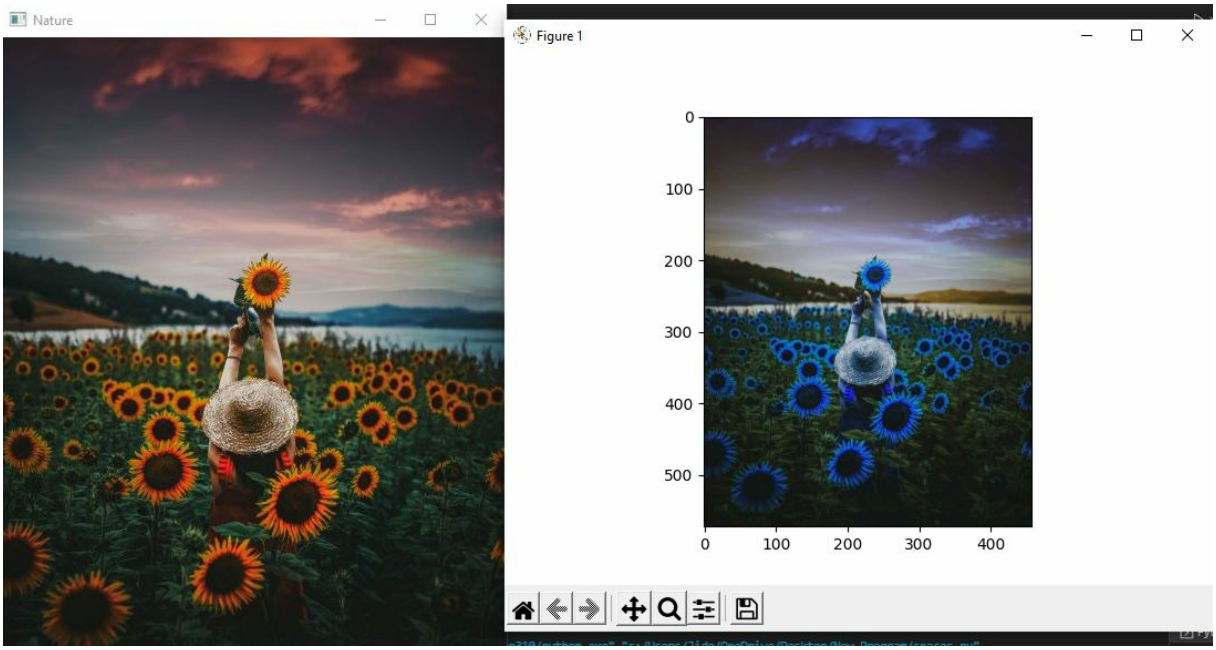
You have seen the LAB version of our BGR image. It looks like a washed-down version of the BGR image. That's the LAB format. It is more tuned to how humans perceive color. At the onset of this tutorial, I mentioned that OpenCV reads images in a BGR format with blue, green, and red. And that's not the current system that we use to represent colors outside of OpenCV. We use the RGB format outside of open CV, which is the inverse of the BGR format.

Now, try to display this IMG image in a Python library that's not OpenCV. You're probably going to see an inversion of colors. And we can do that too. So you can understand what I am trying to explain, try importing a new

Python library and read the same image we have displayed in OpenCV. Let's use the library called matplotlib.pyplot and display the same image variable. Follow this code:

```
import cv2 as cv
import matplotlib.pyplot as plt

img = cv.imread ('images/colours.jpg')

cv.imshow('Nature', img)

plt.imshow(img)
plt.show()
```

If your script shows an error like Module Not Found, you may need to install the Matplotlib module with your cmd or Terminal with the code line "pip install matplotlib."



Now you can see the image we get. Now, if you compare it with the image that openCV read, this is entirely different. These two are entirely different images. And the reason for this is because the first image on the left is a BGR

image, and openCV displays BGR images.

But now, if you tried to take this BGR image and display it in matplotlib, for instance, matplotlib has no idea that this image is a BGR image and displays it as if it were an RGB image. So that's why you see an inversion of color.

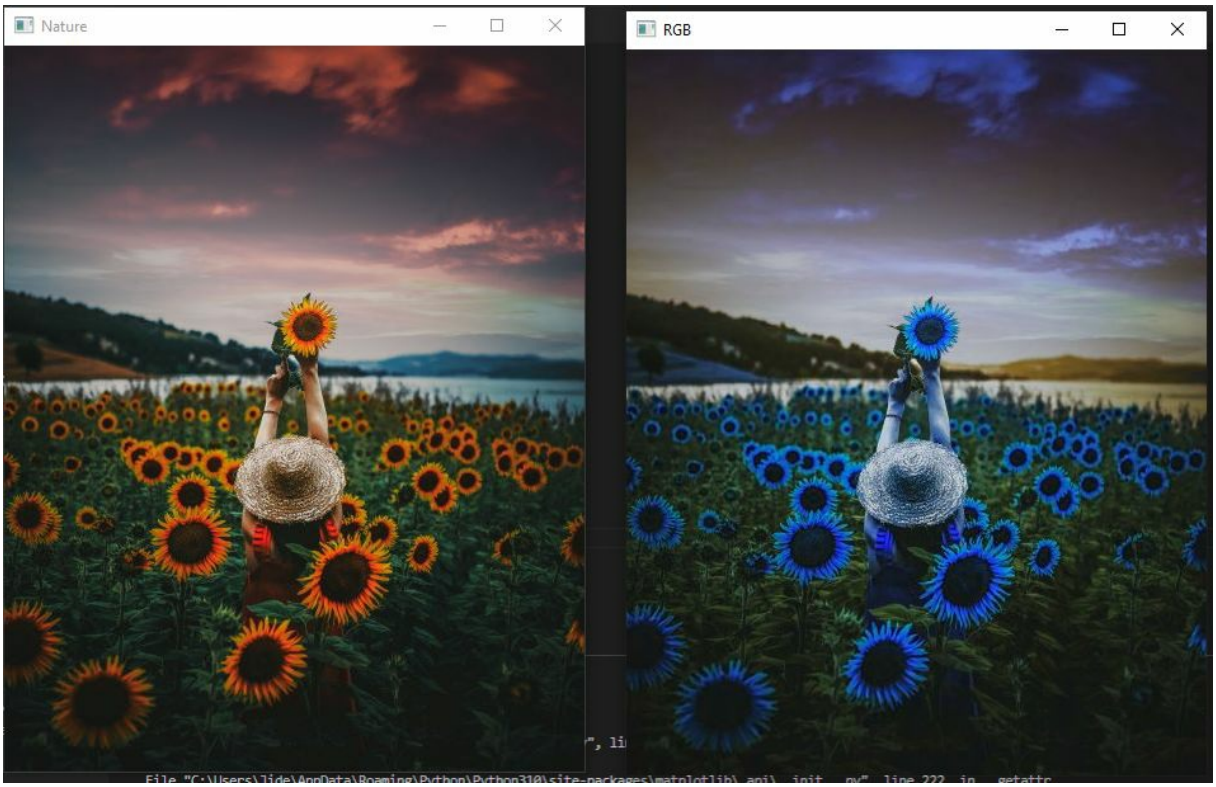
So where there's red in the first one, you see a blue, where there's blue over here, you see a red, and there are ways to convert this from BGR to RGB. And that is by using OpenCV itself.

BGR to RGB

Converting to RGB is as simple as converting to the other color spaces we have discussed. Follow the code:

```
import cv2 as cv
from cv2 import cvtColor

img = cv.imread ('images/colours.jpg')

cv.imshow('Nature', img)

rgb = cv.cvtColor(img, cv.COLOR_BGR2RGB)
cv.imshow('RGB', rgb)

cv.waitKey(0)
```

What we have done is to make OpenCV display the image the same way the matplotlib module would.



Now again, you see an inversion of colors, but this time in OpenCV because now you provided openCV and RGB images. The sweet thing is that if you passed in the RGB image you have converted with OpenCV in the matplotlib show function, it would work fine to show the real colors of the image. That is because matplotlib, by default, is RGB. So that's why it displays the right image when you pass in an RGB image. So just keep this in mind when you're working with multiple libraries, including OpenCV and matplotlib, for instance, because do keep in mind the inversion of colors that tends to take place between these two libraries.

We've essentially converted the BGR to grayscale, BGR, HSV BGR to RGB, and BGR to RGB. We can do the inverse of that. We can convert a grayscale image to BGR, convert an HSV to BGR, convert a LAB to BGR, and RGB to BGR, and so on.

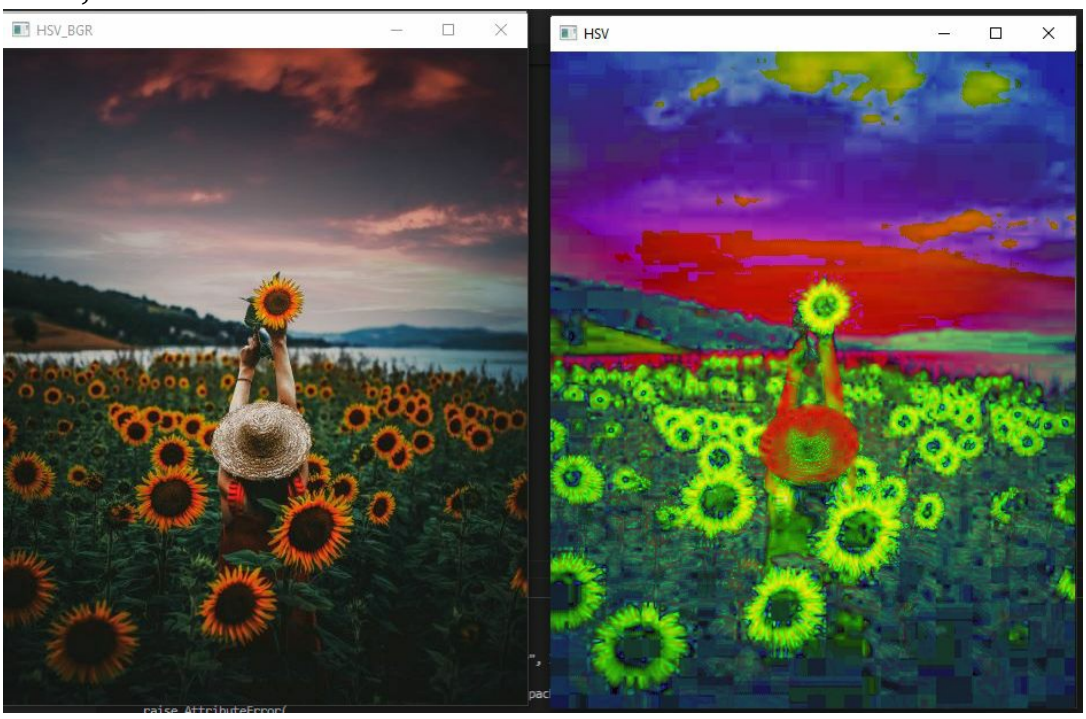
But here's one of the downsides. You cannot convert grayscale images to HSV directly. If you want to do that, you have to convert the grayscale to BGR. And then from BGR to HSV.

HSV to BGR

This is almost similar in the code to all other conversions because we want to essentially use the same cv.cvtColor() function. However, in this case, we will change the src image and pass the HSV instead. Look at the code below:

```
hsv_bgr = cv.cvtColor(hsv, cv.COLOR_HSV2BGR)
cv.imshow('HSV_BGR', hsv_bgr)
```

Now, this is the result:



The HSV image is on the right, and the HSV is converted to the BGR image on the left. This is the same way to convert all the others.

With LAB to BGR, simply replace HSV with LAB in the code, and you will get the converted images.

For instance, if you want to convert from grayscale to LAB, note that there is no direct method. What you could do is convert that grayscale to BGR. And then from BGR to, and maybe that's possible. By directly. I don't think there was a way to do that. If OpenCV could come up with a feature that could achieve that, it would be good, but it's not going to hurt you to write extra

lines of code, at least two or three lines of code extra, moderately hard. In the next chapter, we will be talking about how to split and merge color channels in OpenCV.

Chapter 8 - Color Channels

In this chapter, we're going to be talking about how to split and merge color channels in OpenCV.

Now, a color image basically consists of multiple channels, red, green, and blue. All the images you see around you, all the BGR or the RGB images, are merged into these three color channels. Now openCV allows you to split an image into its respective color channels.

So you can take a BGR image and split it into blue, green, and red components. So that's what we're going to be doing in this chapter. We're going to be taking this image of the flower fields that we had seen in the previous chapter, and we're going to split that into three color channels.

Splitting Channels

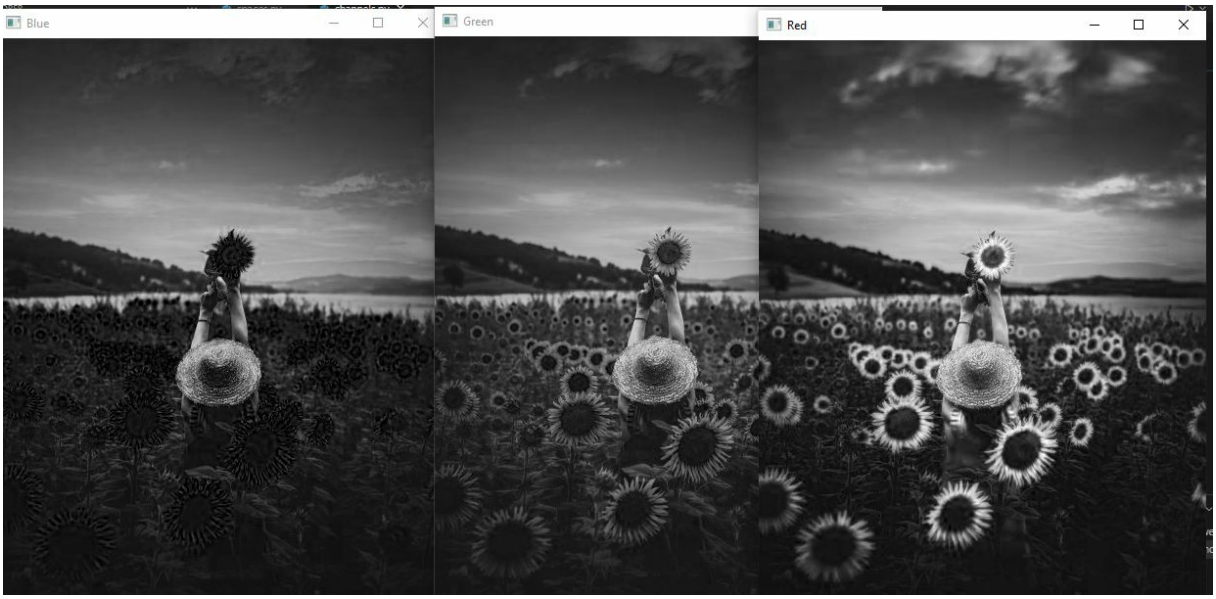
We want to pick our Nature image and split it into the 3 color channels. First, we will split the channels in an RGB image. We will use the function called cv.split().

So the way we do that is by saying b comma g comma r, which stands for the respective color channels, and set this equal to cv.split and pass in the image. So the cv.split basically splits the image into blue, green, and red.

```
import cv2 as cv

img = cv.imread ('images/colours.jpg')

cv.imshow('Nature', img)

b,g,r = cv.split(img, )

cv.imshow('Blue', b)
cv.imshow('Green', g)
cv.imshow('Red', r)

print(img.shape)
print(b.shape)
print(g.shape)
print(r.shape)
```

```
cv.waitKey(0)
```

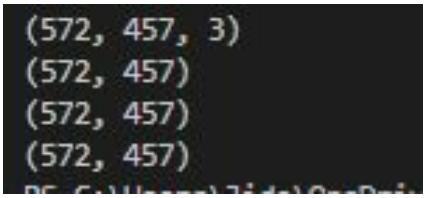
We displayed this image with the cv.imshow() function, passing blue as b. Green image, passing in g and the Red part as r. We also visualized the shapes of the images with the print() method for each. We're printing the shapes and dimensions of the image and the blue, green, and red, and we're also displaying these images.



And these are the images that you get back. You can see the blue, green, and red images. Now, these are depicted and displayed as grayscale images that show the distribution of pixel intensities. Regions where it's lighter showed a far more concentration of those pixel values, and regions where it's darker represented a little or even no pixels in that region.

So take a look at the blue and pick the blue channel first. And if you can, if you compare it with the original image, you will see that the sky is kind of almost white. This shows you that there is a high concentration of blue in the sky and not so much in the grass. Let's take a look at the green. And there is a fairly even distribution of pixel intensities between the grass, the trees, and some parts of the sky. And take a look at the red color channel. And you can see that parts of the trees that are red are whiter, and the grass in the sky is not that white in this red image. So this means that there is not much red color in those regions.

Now, if you look at the Python IDLE terminal, you will see the shapes of the image. Something like this:



Now, the first line there stands for the original image, the BGR image. The additional elements in the tuple here represent the number of color channels. 3 represents three color channels blue, green, and red. Now, if we proceeded to display the shapes of the BGR components one after another in the subsequent lines, we don't see a 3 in the tuple. That's because the shape of that component is 1. It's not mentioned there, but it is 1.

When you try to display this image using cv.imshow(), it displays it as a grayscale image because grayscale images have a shape of 1.

We can try to merge these color channels together. The way we do that is by seeing the merged image.

Merging Color Channels

We will use the OpenCV function called cv.merge(). Follow the codde below to see how it is done when you add the lines to your script.

```
merge = cv.merge ([b,g,r])
cv.imshow('Merged Image', merge)
```

This code will bring back the merged image by basically merging the three individual color channels, red, green, and blue.

Now there is an additional way of looking at the actual color there is in that channel. So instead of showing you grayscale images, it shows you the actual color involved.

So for the blue image, you get the blue color channel for the red channel, and you get the red color for that channel.

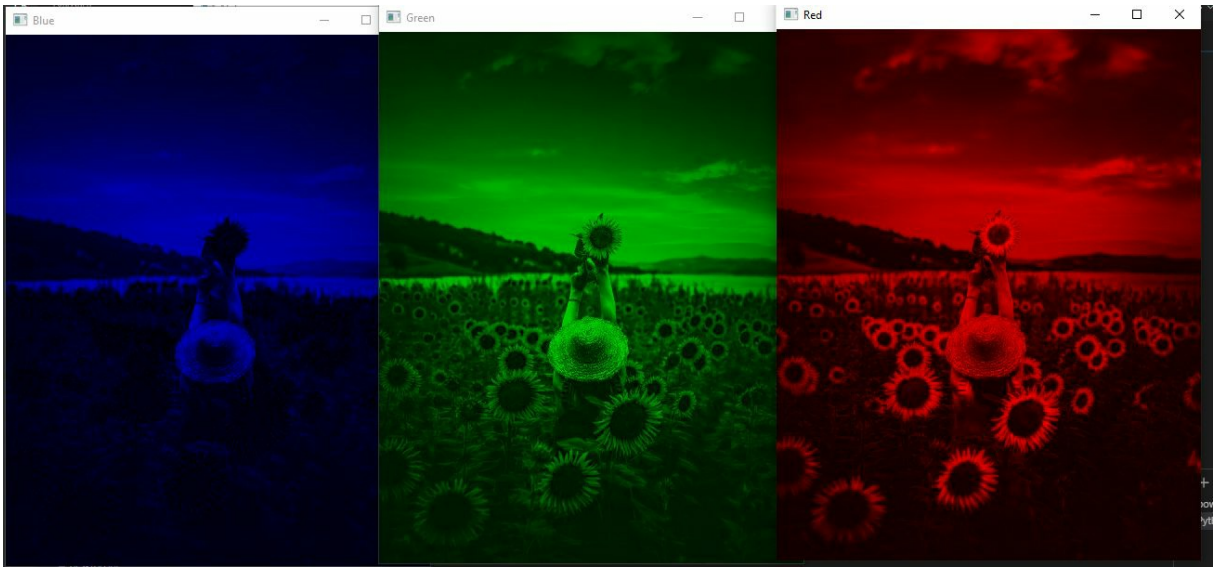
Reconstructing Color Channels

And the way we do that is we have to reconstruct the image. The shapes of these images are grayscale images. But what we can do is create a blank image, a blank image using NumPy. Look at the script now:

```
import cv2 as cv
import numpy as np

img = cv.imread ('images/colours.jpg')

cv.imshow('Nature', img)

blank = np.zeros (img.shape[:2], dtype='uint8')

b,g,r = cv.split(img)

blue = cv.merge([b,blank,blank])
green = cv.merge([blank,g,blank])
red = cv.merge([blank,blank,r])

cv.imshow('Blue', blue)
cv.imshow('Green', green)
cv.imshow('Red', red)

print(img.shape)
print(b.shape)
print(g.shape)
print(r.shape)

merge = cv.merge ([b,g,r])
cv.imshow('Merged Image', merge)

cv.waitKey(0)
```

I've done this blank image basically consisting of the height and the width, not necessarily the number of color channels in the image. So by essentially merging the blue image in its respective compartment, so blue, green, and

red, we are setting the green and the red components to black and only displaying the blue channel. And we're doing the same thing for the green by setting the blue and the red components to black. And the same thing for red is by setting the blue and the green components to black, and now you get the color in its respective color channels.



Take a look at this. You are now able to visualize the distribution much better.

Here you can see later lineup portions represent a high distribution. Lighter portions here represent the high distribution of red, and higher and wider regions represent a high distribution of green. So essentially, if you take these three images of these colors and merge them, you get back the merged image. That's the merged image.

For this chapter, we have discussed how to split an image into three respective color channels, reconstruct the image to display the actual color involved in that channel and merge those color channels back into its original image. In the next chapter, we'll be talking about how to smooth and blur an image using various blurring techniques.

Chapter 10 – The Magic of Blurring

In this chapter, we're going to address the concepts of smoothing and blurring in OpenCV.

Now, in the previous chapters, I mentioned that we generally smooth an image when it tends to have a lot of noise, and noise that's caused from camera sensors is basically problems with lighting when the image was taken. And we can essentially smooth out the image or reduce some of the noise by applying some blurring method.

Previously, we discussed the Gaussian Blur method, which is one of the most popular blurring methods. But generally, you're going to see that Gaussian Blur won't suit some of your purposes. And that's why there are many blurring techniques that we have. And that's what we're going to address in this chapter.

Concepts of Blurring in OpenCV

Before we actually do that, I want to address a couple of concepts. Let's go to an image and discuss what goes on when you try to apply the blur.

Kernel

The first thing that we need to define is something called a kernel or sometimes called a window. And that is essentially this window that you draw over an image. It is essentially a window or a series of rows and columns of pixels that you can draw over a specific portion of an image. Imagine an image as an excel spreadsheet with each pixel as a cell. A Kernel is a range, like in Excel, say A1:C4, of the cells.

So essentially, this window has a size. This size is called a kernel size. Now kernel size is the number of rows and the number of columns in the range you want to work on.

Now, essentially, what happens here is that we have multiple methods to apply some blue. So essentially, blur is applied to the middle pixel as a result of the pixels around it, also called the surrounding pixels. Something happens here as a result of the pixels around the surrounding pixels.

A kernel is an array of numerical coefficients with a fixed size and an anchor

point, usually in the middle of the array. This kernel, also called a structuring element, is just a binary matrix, a matrix made up of 0s and 1s. The way these 0s and 1s are put together will determine the group of pixels over which the minimum (for erode) or maximum (for dilate) is taken.

The erode function will move the kernel over the original image, and all of the original image's pixel values that "fall under" a 1 on the kernel will be used to figure out the minimum value.

Averaging

So with that in mind, let's go back and discuss the first method of blurring, which is averaging.

The image is averaged by using a convolution operation with a normalized box filter on the image. During a convolution operation, the filter or kernel is moved across the image, and the average of all the pixels under the kernel is found. This average is then used to replace the image's central element.

Note: The size of the kernel affects how smooth an image is. If the Kernel size is big, the small parts of the image are taken away. But if the size of the kernel is too small, it can't get rid of the noise.

So essentially, averaging is done by defining a kernel window over a specific portion of an image. This window will essentially compute the pixel intensity of the middle pixel, that is, the true center, as the average of the surrounding pixel intensities.

And we essentially use that result as the pixel intensity for the middle value or the actual center. And this process happens throughout the image.

We're going to blur an image now.

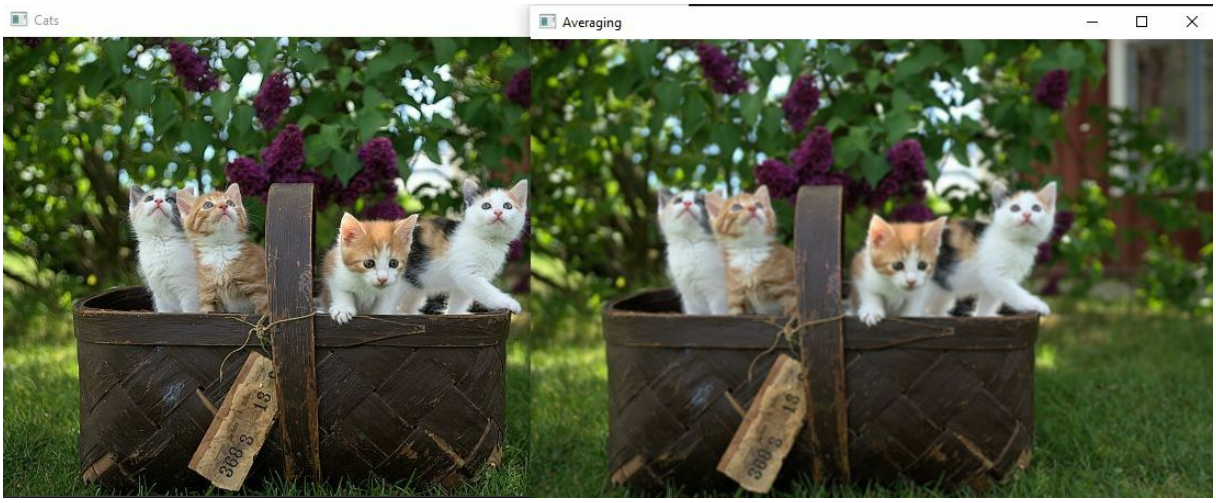
Blurring or Averaging an Image

We have seen this code before. However, you can try it again. Use the following code:

```
import cv2 as cv

img = cv.imread ('images/cats.jpg')
```

```
cv.imshow('Cats', img)

average = cv.blur(img, (3,3))
cv.imshow('Averaging', average)

cv.waitKey(0)
```

And this is the average blur that's applied.



So what the algorithm did in the background was essentially define a candle window of a specified size three by three. And it computed the center value for a pixel using the average of all the surrounding pixel intensities. And the result is that we get a blurred image.

So the higher the kernel size we specified, the more blur there is going to be in the image.

If you increase the ksize value to 7, 7, you get an image with way more blur. Let's move on to the next method, the Gaussian Blur.

Gaussian Blur

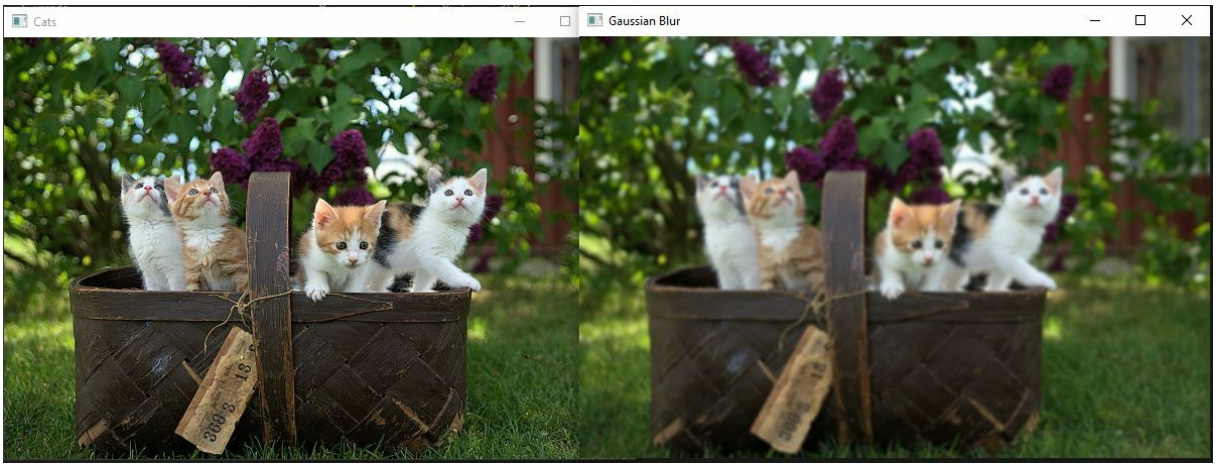
So Gaussian does the same thing as averaging, except that instead of computing the average of all of this running pixel intensity, each running pixel is given a particular weight. And essentially, the average of the products of those weights gives you the value for the true center.

Using this method, you tend to get less blurring than the averaging method.

But the Gaussian Blur is more natural as compared to averaging.

```
import cv2 as cv

img = cv.imread ('images/cats.jpg')

cv.imshow('Cats', img)

# average = cv.blur(img, (3,3))
# cv.imshow('Averaging', average)

gauss = cv.GaussianBlur(img, (7,7), 0)
cv.imshow ("Gaussian Blur", gauss)

cv.waitKey(0)
```

As you can see, we use the cv in the new line of code.GaussianBlur () function. And this function will take in the source image, img, and a kernel size of 7 by 7, just to compare with the averaging you tried with the ksize 7 by 7. And another parameter that we need to specify is sigma x, or the standard deviation in the x-direction, which for now, is just going to be set as zero.



If you can bear with this, you see that both use the same code size, but this is less blurred than the average method. And the reason for this is that a certain weight value was added when computing the blur. Okay, so let's move on to the next method. And that is a median blur.

Median Blur

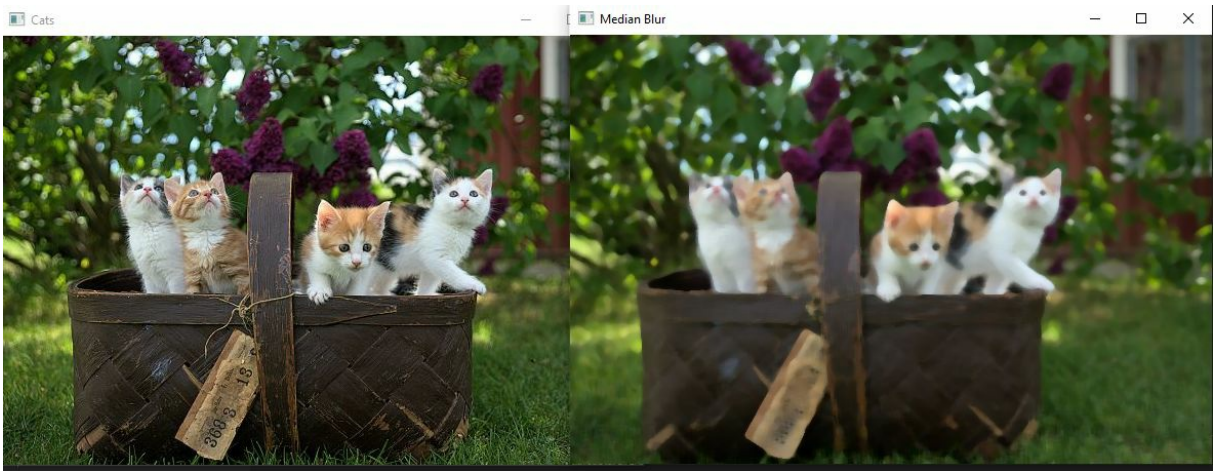
So let's go back to our image. And medium blurring is the same thing as averaging, except that instead of finding the average of the surrounding pixels, it finds the median of the surrounding pixels.

Generally, medium blurring tends to be more effective in reducing noise in an image than averaging and even Gaussian Blur. And it's pretty good at removing some salt and pepper noise that may exist in the image.

In general, people tend to use this image blurring method in advanced computer vision projects that tend to depend on the reduction of a substantial amount of noise. Check out the code to run a Median Blur:

```
import cv2 as cv

img = cv.imread ('images/cats.jpg')

cv.imshow('Cats', img)

# average = cv.blur(img, (3,3))
# cv.imshow('Averaging', average)

# gauss = cv.GaussianBlur(img, (7,7), 0)
#cv.imshow ("Gaussian Blur", gauss)

median = cv.medianBlur(img, 7)
cv.imshow ('Median Blur', median)

cv.waitKey(0)
```

And the way we apply the blur is by saying, let's call this median and set the use the cv.medianBlur () method, we pass in the source image, and this kernel size will not be a tuple of 7 by 7, but instead, just an integer to 7. And the reason for this is because open CV automatically assumes that this kernel size will be a 7 by 7, just based on this integer.



And let's compare it with that. So I set that to seven. And comparing it with Gaussian Blur and averaging blur, you tend to look at this. And you can make up some differences between the two images. So it's as if this was your painting, and it was still drawing. And you take something and smudge over the image, and you get something like this.

Generally, medium blurring is not meant for large kernel sizes like seven or even five in some cases, and it's more effective in reducing some of the noise in the image. For example, you will need to use a small integer like 3 instead of 7 to get the best out of the Median Blur. Try to change the codes all to 3 by 3 in your script. And now, let's have a comparison between the three. Compared with the other two, you can see that there is less blurring when Gaussian when you can sort of make out the differences between the two. The differences are very subtle, but there are a couple of differences between the two.

Finally, the last method we're going to discuss is bilateral blurring.

Bilateral Blurring

Now bilateral bearing is the most effective and sometimes used in a lot of advanced computer vision projects, essentially because of how it blurs.

Now traditional blurring methods blur the image without looking at whether you're reducing edges in the image or not. Bilateral blurring applies blurring but retains the edges in the image.

So you have a blurred image, but you also get to retain the edges. Let’s look at the code for this:

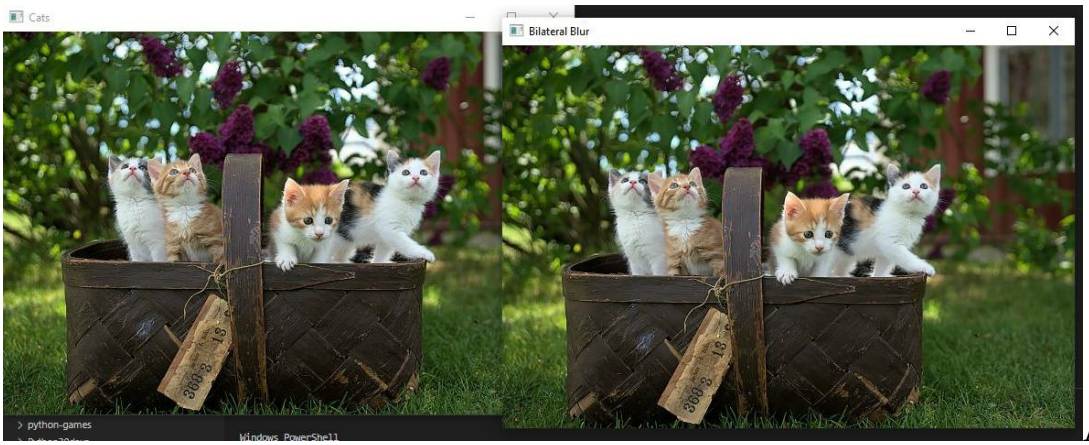
```
import cv2 as cv

img = cv.imread ('images/cats.jpg')

cv.imshow('Cats', img)

# average = cv.blur(img, (3,3))
# cv.imshow('Averaging', average)

# gauss = cv.GaussianBlur(img, (7,7), 0)
#cv.imshow ("Gaussian Blur", gauss)

# median = cv.medianBlur(img, 7)
# cv.imshow ('Median Blur', median)

bilateral = cv.bilateralFilter(img, 5, 15, 15)
cv.imshow("Bilateral Blur", bilateral)

cv.waitKey(0)
```

We will be using the cv.bilateralFilter () function. When we pass in the image, we give it a diameter of the pixel neighborhood. Now notice this isn't a kernel size but, in fact, a diameter. So let's set the diameter to 5 for now, give it a sigma color, which is the color sigma color. A larger value for this color sigma means that there are more colors in the ‘neighborhood’ that will be considered when the blur is computed. And so we set that to 15.

And sigma space is your space sigma. Larger values of this space sigma mean that pixels further out from the central pixel will influence the blurring calculation. And so we set this to 15 too.

Sigma spacing in bilateral filtering takes the value for the central pixel or the true center computed. By giving a larger value for the Sigma space, you indicate whether you want pixels from the extreme ends of the image or a little far from the anchor point of diameter to influence this calculation.

So if you give a huge number, then probably a pixel at the end position or the extreme position of the image might influence the computation of this blurring pixel value.



And this is your bilateral image. So let's compare with all the previous ones that we had. Compared with this. Much better compared with the averaging method. You see that this is way much better. Let's compare with the median. The edges are slightly blurred. If you compare it with the original image, they look the same thing, as if there's no blur applied.

What if we increase the number of the parameters we inputted in the code? Can you try that? Play around with the values. And now you can make our generic that this is starting to look like a median blur.

You would see that the larger the figures, the more the blur until it's starting to show you that this is more looking like a smudged painting version of the original image.

So definitely keep that in mind when you are trying to apply blurring to the image, especially with the bilateral and median lowering, because higher values of this basic mouth or bilateral or the kernel size for medium glowing, and you tend to end up with a washed-up smudged version of this image.

So definitely keep that in mind. But that kind of summarizes whatever we've done in this chapter. We discussed averaging, Gaussian, median and bilateral blurring. So in the next chapter, we'll be talking about bitwise operators in OpenCV.

Chapter 11 – Bitwise Operations

In this chapter, we're going to be talking about bitwise operators in OpenCV.

Now, there are four basic bitwise operators and or XOR and not. Suppose you've ever taken an introductory CS course. In that case, you will probably find these terms, familiar bitwise operators. As we'll see in the next chapter, they are used a lot in image processing, especially when working with masks.

In this section, we'll look at the AND, OR, XOR, and NOT bitwise operations. Even though they are very simple and low-level, these four operations are the most important parts of image processing, especially when working with masks in the next part of this series.

Bitwise operations work in binary and are shown as grayscale pictures. If a pixel's value is zero, it is turned "off," and if it has a value greater than zero, it is turned "on." Let's move on and start writing code.

Create A Rectangle and Circle

We need NumPy for this operation. Because we are going to create a blank variable, and we can give it a datatype like we use the np. zeros() method. Look at the code:

```
import cv2 as cv
import numpy as np

blank = np.zeros((400, 400), dtype="uint8")

rectangle = cv.rectangle(blank.copy(), (30,30), (370,370), 255, -1)
circle = cv.circle(blank.copy(), (200,200), 200, 255, -1)
cv.imshow('Rectangle', rectangle)
cv.imshow('Circle', circle)
```

```
cv.waitKey(0)
```

So, we have used NumPy to draw a blank image on which we will draw a rectangle and a circle.

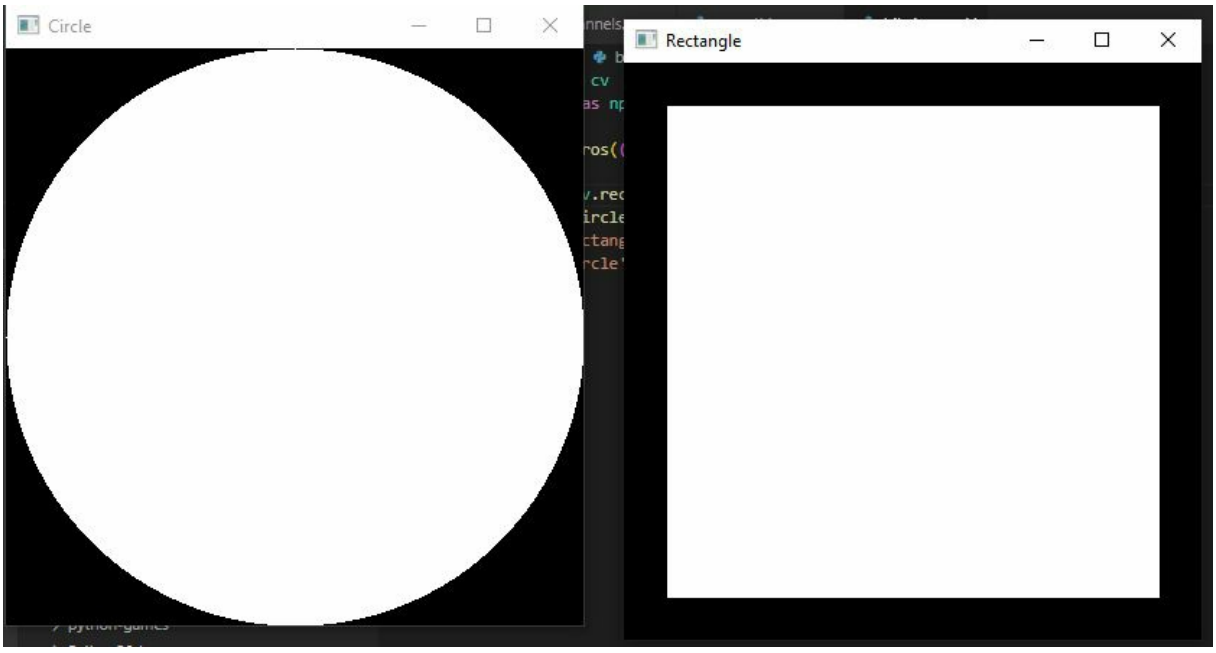
Let me try to explain the code:

I used the cv.rectangle() function you have learned earlier and the cv.circle() function you have also seen.

However, for both, we used the blank.copy() method. And we pass in the starting point, giving it a margin of around 30 pixels on either side. So we start from 30, 30. And we can go all the way across to 370, 370.

The function requires that we give it a color code. Since this is not a color image but rather a binary image, we can just give it one parameter, so 255. White, and give it a thickness of negative one (-1) because we want to fill this image.

We created another circle variable and did almost the same thing. We give it a center, so 200 by 200 and a radius of 200. We used the color setting of 5, and let's fill in the circle with -1.



So we have two images that we're going to work with this image of the rectangle and this image of a circle.

Bitwise AND

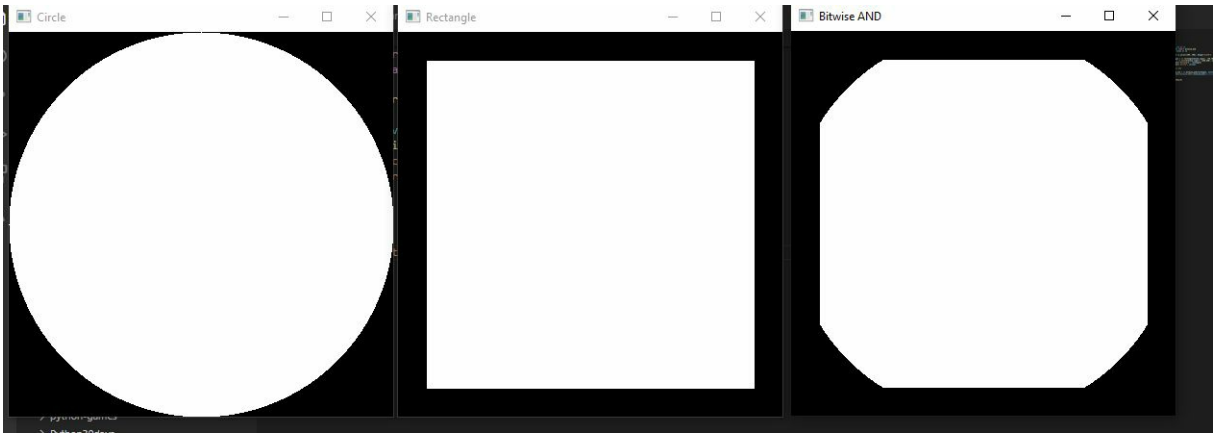
So let's start with the first basic bitwise operator, and that is bitwise AND.

Before discussing bitwise AND is, let me show you what it does.

Here is the code to run a bitwise_and() function:

```
import cv2 as cv
from cv2 import bitwise_and
import numpy as np

blank = np.zeros((400, 400), dtype="uint8")

rectangle = cv.rectangle(blank.copy(), (30,30), (370,370), 255, -1)
circle = cv.circle(blank.copy(), (200,200), 200, 255, -1)
cv.imshow('Rectangle', rectangle)
cv.imshow('Circle', circle)

#bitwise_AND

bitwise_and = cv.bitwise_and(rectangle, circle)
cv.imshow("Bitwise AND", bitwise_and)

cv.waitKey(0)
```

So essentially, I passed in two source images that are these two images, rectangle and circle, into the bitwise_and () function.



So essentially, what bitwise AND did was it took the two images, placed them on top of each other, and returned the intersection, as you can see in the screenshot above.

And you can make out when you take this image, put it over this image, you

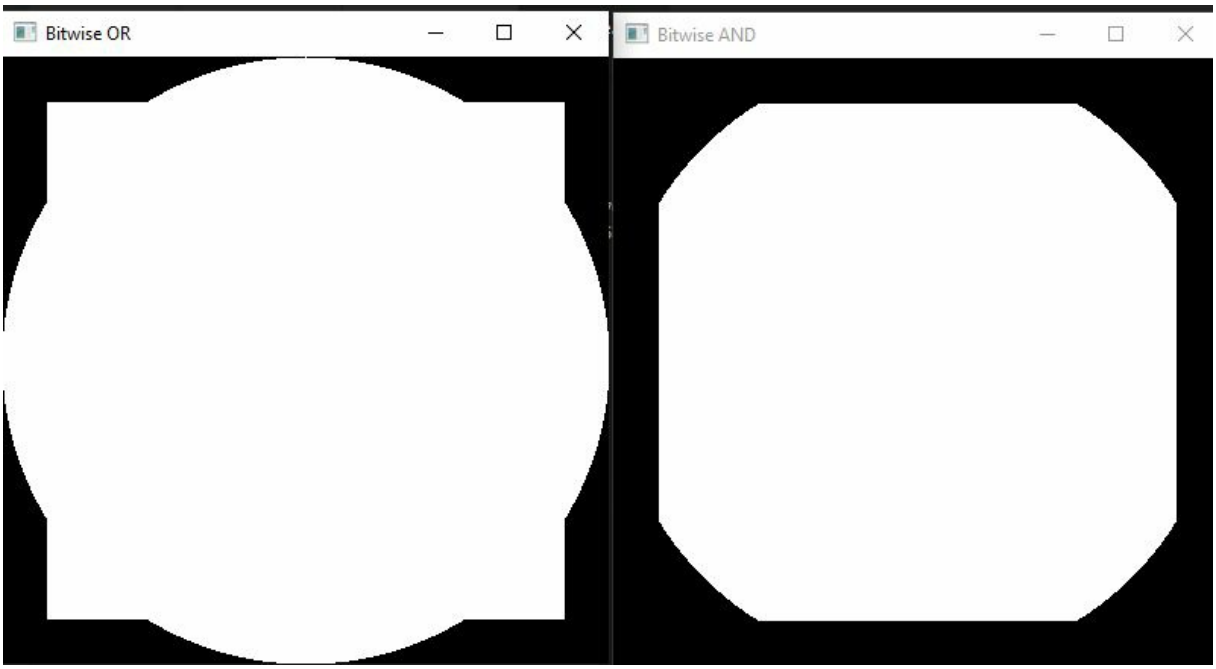
have some triangles that are common to both of these images. And so those are set to black, while the common regions are returned.

Bitwise OR

Bitwise OR simply returns both the intersecting as well as the non intersecting regions.

Here is the code:

```
import cv2 as cv
from cv2 import bitwise_and
import numpy as np

blank = np.zeros((400, 400), dtype="uint8")

rectangle = cv.rectangle(blank.copy(), (30,30), (370,370), 255, -1)
circle = cv.circle(blank.copy(), (200,200), 200, 255, -1)
cv.imshow('Rectangle', rectangle)
cv.imshow('Circle', circle)

#bitwise_AND

bitwise_and = cv.bitwise_and(rectangle, circle)
cv.imshow("Bitwise AND", bitwise_and)

#bitwise_OR
bitwise_or = cv.bitwise_or (rectangle, circle)
cv.imshow("Bitwise OR", bitwise_or)

cv.waitKey(0)
```



It's bitwise OR returns this funky looking this funky looking shape. It essentially took these two images, put them over each other from the common regions, and found regions that are not common to both of these images and superimposed them. So, you can just put them together and find the resulting shape, and this is what you get.

Bitwise XOR

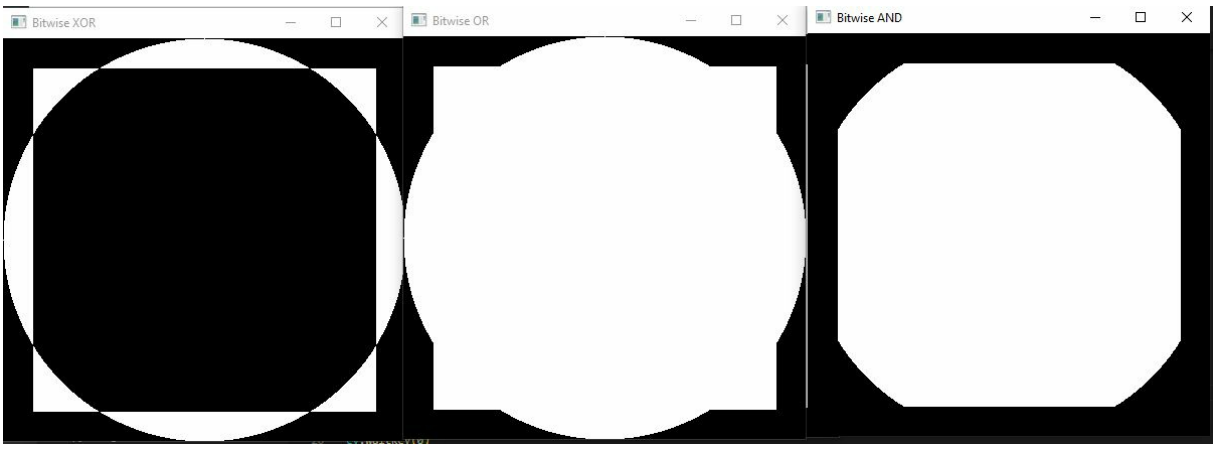
The next is bitwise XOR, which is good for returning the non intersecting regions. So XOR only finds the non intersecting regions.

Let’s look at the code:

```
import cv2 as cv
from cv2 import bitwise_and
import numpy as np

blank = np.zeros((400, 400), dtype="uint8")

rectangle = cv.rectangle(blank.copy(), (30,30), (370,370), 255, -1)
circle = cv.circle(blank.copy(), (200,200), 200, 255, -1)
cv.imshow('Rectangle', rectangle)
cv.imshow('Circle', circle)

#bitwise_AND
```

```
bitwise_and = cv.bitwise_and(rectangle, circle)
cv.imshow("Bitwise AND", bitwise_and)

#bitwise_OR
bitwise_or = cv.bitwise_or (rectangle, circle)
cv.imshow("Bitwise OR", bitwise_or)

#bitwise_XOR
bitwise_xor = cv.bitwise_xor (rectangle, circle)
cv.imshow("Bitwise XOR", bitwise_xor)
```

```
cv.waitKey(0)
```



And here we have the non intersecting regions of these two images when you put them over each other.

And just to recap, this bitwise AND returns the intersection regions, bitwise OR returns the nonintersecting regions as well as the intersecting regions, while bitwise XOR returns only the nonintersecting regions.

So essentially, if you take this bitwise XOR and subtract it from bitwise OR, you get bitwise AND. And conversely, if you subtract bitwise AND from bitwise OR, you get bitwise XOR.

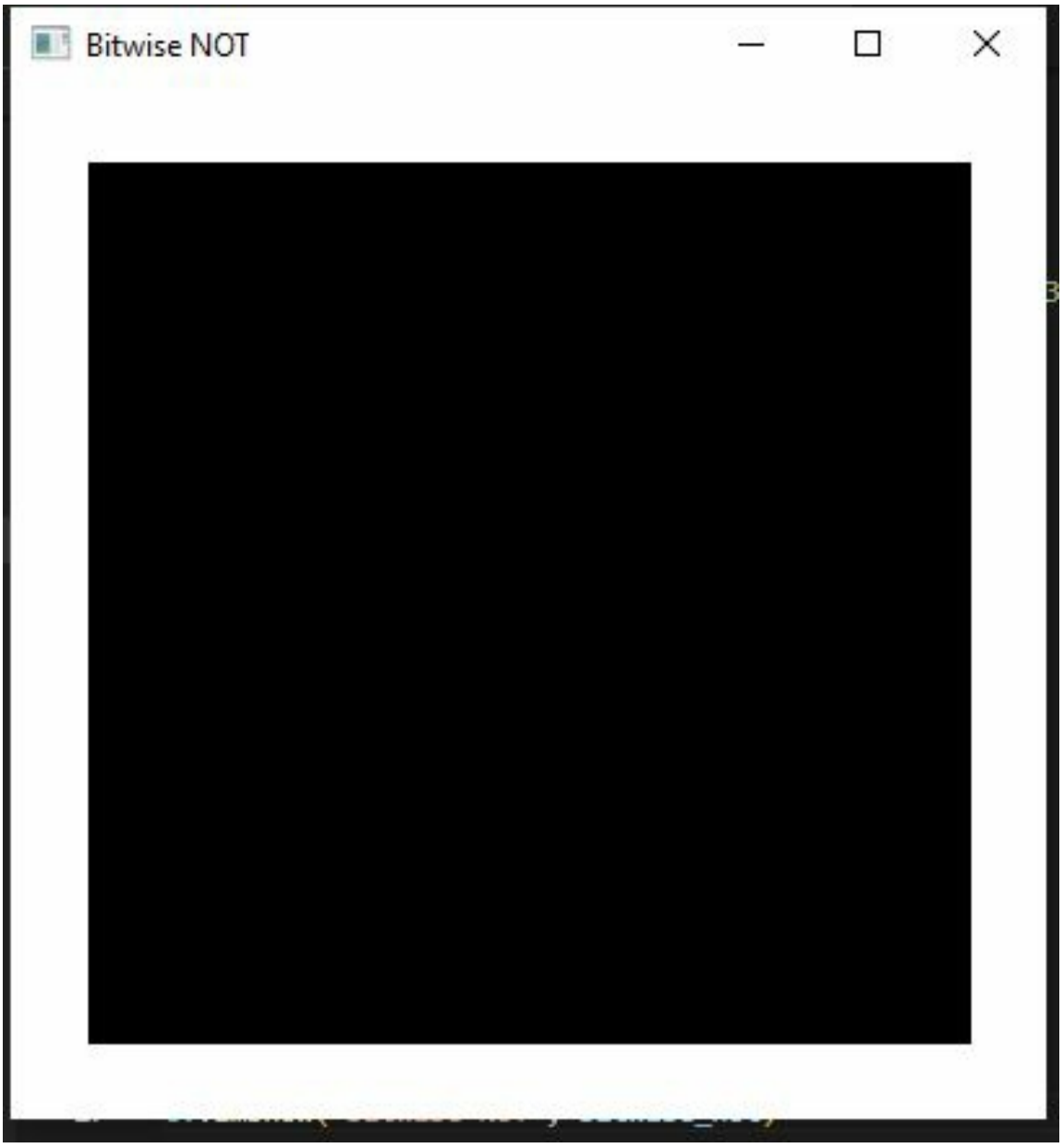
Bitwise NOT

And finally, the last method we can discuss is bitwise NOT. Essentially, it doesn't return anything. What it does is it inverts the binary color. Let us see the code:

```
import cv2 as cv
from cv2 import bitwise_and
import numpy as np

blank = np.zeros((400, 400), dtype="uint8")

rectangle = cv.rectangle(blank.copy(), (30,30), (370,370), 255, -1)
circle = cv.circle(blank.copy(), (200,200), 200, 255, -1)
cv.imshow('Rectangle', rectangle)
cv.imshow('Circle', circle)

#bitwise_AND

bitwise_and = cv.bitwise_and(rectangle, circle)
cv.imshow("Bitwise AND", bitwise_and)

#bitwise_OR
bitwise_or = cv.bitwise_or (rectangle, circle)
cv.imshow("Bitwise OR", bitwise_or)

#bitwise_XOR
bitwise_xor = cv.bitwise_xor (rectangle, circle)
cv.imshow("Bitwise XOR", bitwise_xor)

#bitwise_NOT
bitwise_not = cv.bitwise_not (rectangle)
cv.imshow("Bitwise NOT", bitwise_not)
```

```
cv.waitKey(0)
```

And basically, what it did is if you look at this image below, it found all the white regions, all the white pixels in the image and inverted them to black, and all the black images it inverted to white. Essentially, it converted the white to black and then from black to white.



Try that with the circle. Did you get a black hole? Great! Well done.

For this chapter, I just wanted to introduce you all to the idea of bitwise operations and how it works. In the next chapter, we'll discuss how to use these bitwise operations in a concept called masking.

Chapter 12 - Masking

In this chapter, we're going to be talking about masking in open CV. Now in the previous chapter, we discussed bitwise operations. And using those bitwise operations, we can essentially perform masking in OpenCV.

Masking essentially allows us to focus on certain parts of an image that we'd like to focus on. And as I said before, we can make regions of interest by using both bitwise operations and masks. This lets us pull out regions from images that can have any shape we want.

A mask lets us focus on only the parts of an image that we find interesting. For example, let's say we wanted to build a computer system that could recognize faces. We only want to find and describe the parts of the image with faces. We don't care about anything else in the image. As long as we can find the faces in the picture, we can only make a mask that will show the faces.

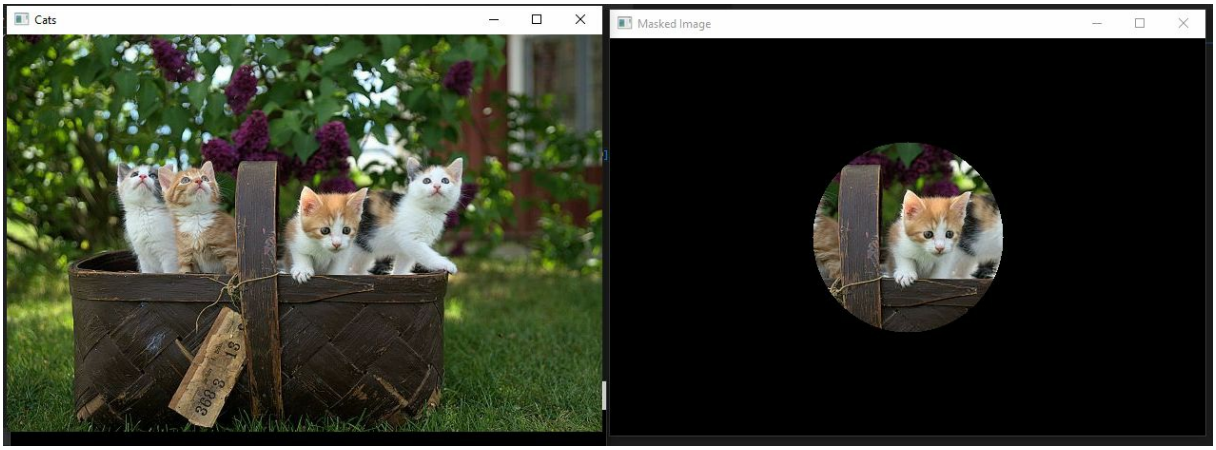
Image Masking with OpenCV

The first thing to do is to import NumPy and set it in the blank image for the masking. Then, we make a NumPy array with the same width and height as our original image and fill it with 0. Look at the code below:

```
import cv2 as cv
import numpy as np

img = cv.imread ('images/cats.jpg')
cv.imshow('Cats', img)

blank = np.zeros(img.shape[:2], dtype='uint8')

mask = cv.circle(blank, (img.shape[1]//2,img.shape[0]//2), 100, 255, -1)
cv.imshow('Mask', mask)

masked = cv.bitwise_and(img,img,mask=mask)
cv.imshow('Masked Image', masked)

cv.waitKey(0)
```

We will use the np.zeros of the size of size image dot shape with the first 2

values in the array. Now, this is extremely important. The dimensions of the mask have to be the same size as that of the image.



And this is essentially our mask. There's the blank image we're working with. And this is the image that we want to mask. We have created a mask over the image only to display one portion of the image, which is the face of a cat.

You can play around with the values. For example, you can tweak the line “mask = cv.circle(blank, (img.shape[1]//2,img.shape[0]//2), 100, 255, -1)” and add + 45 in front of the first img.shape list. That would essentially move the circle on top of another cat. You can also draw a rectangle instead of a circle as in the code.

And essentially, you can play around with these as you feel fit. You can maybe try different shapes, weird shapes. And the way you can get these weird shapes, essentially creating a circle or rectangle and applying bitwise AND you get this weird shape. And then you can use that weird shape as your mask.

So that's it for this chapter. We talked about masking. Again, nothing to do differently. We've essentially used the concept of bitcoins from the previous chapter. The use of masks enables us to narrow our computation to only those parts of the image that we're interested in using. Computer science topics such as image classification and object detection can be significantly improved by focusing our computations on the areas of interest to us.

Let's say we wanted to create a system for identifying different varieties of flowers.

To classify the flowers, we are most likely only concerned with the color and texture of the petals. However, because we're taking the picture in the wild, we'll have a lot of other elements in the picture, such as dirt from the ground, insects, and other flowers. How are we going to identify and categorize just the flower that we care about? The answer is masking, as we'll see in the next chapter.

Chapter 13 - Histogram Computation

In this chapter, we're going to be talking about computing histograms in OpenCV.

Now histograms allow you to visualize the distribution of pixel intensities in an image. So whether it's a color image or whether it's a grayscale image, you can visualize these pixel intensity distributions with the help of a histogram, which is kind of like a graph or a plot that will give you a high-level intuition of the pixel distribution in the image.

So we can compute a histogram for grayscale images and compute a histogram for RGB images.

Working with CalcHist() Method

An image's histogram can be viewed as a graph or plot that shows the intensity distribution in an image, with the x-axis representing pixel values and the y-axis representing the image's pixel count.

OpenCV's calcHist() function can be used to determine an image's histogram.

The source image is one of five parameters passed to the calcHist() function. Channel, mask, histSize, and range are all defined here.

A square bracketed value must be entered for the parameter source image to calculate a histogram.

The index used to calculate the histogram is called the parameter channel. The channel's value is set to [0] for a grayscale image. Blue, green, and red colors each have a channel value of [0], [1], [2], and the value is specified in square brackets.

The histogram does not have to be found in the area specified by the parameter mask. If we don't want to mask any part of the image, the value of mask id None should be used.

You can specify the number of BINs in the histSize parameter using square brackets. Histograms can be divided into 16 subparts, and the sum of all the pixels in each subpart is known as BIN, which is represented as histSize in a histogram graph.

If the parameter range is [0,256], it specifies the range of intensity values.

If you use the calcHist() function on an image, you'll get a 256*1 array with each element representing a one-pixel value.

Histogram for Grayscale Images

So we're going to start off with computing histograms for grayscale images. If you have an RGB image, you must first convert it to grayscale with the cvt.Color() function.

```
import cv2 as cv
import matplotlib.pyplot as pIt

img = cv.imread ('images/cats.jpg')
cv.imshow('Cats', img)

gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
cv.imshow('Gray', gray)
```

```
gray_hist = cv.calcHist([gray], [0], None, [256], [0,256] )
plt.figure()
plt.title('Grayscale Histogram')
plt.xlabel('Bins')
plt.ylabe1('# of pixels')
plt.plot(gray_hist)
plt.xlim([0,256])
Plt.show()
```

```
cv.waitKey(0)
```

Now to actually compute the grayscale histogram. What we need to do is essentially call this cv.calcHist (). This method will essentially compute the histogram for the the image that we pass into using the following syntax: (images: List[Mat], channels: List[int], mask: Mat | None, histSize: List[int], ranges: List[int], hist: ... = ..., accumulate: ... = ...).

This image is a list, so we need to pass in a list of images. Since we're only interested in computing a histogram for one image, we pass in the grayscale image (gray). Next, we have to pass in the number of channels that specifies

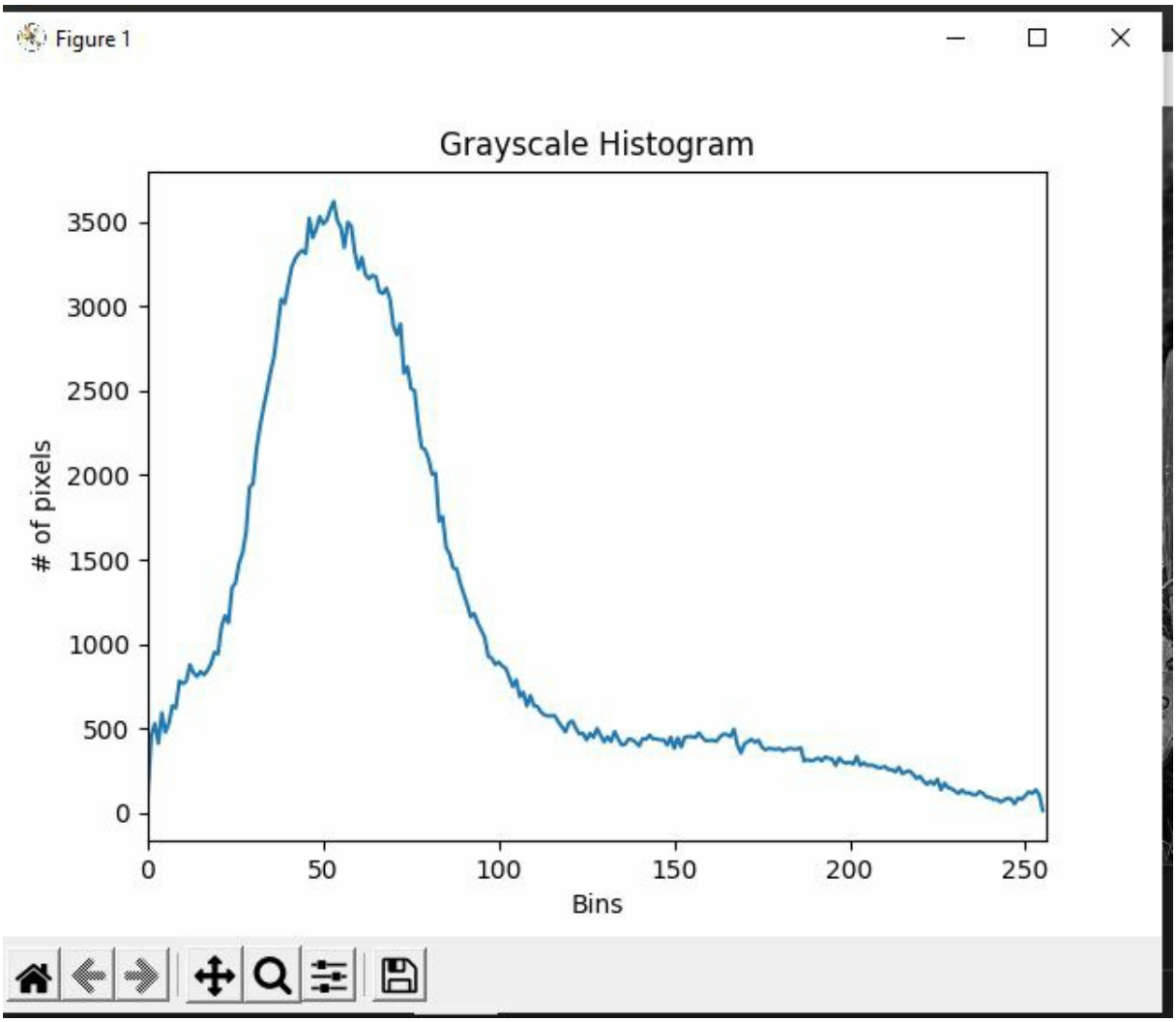
the index of the channel we want to compute a histogram for that since we are computing the histogram for a grayscale image, we wrapped it as a list and passed in zero.

The next thing we do is provide a mask, but before that, we want to compute a histogram for a specific portion of an image. But for now, just have this Histsize to None. Histsize is the number of bins we want to use to compute the histogram.

Essentially, when we plot a histogram, I'll talk about this concept of bins. But essentially, for now, just set this to 256 wrapped as a list. And that's wrapped out as a list. And the next thing I want to do is specify the range of the range of all possible pixel values. Now for our case, this will be 0, 256. And that's it.

Then to plot the image, we use matplotlib. And then we instantiated plt.figure(), gave it a label across the x-axis and called it Bins. The y label # of pixels.

We give it a limit across the x-axis with plt.xlim of a list of 02256. And finally, we can display this image.



And this is the distribution of pixels in this image. As you can see, the number of bins across the x-axis represents the intervals of pixel intensities.

So as you can see, there is a peak in this region, which means that this is close to 50 to 60. So this means that in this image, there are close to 4000 pixels that have an intensity of about 60.

And as you can see that there's a lot of, there's a lot of peeking in this region, so between probably 40 to 70, there is a peak of pixel intensities of close to 3000 pixel intensities in this image. So this is essentially computing the grayscale histogram for the entire image.

You can try this with a different image. Play around and see the difference in the intensity of pixels.

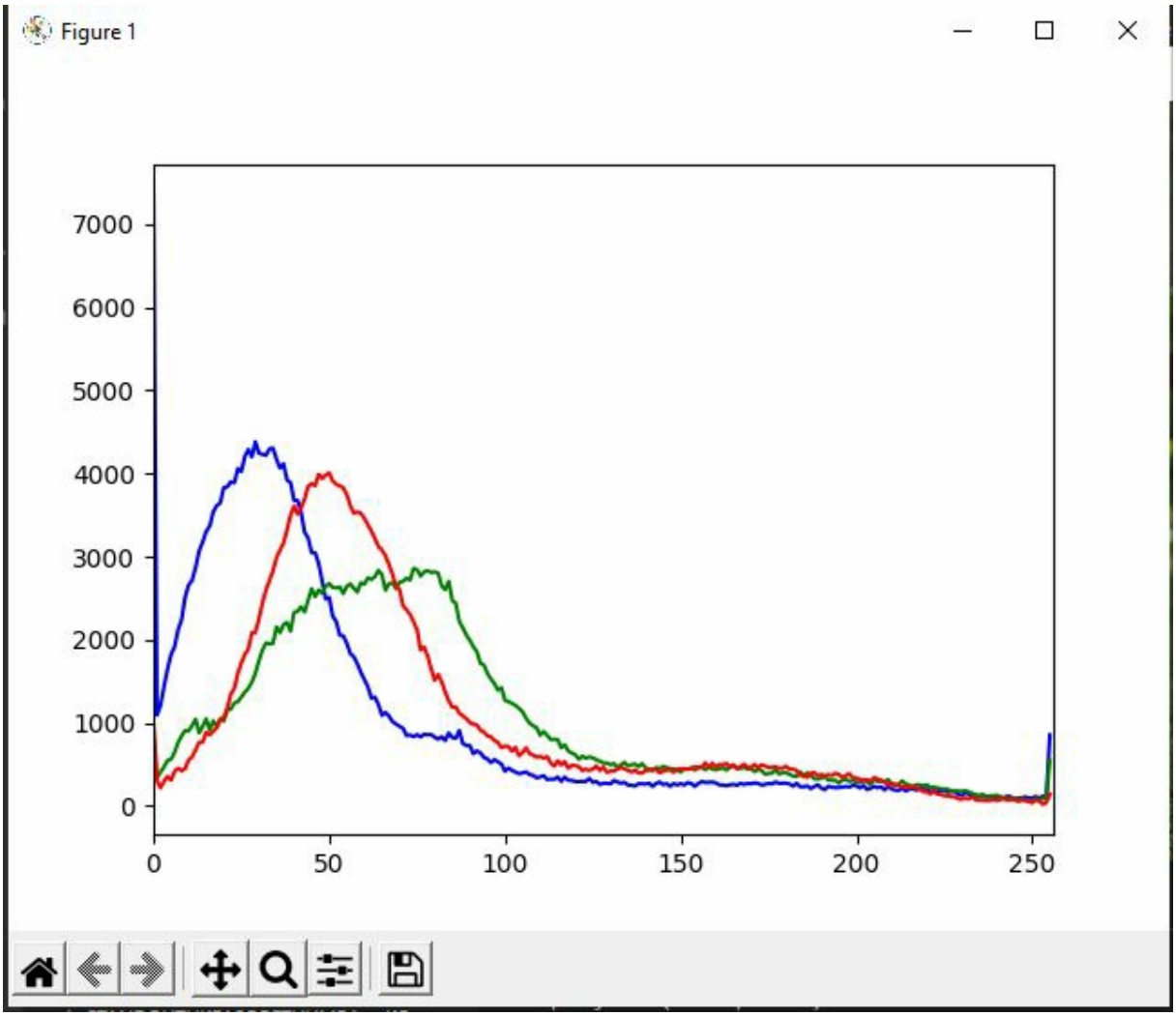
Histogram Computation for RGB Images

To compute a color histogram, that is to compute a histogram for a color image to an RGB image. Remember how we talked about the color channels, right? We will use it here.

```
import cv2 as cv
import matplotlib.pyplot as plt

img = cv.imread ('images/cats.jpg')
cv.imshow('Cats', img)

# gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
# cv.imshow('Gray', gray)

color = ('b','g','r')
for k,color in enumerate(color):
    histogram = cv.calcHist([img],[k],None,[256],[0,256])
plt.plot(histogram,color = color)
plt.xlim([0,256])
plt.show()
```

```
cv.waitKey(0)
```

The modules cv2, NumPy, and matplotlib are imported in the code above. The imread() function will then be used to read the image and calculate its histogram. This is followed by a step where we specify the colors that will be used as values of colors in calcHist() so that we can calculate the histogram of an image and plot it to the screen. The results are shown in the image below.



So essentially, that's it for this chapter. Histograms allow you to analyze the distribution of pixel intensities, whether for a grayscale image or a colored image. Now, these are helpful in a lot of advanced computer vision projects. When you are actually trying to analyze the image that you get, maybe try to equalize the image so that there's no peeking of pixel values here and there.

In the next chapter, we'll be talking about how to threshold an image and the different types of thresholding.

CHAPTER 14 - THRESHOLDING/BINARIZING IMAGES

In this chapter, we're going to be talking about thresholding in OpenCV. OpenCV is an image processing technique. OpenCV's threshold is helpful in determining the values of individual pixels. A threshold value has been set for each of the pixels in this image. Suppose the pixel value is greater than the threshold value. In that case, the image is segmented by setting this value to the maximum and setting it to zero if it is less. This is the most common method of image segmentation because it yields the best results. We'll be using a function in Python to set this value in the near future.

Now, thresholding is a binarisation of an image. In general, we want to take an image and convert it to a binary image that is an image where pixels are either zero or black, or 255, or white.

A very simple example of thresholding would be to take an image and take some particular value that we're going to call the thresholding value. And compare each pixel of the image to this threshold value. If that pixel intensity is less than the threshold value, we set that pixel intensity to zero. And, and if it is above this threshold value, we set it to 255, or white.

The images must be segmented in order to produce binary images. OpenCV threshold was used to perform this segmentation. Simple thresholding and adaptive thresholding are both used here. There must be a corresponding threshold value for the pixel value that is used.

As mentioned before, pixels with values less than or equal to the threshold are set to 0, while pixels with values greater than or equal to the threshold are set to the maximum value.

So in this sense, we can essentially create a binary image just from a regular standalone image. So in this chapter, we're going to talk about two different types of thresholding, simple thresholding and adaptive thresholding.

Simple Thresholding

So essentially, to apply this this idea of simple thresholding, we essentially use the cv.threshold () function. Now, this function returns a threshold. And this, in essence, takes in a grayscale image. The grayscale image has to be passed in to this thresholding function. If you have a BGR image, you have to

convert it to grayscale.

Then what we do is we pass in a threshold value, we set this to 150 for now, and we have to specify something called a maximum value. So if that pixel value is greater than is greater than 150, what do you want to set it to? In this case, we want to binarize the image. So we set it to 245. And finally, we can specify a thresholding type.

Now, this thresholding type is essentially cv.thresh_binary(). And what this does is looks at the image and compare each pixel value to this threshold value. And if it is above this value, it sets it to 255. Otherwise, it infers that it sets it to zero if it falls below.

So essentially returns two things thresh: the thresholded or binarized image and the threshold, which is the same value that you passed 150. The same threshold value you pass in will be returned to this threshold value.

Check the code:

```
import cv2 as cv
```

```
img = cv.imread ('images/cats.jpg')
cv.imshow('Cats', img)

gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

#Simple Thresholding
threshold, thresh = cv.threshold(gray, 150, 255, cv.THRESH_BINARY)
cv.imshow("Simple Thresh", thresh)
```

```
cv.waitKey(0)
```

This is a thresholded image that you get.

Again, this is nothing different from when we discussed thresholding in one of the previous chapters, but this is what you get. So you can play around with these threshold values, and let's see what that does.

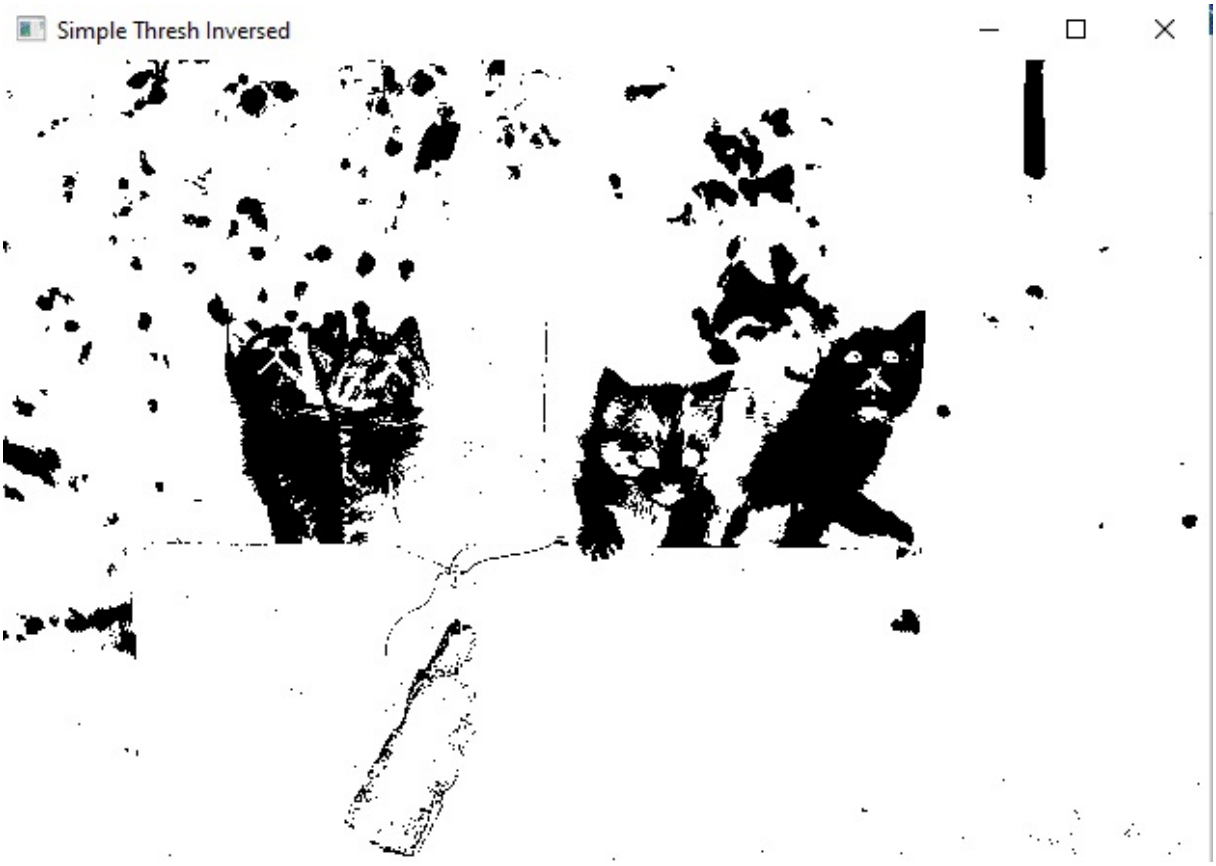
So what we can do after this is create an inverse thresholded image.

```
import cv2 as cv
```

```
img = cv.imread ('images/cats.jpg')
cv.imshow('Cats', img)

gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

#Simple Thresholding
threshold, thresh = cv.threshold(gray, 150, 255, cv.THRESH_BINARY)
cv.imshow("Simple Thresh", thresh)

#Inverse
threshold, thresh_inv = cv.threshold(gray, 150, 255, cv.THRESH_BINARY_INV)
cv.imshow("Simple Thresh Inversed", thresh_inv)
```

```
cv.waitKey(0)
```

In this case, we have used the inverse form of the threshold. And this is essentially the inverse of this image. Instead of setting pixel intensities greater than 150 to 255, it sets whatever values are less than 150 to 255.



So that's essentially what you get. Right, all the black parts of this image will change to white, and all the white parts of the image will change to black. So that's a simple threshold. Let's move on now to the adaptive threshold.

Adaptive Thresholding

As you can imagine, we got different images when we provided different threshold values.

Now, kind of one of the downsides to this is that we have to specify a specific threshold value manually. Now, in some cases, this might work. In more

advanced cases, this will not work.

So one of the things we could do is we could essentially let the computer find the optimal threshold value by itself. And using that value that refines it binary rises over the image. So that's the essence of the entire crux of adaptive thresholding.

So let's set up a variable called adaptive_thresh. And set this equal to cv.adaptive threshold(). And inside, I want to pass in a source image, which is gray. I'm going to pass in a maximum value, which is 255. Now notice there is no threshold value. The adaptive method tells OpenCV which method to use when computing the optimal threshold value. So, we're just going to set this to the mean of some neighborhood of pixels.

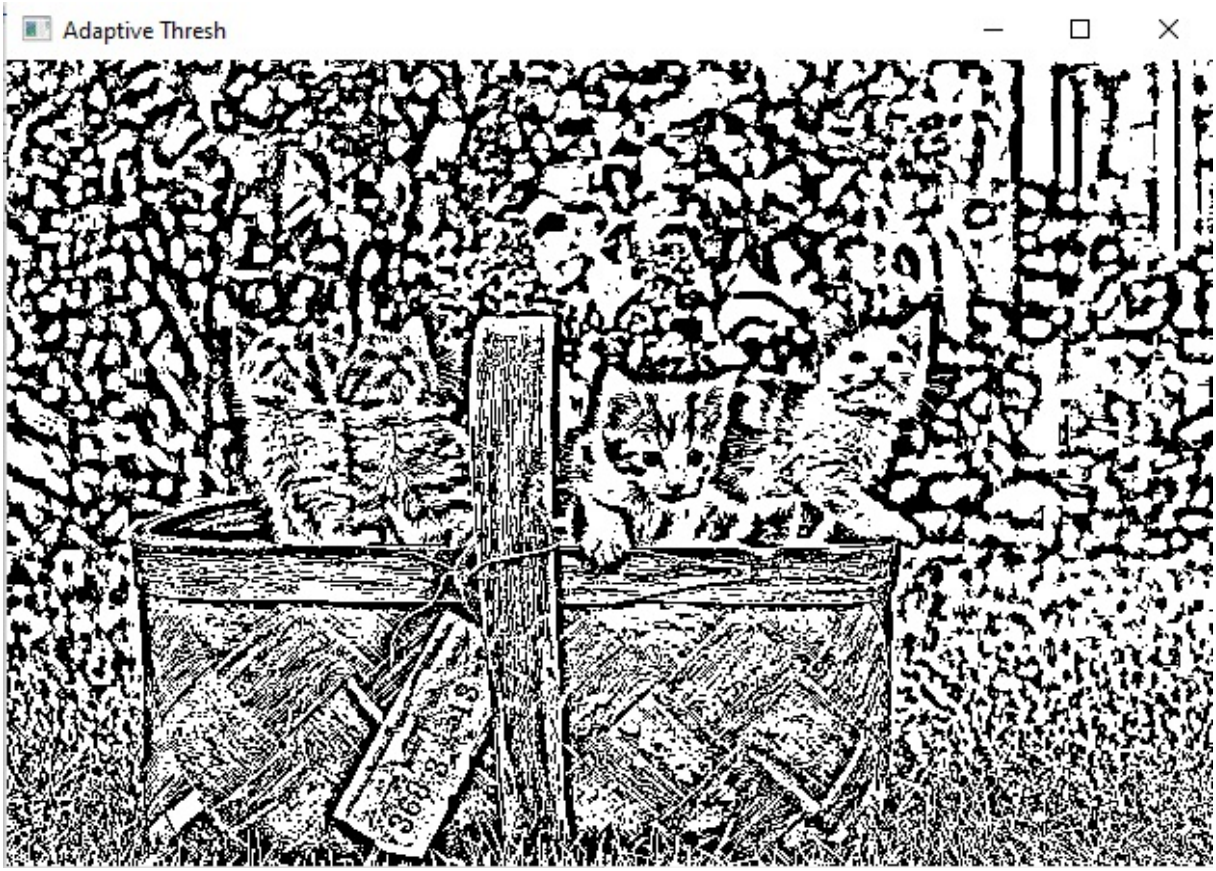
Next, we'll set up a threshold type. This is cv.thresh_binary, which works differently from the first example. And two other parameters that I want to specify are the block size, which is essentially the neighborhood size of the kernel size, which OpenCV needs to use to compute the mean to find the optimal threshold value. So, for now, let's set this to 11. And finally, the last method we have to specify is the c value. Now, this c value is an integer that is subtracted from the mean, allowing us to fine-tune our threshold. So again, don't worry too much about this. You can set this to zero. But for now, let's set this to 3. And finally, once that's done, we can go ahead and try to display this image. So let's call this adaptive thresholding. And we can pass in adaptive cash. So let's save that and run. And this is essentially your adaptive thresholding method. So essentially, what we've done is we've defined a kernel size or window that is drawn on this image. In our case, this is 11 by 11. And so what open CV does is it essentially computes a mean over those neighborhood pixels and finds the optimal threshold value for that specific part. And then it slides over to the right, and it slides, it does the same thing. And it's lines down and does the same thing so that it essentially slides over every part of the image. So that's how adaptive thresholding works.

Here is the code:

```
import cv2 as cv
```

```
img = cv.imread ('images/cats.jpg')
cv.imshow('Cats', img)

gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

#Simple Thresholding
threshold, thresh = cv.threshold(gray, 150, 255, cv.THRESH_BINARY)
cv.imshow("Simple Thresh", thresh)

#Inverse
threshold, thresh_inv = cv.threshold(gray, 150, 255, cv.THRESH_BINARY_INV)
cv.imshow("Simple Thresh Inversed", thresh_inv)

#Adaptive Thresholding
adaptive_thresh = cv.adaptiveThreshold(gray, 255, cv.ADAPTIVE_THRESH_MEAN_C,
cv.THRESH_BINARY, 11, 3)
cv.imshow('Adaptive Thresh', adaptive_thresh)

cv.waitKey(0)
```

This is the result:



If you wanted to fine-tune this, we could change this to a threshold. We will use binary_inv. You're just to see what's going on under the hood. So all the white parts of the image will change the black, and all the black parts of the image have changed to white. You can play around with these values.

You can also try the Gaussian instead of the mean. The only difference that Gaussian applied was essentially adding a weight to each pixel value and computing the mean across those pixels. So that's why we were able to get a better image than when we used the mean. But essentially, the adaptive thresholding mean works effectively.

In some cases, the Gaussian works. In other cases, there's no real one size fits all. So really play around with these values and see what you get. But that's essentially all we have to discuss.

For this chapter, we talked about two different types of thresholding, simple thresholding, and adaptive thresholding. In simple thresholding, we have to specify a threshold value manually. And in adaptive thresholding, OpenCV does that for us using a specific block size or current size and other factors,

computing the threshold of value on the basis of the mean or the basis of the Gaussian distribution.

So in the next chapter, we're going to be discussing how to compute gradients and edges in an image.

Chapter 15 – Gradients and Edge Detection in OpenCV

In this chapter, we're going to be talking about gradients and edge detection in OpenCV.

It is possible to detect objects or regions' boundaries (edges) within an image using an image-processing technique known as "edge detection." Images would be incomplete without their edges. Edges reveal the structure of an image's underlying structure. So edge detection is heavily used in computer vision processing pipelines.

Now, you could think of gradients as these edge-like regions that are present in an image. Now, they're not the same thing. Gradients and edges are entirely different things from a mathematical point of view. But you can get away with only thinking of gradients as edges from a programming perspective.

So essentially, in the previous chapter, we've discussed the canny edge detector, which is essentially kind of an advanced edge detection algorithm. That is essentially a multi-step process.

But in this chapter, we're going to be talking about two other ways to compute edges in an image.

How Do We Detect the Edges?

The intensity of a pixel changes dramatically at the edge of a picture. To locate edges, we must examine the adjacent pixels for any changes that might indicate their presence. We have learned how to use Canny Edge Detector. In this chapter, we will learn about OpenCV's Sobel Edge Detection and Laplacian methods, two important algorithms for edge detection. We'll go over the basics and show you how to put them to use in OpenCV.

And that is the Laplacian and the Sobel method.

Laplacian Edge Detector

So we will start off with the Laplacian edge detector. So we will do this with the cv.Laplacian() method. And this method is it will take in a source image,

which is gray, and the next argument is the ddepth or data depth. Now, we set this to cv.64F. I'm going to explain later. Look at the code below:

```
import cv2 as cv
from cv2 import cvtColor
import numpy as np

img = cv.imread ('images/cats.jpg')
cv.imshow('Cats', img)

gray = cvtColor(img, cv.COLOR_BGR2GRAY)

#laplacian method
lap = cv.Laplacian(gray, cv.CV_64F)
lap = np.uint8(np.absolute(lap))
cv.imshow ('Laplacian', lap)

cv.waitKey(0)
```

And this is essentially the Laplacian edges in the image. It looks like an image that is drawn over a chalkboard and then smudged just a bit.

And it looks like a pencil shading off the original image. It's all the edges that exist in the image, or at least most of the edges in the image are essentially drawn over with the pencil and then lightly submerged.

So that's essentially the Laplacian edges. So again, don't worry too much about why we converted this to in the uint8, and then we computed the absolute value. But essentially, the Laplacian method computes the gradients of this grayscale image.

Generally, this involves a lot of mathematics, but Essentially, when you transition from black to white and white to black, that's considered a positive and a negative slope.

Now, images themselves cannot have negative pixel values. So what we do is compute the absolute value of that image. So all the pixel values of the image are converted to the absolute values. And then, we convert that to a uint8 image-specific datatype. So that's the crux of what's going on right over here.

Sobel Edge Detection

So let's move on to the next one. And that is the Sobel gradient magnitude representation. So essentially, this does that Sobel computes the gradients in two directions, the x and y.

Suppose you are using the Python programming language's OpenCV library. In that case, you can use the OpenCV Sobel operator() command, which can be used to detect both vertical as well as horizontal edges within an image. It's a crucial part of image processing because it's one of the most fundamental operations that take place during the process. OpenCV Sobel operator command helps us introduce the total number of pixels (data being fed) to be processed by the system and aids in maintaining the image's structural dimensions and aspects. Sobel edge detector is a gradient-based method based on the first-order derivatives. The X and Y axes' first derivatives are calculated separately.

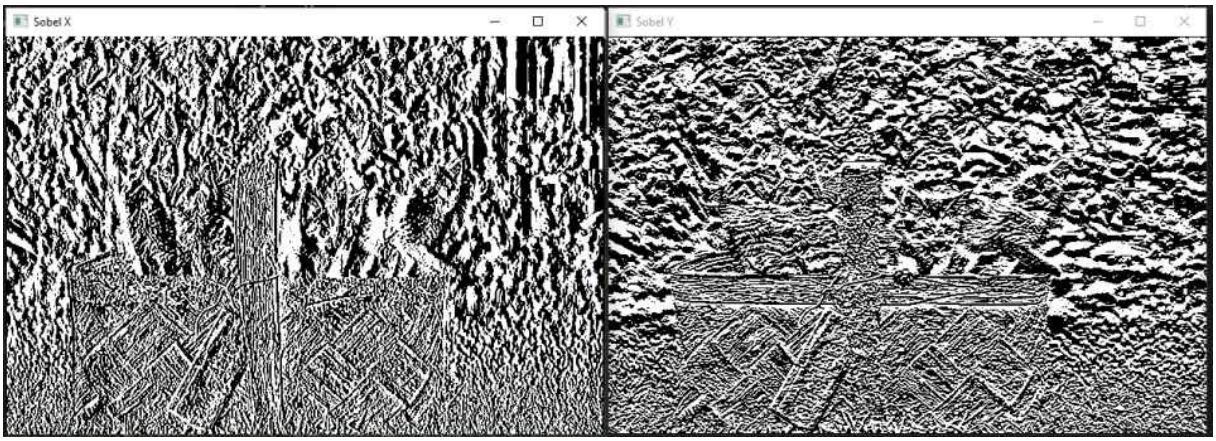
For horizontal and vertical changes, the operator uses two 3X3 kernels that are convolved with the original image to calculate approximations of the

derivatives.

It's a technique based on a gradient and the first-order derivatives of that gradient. It works by calculating the first derivative of the image, which is provided for the Y-axis and the X-axis, and operating separately on each. Using two 3 by 3 kernels, the operator concatenates the user's photo with the kernels to produce the final image. Calculation of the approximation values of the vertical and horizontal derivatives. So watch the code below closely:

```
import cv2 as cv
from cv2 import cvtColor
import numpy as np

img = cv.imread ('images/cats.jpg')
cv.imshow('Cats', img)

gray = cvtColor(img, cv.COLOR_BGR2GRAY)

#laplacian method
lap = cv.Laplacian(gray, cv.CV_64F)
lap = np.uint8(np.absolute(lap))
cv.imshow ('Laplacian', lap)

#Sobel
sobelx = cv.Sobel(gray, cv.CV_64F, 1, 0)
sobely = cv.Sobel(gray, cv.CV_64F, 0, 1)

cv.imshow('Sobel X', sobelx)
cv.imshow('Sobel Y', sobely)

cv.waitKey(0)
```

Using the cv.Sobel() method, we displayed the gradients that are computed along the x-axis and the y-axis individually. We passed in the image, gray. We passed in a data depth, which is cv.CV_64F. And we give it an x-direction 1 and the y-direction 0. We do the same for the y axis



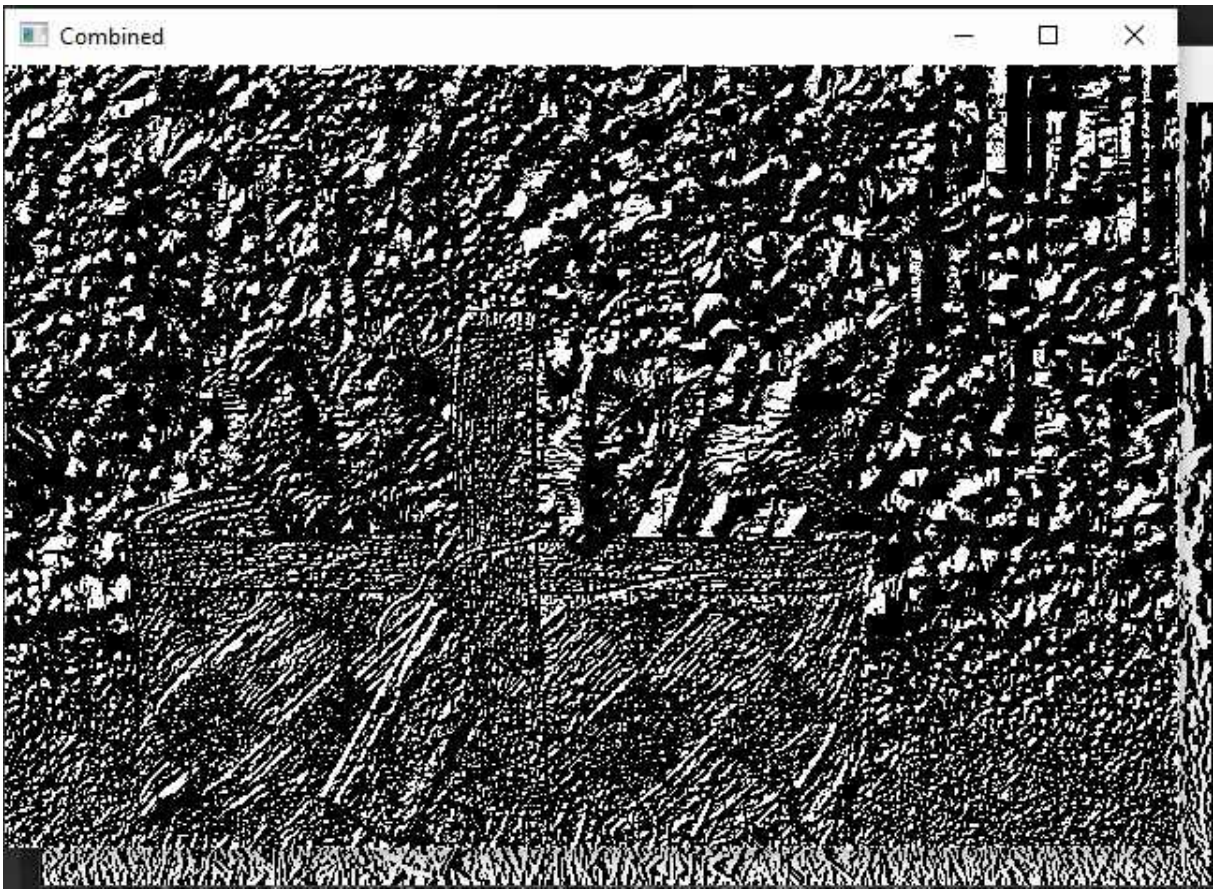
So you can see a lot of y horizontal specific gradients, and the Sobel x was computed across the y axis. So you can see the y-axis specific gradients.

We can get the combined Sobel image by combining these two Sobel x and Sobel y.

How can we do that? Simple. We use bitwise operators! Let us add this code to our script:

```
combined_sob = cv.bitwise_or(sobelx, sobely)
cv.imshow('Combined', combined_sob)
```

And this is essentially the combined Sobel that you get.



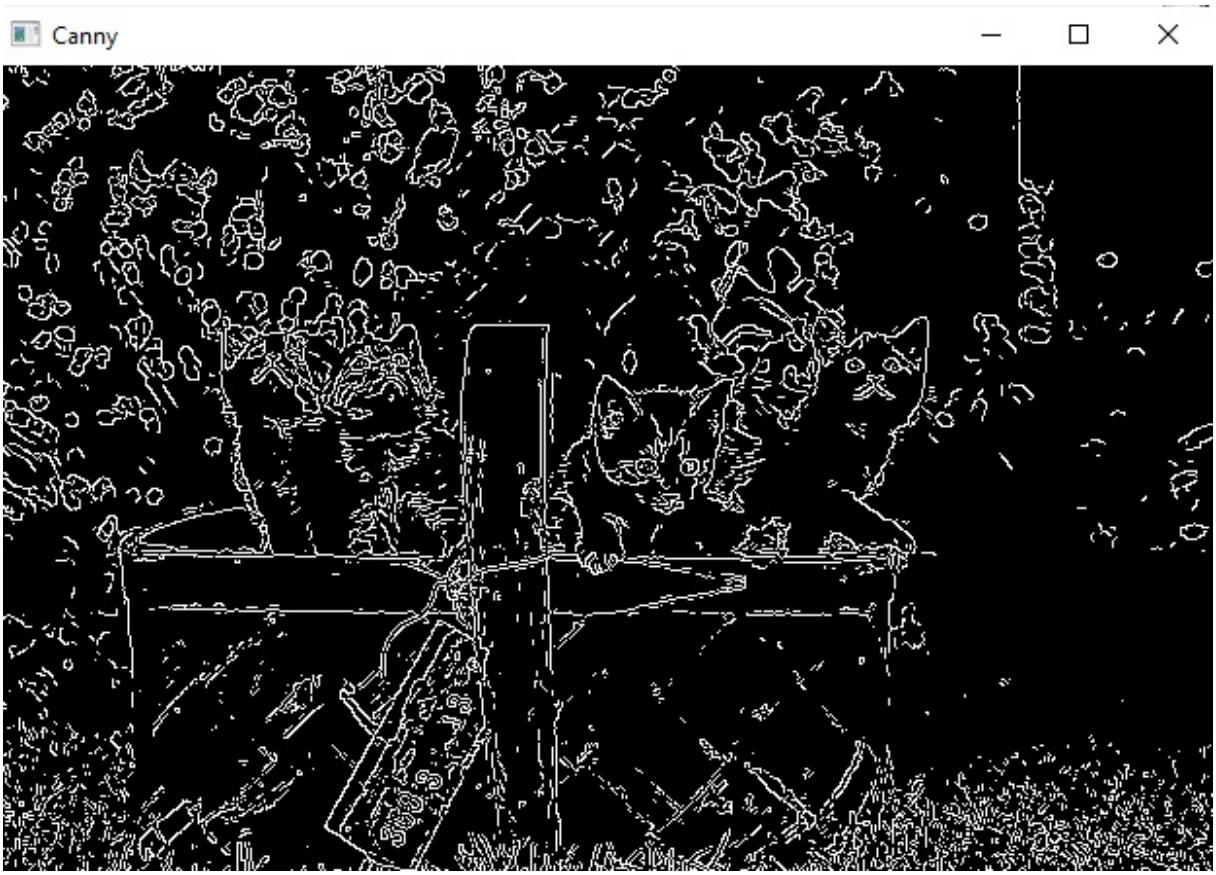
So it essentially took the other two Sobels, and using the bitwise OR, it got this image.

You can look at the image we got from the combined Sobel and compare it with the Laplacian edge detector. They are two completely different algorithms, so your results will be completely different.

Now, we can make a comparison of both Laplacian and the Sobel with the Canny edge detector.

You know the code for the Canny Edge Detector tool, don’t you?

```
canny = cv.Canny(gray, 150, 175)
cv.imshow("Canny", canny)
```



And let's see what that gives us. So let's compare that with the other two. So that's essentially it.

You can see the Canny in gradient representation, which essentially returns this pencil shading version of the edges in the image. Combined, Sobel computes the gradients in the X in the Y direction. And we can combine these two with bitwise OR, and Canny is a more advanced algorithm that uses Sobel in one of its stages.

As I mentioned, Canny is a multi-stage process. One of its stages is using the Sobel method to compute the gradients of the image. So essentially, you see that the canny edge detector is a cleaner version of the edges that can be found in the image. So that's why in most cases, you're going to see the Canny Edge Detector tool used.

But in more advanced cases, you're probably going to see a Sobel a lot. Laplacian is not used that commonly.

Moving on to the next section, we will be discussing face detection and face

recognition in OpenCV. We're going to touch on using haad cascades to perform some face detection and face recognition. There will be two parts. Face Recognition with OpenCV is a built-in face recognizer. And the second part will be building our own deep learning model to recognize some faces in an image.

Section #3 - Faces:

Chapter 16 - Face Detection with Haar Cascades

We are now with the last part of this Python and openCV tutorial, where we will talk about face detection and face recognition in openCV. For the past few years, face detection has been one of the most hotly debated topics in computer vision.

Everywhere you look these days, you'll find people using this technology. When you upload a photo to Facebook, the social media site automatically tags the people in it, making it easier for you to remember who is in the photos you're uploading.

So what we're going to be doing in this chapter is discussing how to detect faces in OpenCV using something called a haar cascade.

In the next chapter, we will talk about how to recognize faces using OpenCV's built-in face recognizer. And after that, we will be implementing our deep learning model to recognize the face of Simpson characters and classify each one by name. We're going to create that from scratch and use OpenCV for all the pre-processing and displaying of images and other image related things.

Face Detection

Now, face detection is different from face recognition. Face Detection merely detects the presence of a face in an image, while face recognition involves identifying whose face it is. Now, we'll talk more about this later on in this course. For the detection of objects in an image, the Haar Cascade algorithm uses a function known as the cascade function, as well as many negative images and positive images, and was first proposed in 2001 by Michael Jones and Paul Viola in their research paper "Rapid Object Detection Using A Boosted Cascade of Simple Features."

But essentially, face detection is performed using classifiers. A classifier is essentially an algorithm that decides whether a given image is positive or negative, whether a face is present or not.

Now classifier needs to be trained on thousands and tens of thousands of

images with and without faces. But fortunately for us, openCV already comes with many pre-trained classifiers that we can use in any program. So essentially, the two main classifiers that exist today are haar cascades, and more advanced classifiers are local binary patterns. We're not going to explain local binary patterns in this course.

But essentially, the most advanced is haar cascade classifiers. They're not as prone to noise in an image compared to the other cascades.

Haar Cascade Classifier

First, we'll go over what Haar cascades are and how to use them with the OpenCV library in this tutorial. A Haar classifier, or a Haar cascade classifier, is an algorithm developed by OpenCV developers to learn objects and identify them in images and videos.

In the OpenCV library, there is a collection of pre-trained Haar cascades. Haar cascades are typically employed for:

Recognition of individuals through the use of facial recognition software

Eye detection

Mouth detection

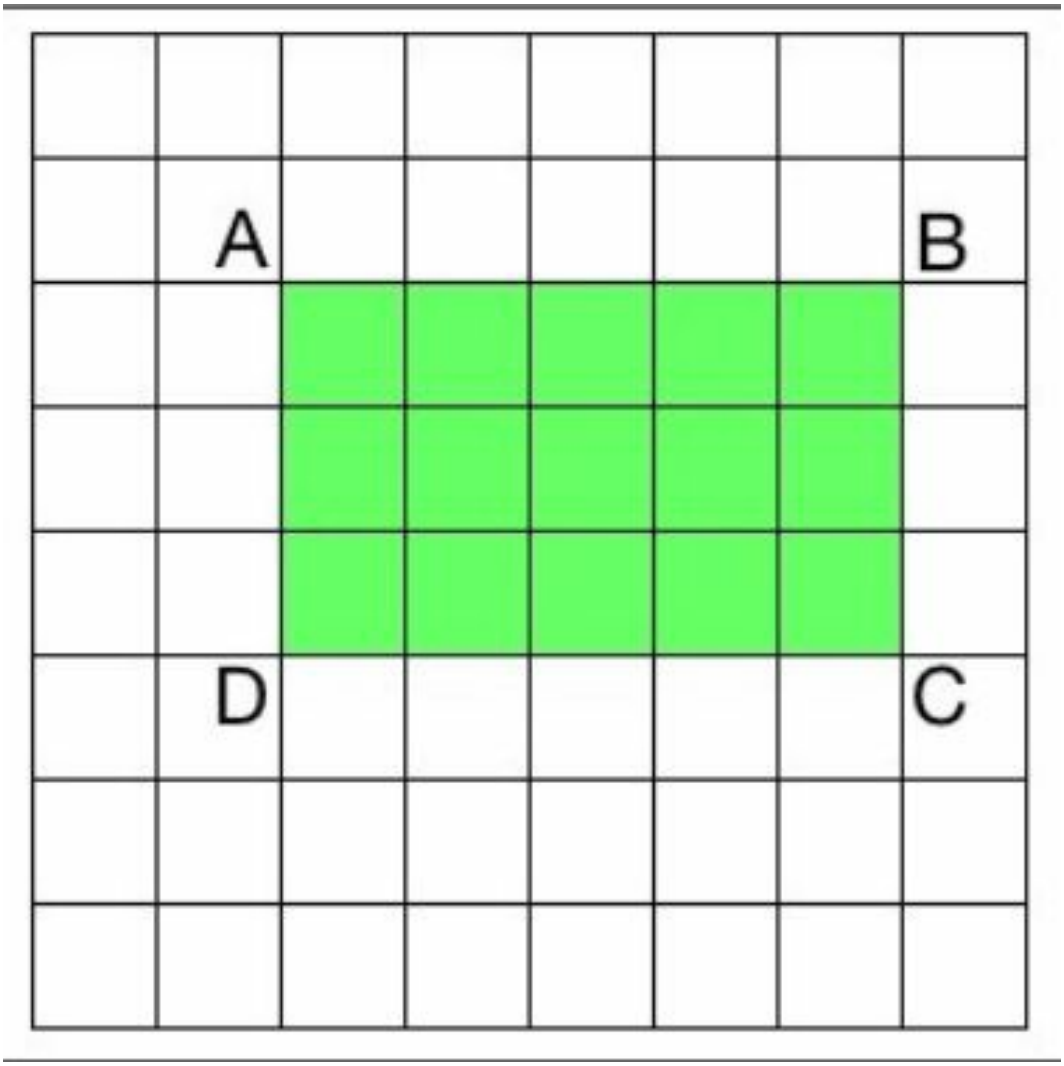
Detection in whole or in part of the human body

There are also other pre-trained Haar cascades, such as one for detecting Russian license plates and another for detecting cat faces.

Integral Images

If we want to find the Haar features of an image, we will have to find the sums of a lot of different rectangles or pixels in the image. If we want to build the feature set well, we need to do these sums at many different sizes. These Haar cascades use what we call integral images to calculate the regions of the image that we want to use. This is sometimes called ROI (region of interest).

The screenshot below is taken from the documentation provided by the developers of OpenCV. Now, this is used to explain how one region is a rectangle.



To figure out the total of any rectangle in an image, we don't have to add up all the things in that rectangle. Let's say that AP stands for the sum of all the points in the rectangle made by the top-left point and the point P, which are diagonally opposite corners in the image. So, if we want to figure out how big the area of the rectangle ABCD is, we can use this formula:

The area of the rectangle ABCD is equal to AC – (AB + AD – AA).

So we just use cascades to figure out the area of any rectangle and pull out the features.

Detecting Faces

OpenCV has an excellent framework for finding faces. All we have to do is load the cascade file and use it to find faces in an image. The Haar Cascade

can be found in the official OpenCV GitHub, where they maintain a repository of pre-trained Haar cascades.

If you follow that page, you will find the Haar cascade classifiers. And as you can see, there are plenty of haar cascades that openCV makes available to the general public. You have a Haar Cascade pretrained to detect a smile, for detection of the upper body, and things like that. Feel free to use whatever you want. But in this chapter, we're going to be performing face detection. And for this, we're going to use the haarcascade_frontalface_default.XML.

If you are on the page, go ahead and open that GitHub folder. You're going to get about 33,000 lines of XML code. What you have to do is go to the button that reads “Raw” on the top of the page, and you'll get all the raw XML code. When you are on the code page, all you have to do is click Ctrl + A or Command + D if you're on a Mac, and click Ctrl + C or Command + C to copy all the XML code.

After that, go to your Visual Studio code or your editor and create a new file. Name your file haar_face.xml. Once your new XML file is open with a blank page, paste in all the 33,000 lines of XML code you copied from the OpenCV page. Save that, and our Haar classifier is ready to work!

So we're going to be using this Haar Cascade classifier we have just created to detect faces present in an image.

Now, let us go back to OpenCV. Create a new file. Call it face_detect.py and save. So in this file called face_detect.py, import cv2 as cv and read in an image of a person. It can be your picture or any other person.

The first thing I want to do is convert the image to grayscale. Face detection does not involve skin tone or the colors that are present in the image. These haar cascades essentially look at an object in an image. Using the edges, tries to determine whether it's a face. We don't need color in our image.

Now, look at the code below:

```
import cv2 as cv

img = cv.imread('images/lady.jpg')

cv.imshow("Lady", img)
```

```
gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

haar_cascade = cv.CascadeClassifier('haar_face.xml')

faces_rect = haar_cascade.detectMultiScale(gray, scaleFactor=1.1, minNeighbors=3)

print(f'Number of faces found = {len(faces_rect)}')
```

```
cv.waitKey(0)
```

So the way we do that is by essentially creating a haar cascade variable. So let's set this to haar_cascade. And we're going to use the cv.CascadeClassifier() method for this. We will then pass the path to the haar classifier XML file we have created first. That is as simple as saying 'haar_face.xml.' This cascade classifier class will read in those 33,000 lines of XML code and store that in a variable called haar_cascade.

So now that we've read in all haar cascade files, we write the next line of code to actually try to detect the face in the image we have read in before.

So the next line says faces_rect is equal to haar_cascade.detectMultiScale, and we pass in the image that we want to detect based on, which is gray. Next, we're going to pass in a scale factor set to 1.1.

Give it a variable called minimum neighbors, which essentially is a parameter that specifies the number of neighbors rectangle should have to be called a face. So let's set this to 3. So that's it. That's all we have to do. And essentially, what these lines do is this detect multiscale. An instance of the cascade classifier class will take this image, use these variables called scale factor and minimum labels to detect a face and return the rectangular coordinates of that face essentially as a list of faces_rect. That's exactly why we are giving it faces_rect to the rectangle.

So you can essentially print the number of faces that were found in this image by using the print() function and nesting the len() function. So we print the length of this faces_rect variable.

When you save that and run, you can see the number of faces found. In my

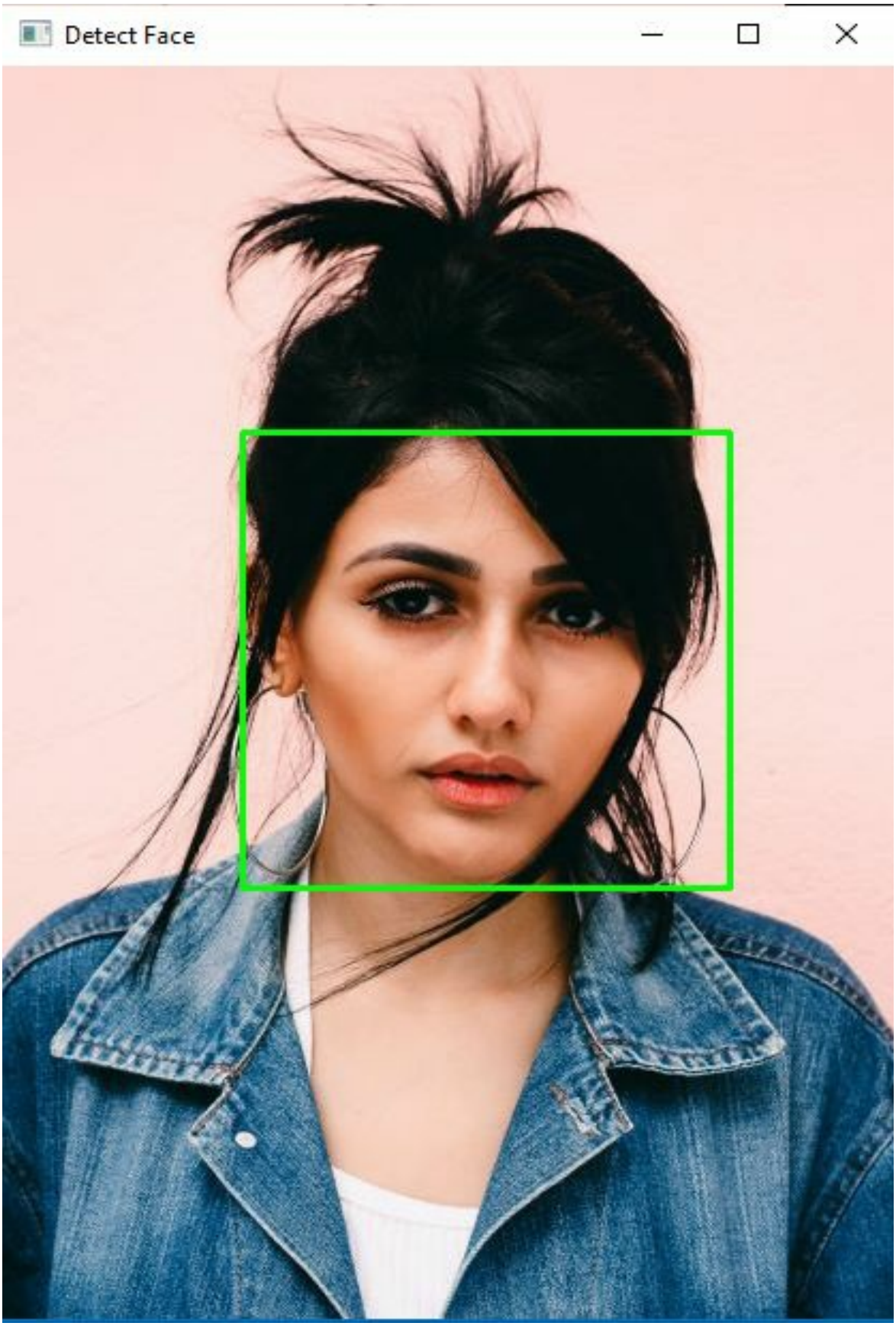
case, it returned 1, which is true because there's only one person in this image.

Now utilizing the fact that this faces_rect is essentially the rectangular coordinates for the faces that are present in the image, what we can do is we can essentially loop over this list and essentially grab the coordinates of those images and draw a rectangle over the detected faces.

So let's do that. So the way we do that is by saying for (x,y,w,h) in faces_rect, what we're going to do is we're going to draw a rectangle cv.rectangle over the original image. So we use the cv.rectangle() function and pass in the original image, img. Next, we give it a point one. This point one is essentially x, y. And point two is essentially x + w, y + h. Let's give it a color. Let's set this to green. So 0, 255, 0, give it a thickness of 2. And that's it. And we can print this, or we can display this image with cv.imshow. Look at the working code below:

```
import cv2 as cv

img = cv.imread('images/lady.jpg')

cv.imshow("Lady", img)

gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

haar_cascade = cv.CascadeClassifier('haar_face.xml')

faces_rect = haar_cascade.detectMultiScale(gray, scaleFactor=1.1, minNeighbors=3)

print(f'Number of faces found = {len(faces_rect)}')

for (x,y,w,h) in faces_rect:
    cv.rectangle(img, (x,y), (x+w,y+h), (0,255,0), thickness=2)

cv.imshow('Detect Face', img)

cv.waitKey(0)
```

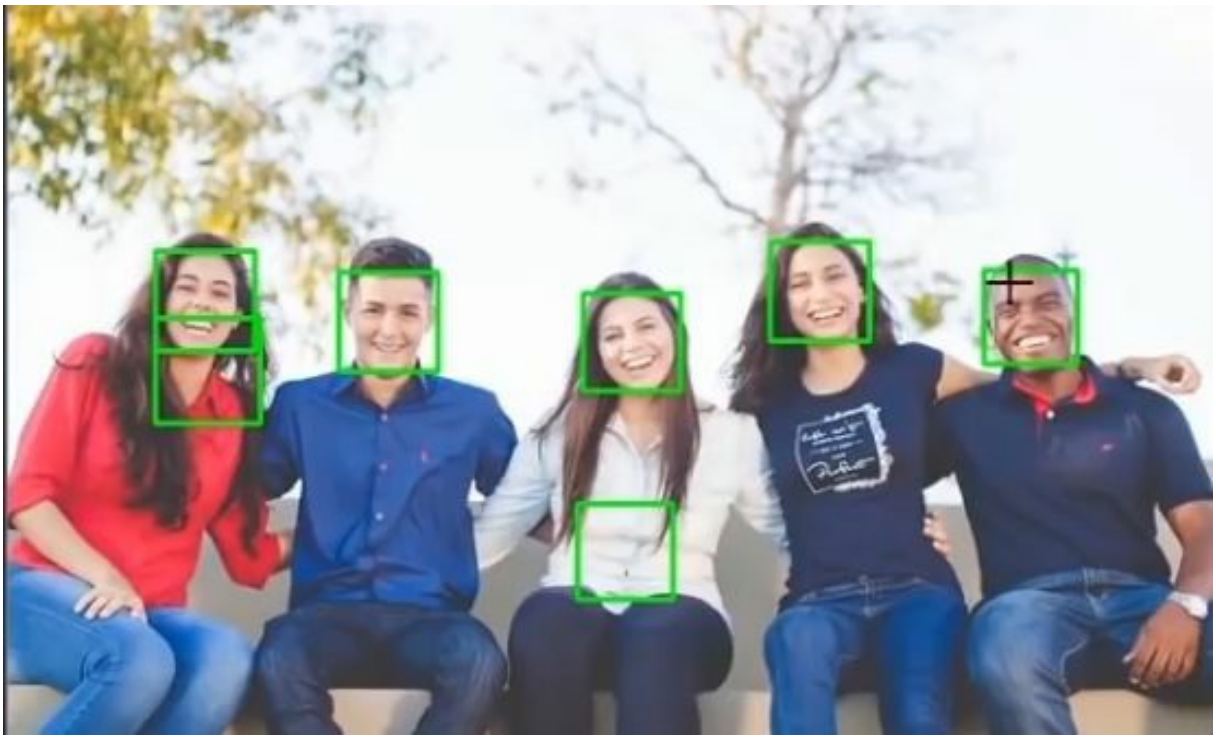
And if you look at this image, you can essentially see the rectangle that was drawn over this image.


Detect Face

So this, in essence, is the face that openCV's haar cascades found in this image.

So, how about you try this with different images. You can even have images that have a couple of people or an image with five or more people and try to see how many faces these haar cascades could detect in these images.

There are cases where OpenCV Haar cascades detect objects wrongly. For instance, in the screenshot below, OpenCV mistakes people's necks and stomachs for faces. Haha! There is a logical explanation for that.



For one thing, Haar Cascade classifiers are very sensitive to noise in an image, and the algorithm uses shadows and contours to identify objects. A way to minimize this error is by modifying the scale factor and the minimum neighbors.

Haar cascades are not the most effective object recognition program you can use, especially if you want to build a program with computer vision. However, it is a great way to start.

This chapter discussed how to detect faces in OpenCV using openCV’s haar cascades. In the next chapter, we will talk about how to recognize faces in OpenCV using the OpenCV built-in face recognizer.

Chapter 17 - Object Recognition with OpenCV's built-in recognizer

In this chapter, we will learn how to build a face recognition model in OpenCV using OpenCV's built-in face recognizer. In the previous chapter, we dealt with detecting faces in open CV using haar cascades. This chapter will cover how to recognize faces in an image.

So, when you hear the word "Face-Recognition," what comes to mind? You have often seen a face and remembered that it belongs to a certain person or when you meet someone for the first time, you don't know them, right? You look at this person’s face, eyes, nose, mouth, color, and overall look. This is your mind gathering information about several angles of their face to help you recognize that person's face. Then he tells you his name, Sam. Your mind now knows that the face information it just learned belongs to Sam. Now that your mind has been trained, it is ready to recognize Sam's face. The next time you see Sam or his face in a picture, you'll know who he is. Face recognition works like this. The more you see Sam, the more information your mind will gather about him, especially about his face, and the easier it will be for you to recognize him.

Now, how can we code face recognition with OpenCV? How does the computer accomplish that?

Face/Image Recognition is a part of OpenCV. It is a way to find and identify things in a digital video or image.

OpenCV is a library for Python that was made to improve computer vision. Face recognition is already built into OpenCV in the form of its FaceRecognizer class.

The method used here is pretty easy to understand:

- Taking pictures of the people whose faces we want to identify.
- Training the model with the images that have been collected.

- Feeding the model different pictures of faces is a way to test it.

OpenCV Built-in Face Recognizers

There are three built-in face recognizers in OpenCV, and you can use any of them by changing just one line of code. Here are the names and OpenCV functions to call each of these face recognizers.

- EigenFaces Face Recognizer Recognizer - cv.face.createEigenFaceRecognizer()
- FisherFaces Face Recognizer Recognizer - cv.face.createFisherFaceRecognizer()
- Local Binary Patterns Histograms (LBPH) Face Recognizer - cv.face.createLBPHFaceRecognizer()

Let me briefly explain how each one works:

EigenFaces Face Recognizer

This algorithm recognizes that not all facial features are equally important. You can recognize someone by his eyes, nose, cheeks, forehead, and how they differ. You're focusing on the face's areas of maximum change (variance). Eyes to nose and nose to mouth have significant differences. When comparing multiple faces, these parts are the most useful and vital.

EigenFaces face recognizer looks at all training images of all people as a whole and extracts important and useful components (those that catch the most variance/change), and discards the rest. It extracts important training data components and saves memory by discarding less important ones. It extracts principal components.

FisherFaces Face Recognizer

This algorithm is better than EigenFaces. Eigenfaces face recognizer looks at all training faces at once and combines principal components. This approach has drawbacks because some images can have sharp differences dominating the rest of the images, and you may end up with external features like light that isn't useful for discrimination.

Instead of extracting features that represent all faces, Fisherfaces algorithm extracts features that differentiate between people. So, one person's traits don't dominate the others, and you have distinguishing traits.

Local Binary Patterns Histograms (LBPH) Face Recognizer

Instead of looking at the whole image, find local features. LBPH compares each pixel to its neighbors to find an image's local structure.

Compare the center pixel of a 3x3 window with its neighbors at each image move. 1 denotes neighbors with less or equal intensity than a center pixel, 0 others. Then read these 0/1 values under a 3x3 window in clockwise order to get a local binary pattern like 11100011. Do this to the entire image to get local binary patterns.

LBPH will extract 100 histograms from the training data set and store them for later recognition. The algorithm remembers whose face is represented by the histogram.

When you feed a new image to the recognizer for recognition, it will generate a histogram, compare it to its histograms, find the best match, and return the person label associated with that histogram.

In this program, we will use 3 different OpenCV libraries that you must install.

They are os, NumPy and cv2. Os is the Python module we want to use to read our folders and file names to train the algorithm.

Collecting Images

The more pictures that are used to train the AI, the better. A face recognizer is usually trained with a lot of different pictures of the same person, so it can learn how they look when they are laughing, sad, happy, crying, with or without a beard, etc. So that you can easily follow this tutorial, we will only use 12 pictures for each person.

So our training data is made up of 12 pictures of each 2 different people (Eminem and Donald Trump). All of the data about training our program is in

the same folder I name train-data. This is important to keep all the data in a well organized directory for OpenCV to read easily. Each person has their own folder in the train-data folder. The folder for the 12 images of each is named accordingly, i.e., Eminem," “Donald Trump.”

So what I'm essentially going to do is we're going to use OpenCV's built-in face recognizer. And we're going to train that right now. So on all of these images in these folders, this is sort of like building a mini-sized, deep learning model, except that we're not going to build any model from scratch. We're going to use OpenCV's built-in face recognizer.

Preparing training data

And we're going to train that recognizer on the images. Open your Visual Studio Code and create a new file. Import os, cv2, and NumPy as NP.

So the first thing I want to do is essentially create a list of all the people in the images. Because we only have two, it is simple.

And what I'm going to do next is I'm essentially going to create a variable called DIR and set this equal to the base folder, that is, this train-data folder which contains the folders of the people. So with that done, what we can do is we can essentially create a function called def create_train, which will essentially loop over every folder in this base folder. And inside that folder, it's going to loop over every image and essentially grab the face in that image and add that to our training set.

So our training set will consist of two lines. The first one is called features, which are essentially the image arrays of faces. We set this to an empty list. And the second list will be our corresponding labels. So for every face in this features list, what is its corresponding label, and whose face it belongs to, one image could belong to Eminem, the second image could belong to Donald Trump, and so on.

```
import cv2 as cv
import os
import numpy as np

people = ["Donald Trump", "Eminem"]
DIR = r'C:\Users\Desktop\train-data'
```

```
haar_cascade = cv.CascadeClassifier('desktop\haar_face.xml')

features = []
labels = []

def create_train():
    for person in people:
        path = os.path.join(DIR, person)
        label = people.index(person)

        for img in os.listdir(path):
            img_path = os.path.join(path, img)

            img_array = cv.imread(img_path)
            gray = cv.cvtColor(img_array, cv.COLOR_BGR2GRAY)

            faces_rect = haar_cascade.detectMultiScale(gray, scaleFactor=1.1,
minNeighbors=4)

            for (x,y,w,h) in faces_rect:
                faces_roi = gray [y:y+h, x:x+w]
                features.append(faces_roi)
                labels.append(label)

create_train()

print(f'Length of Features = {len(features)}')
print(f'Length of Labels = {len(labels)}')
```

```
cv.waitKey()
```

As you can see in this code above, there is a mapping of data done. Let me explain:

The data that the OpenCV face recognizer accepts is in a specific format. It takes two vectors: one of all the persons' faces and the other of integer labels for each face so that when a face is processed, the face recognizer knows

which person that face belongs to.

That is why we need a list for labels and features.

In the function we created, we used a for loop to make OpenCV read the images in the folders we have prepared for the training set. And we used the face detection code we learned in the previous chapter in the following line.

And now that we have a faces region of interest, I append that to the features list. And we can append the corresponding label to the labels list. The idea behind converting a label to numerical values is to reduce the strain that your computer will have by creating some sort of mapping between a string and the numerical label. Now the mapping we will do is the index of that people list. So let's say that I grab the first image, which is an image of Eminem. The label for that would be 0 because Eminem is at the zeroeth index of this people list. Then, Donald Trump would have a label of 1 because it is at the second position or the first index in this people's list.

After you run the script, you should get a list of features and labels in the Terminal window like this:

```
Length of Features = 26
Length of Labels = 26
```

We get the length of the features 26 and the length of labels 26. That is to say, OpenCV is reading 26 faces and 26 corresponding labels to these faces.

Now that we have prepared the data and are sure that OpenCV can read the folders we have for this training, we can start to train the face recognizer so that it can recognize the faces of the two people we have trained it on.

Training The Face Recognizer

We will use the features and labels list now that it's appended to train our recognizer on it. How do we do that? We will first instantiate our face recognizer.

I'll use the LBPH face recognizer, but you can use any other face recognizer you want. The code will remain the same regardless of which OpenCV face recognizer you use. Just as we have explained earlier, only one line, the face recognizer initialization line, needs to be changed.

cv2.face.createLBPHFaceRecognizer()

This new function will essentially instantiate the face recognizer. Now we can train the recognizer on the features list and the labels and the labels list. We will do this with a method we call the line train(faces-vector, labels-vector) from the face recognizer.

So the way we do that is by saying face_recognizer.train, and we can pass in the features list and pass in the labels list. And before we do that, we should convert this features and labels list to NumPy arrays. This is because OpenCV expects the labels vector to be a NumPy array.

```
create_train()
print('Training Done ----------')

features = np.array(features, dtype=object)
labels = np.array(labels)

face_recognizer = cv.face.LBPHFaceRecognizer_create()
face_recognizer.train(features, labels)
```

```
np.save('features.npy', features)
np.save('labels.npy', labels)
```

This is the code to train our Face Recogniser. So let's save that and run cool. So essentially, the face recognizer is trained, and we can now use this.

But the problem here is that if we plan to use this face recognizer in another file, we'll have to separately and manually repeat this process, this whole process of adding those images to a list and getting the corresponding labels, and then converting that to NumPy arrays, and then training all over again. Fortunately, openCV allows us to save this trained model so that we can use it in another file in another directory in another part of the world just by using that particular YML source file.

We will do that by adding the following line to the code right before the np.save method:

```
face_recognizer.save('trained_faces.yml')
```

All we have to do is save the code and run it. You will be able to see the new YML file in your OpenCV directory, faces, and features.nPy and labels.nPy.

Now, we can use this trained model to recognize faces in any image. This time, we will need a new Python file. This is the file for the script of the images we want to read and recognize the faces of Eminem and Donald Trump.

Face Recognition Testing

Now, the fun part. We will need to import all the modules for the program:

We need Numpy and cv2. We don't need os anymore because we're not looping over directories and training anything. Look at the code below:

```
import numpy as np
import cv2 as cv

haar_cascade = cv.CascadeClassifier('haar_face.xml')

people = ["Donald Trump", "Eminem"]
features = np.load('features.npy')
labels = np.load('labels.npy')

face_recognizer = cv.face.LBPHFaceRecognizer_create()
face_recognizer.read ('trained_faces.yml')

img = cv.imread(r'C:\Users\images\images.jpg')

gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
cv.imshow('Person', gray)
```

What we're going to do is we're going first to detect the face in the image.

```
import numpy as np
import cv2 as cv

haar_cascade = cv.CascadeClassifier('haar_face.xml')

people = ["Donald Trump", "Eminem"]
# features = np.load('features.npy')
# labels = np.load('labels.npy')

face_recognizer = cv.face.LBPHFaceRecognizer_create()
face_recognizer.read ('trained_faces.yml')

img = cv.imread(r'C:\Users\Desktop\images\images.jpg')

gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
cv.imshow('Person', gray)

#detect face
faces_rect = haar_cascade.detectMultiScale(gray, 1.1, 4)

for (x,y,w,h) in faces_rect:
    faces_roi = gray [y:y+h, x:x+h]

    label, confidence = face_recognizer.predict(faces_roi)
    print(f'Labe1 = {people[label]} with a confidence of {confidence}')
    cv.putText(img, str(people[label]), (20,20), cv.FONT_HERSHEY_COMPLEX, 1.0,
(0,255,0), thickness=2)
    cv.rectangle(img, (x,y), (x+w, y+h), (0,255,0), 2)

cv.imshow('Detected', img)

cv.waitKey(0)
```

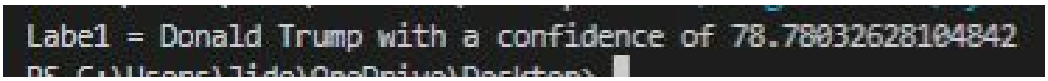
In the above code, you can see that we used the haar cascade and a for loop to loop over every face in the faces_rect variable. Next, We grab the region of interest to what we're interested in finding. And now, we predict using this face recognizer to get the label and a confidence value.

Using the face_recognizer.predict method, we predict on this faces_roi and print the value of label and confidence.

Next, we put some text on the image to show us who it is from the labels we have provided. We give it a font face of cv.FONT_HERSHEY_COMPLEX, a font scale of one point of 1.0 and color of 0, 255,0 and a thickness of 2. Then we tell it to draw a rectangle over the face.

In the above code, you see how I commented out labels and features.npy. This is because we do not necessarily need them anymore. However, if you want to use them since the data types are objects, you can add allow_pickle = true to the line of the code.

This is the result.



It says That it is Donald Trump with the confidence of 78%. You can go back and try different pictures that are essentially not part of the images you used in training or the train-data folder.

For this chapter, we discussed face recognition. In OpenCV, we essentially build a features list and a labels list, and we train a recognizer on those two lists. And we saved a model as a YML source file. In another file, we essentially read in that saved model saved YML source file. And we essentially make predictions on an image.

Face recognition is an exciting project to work on, and OpenCV has made it incredibly simple to code. A fully functional face recognition application is

only a few lines of code away, and we can switch between all three face recognizers with a single line of code change. That's all there is to it.

And so in the next chapter, which will be the last chapter in this course, we will discuss how to build a deep learning model to detect and classify between Simson characters.

Chapter 17 – Capstone - Computer Vision Project: The Simpsons

Hey, everyone, and welcome to the last chapter in this Python and OpenCV course. Previously, we've seen how to detect and recognize faces in OpenCV, and the results were varied. Now, there are a couple of reasons for that. One is due to the fact that we only had 26 images to train the recognizer on. This is a significantly small number, especially when you're training recognizes and building models.

Ideally, you'd want to have at least a couple of 1000 images per class or per person. The second reason is that we weren't using a deep learning model. Now, as you go deeper into the field of computer vision, you will see that it is important to develop a deep learning model. So that's what we're going to be doing in this chapter.

Building a deep computer vision model to classify the movie characters in OpenCV requires some complex processes for preprocessing the data. This involves several calculations, performing some sort of image normalization, mean, subtraction, and things like that.

But in this chapter, we're going to be building a very simple model. So we're not going to be using any of those techniques. We'll only be using the OpenCV library to read an image and resize them to a particular size before feeding it into the network.

Now, don't worry if you've never used a built a deep learning model before. But this chapter will be using tensor flows implementation and Keras. now I want to keep this chapter real simple so that you have an idea of what goes on in more advanced computer vision projects. And Keras comes with a lot of boilerplate code. So if you've never built a deep learning model before, don't worry. Keras will handle that for you.

Another one of the prerequisites to building a deep learning model is having a GPU. Now GPU is a graphical processing unit that will help speed up the training process of a network. But if you don't have one, again, don't worry because we'll be using Kaggle, a platform that offers free GPUs for us to use.

Setting Up

So real simple, we need a couple of packages installed before we get started.

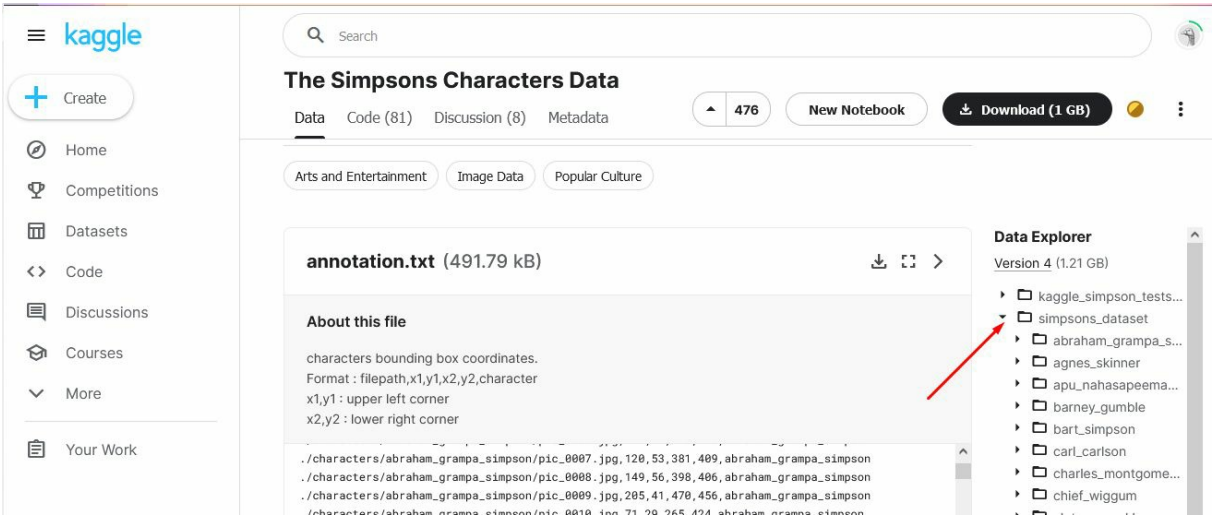
- Caer Library
- Canaro
- Tensorflow

First of all, you need Caer. So if you haven't already installed Caer at the beginning of this course, go ahead and do a pip install caer in your cmd or Terminal.

The next package you require is Canaro. It is an image processing library for computer vision in Python that uses the Keras framework for deep learning models, and it is developed by OpenCV's community of developers. It will appear surprisingly valuable for you if you plan to go deeper into building deep computer vision models. Now, installing this package on your system will only make sense if you already have a GPU on your machine. If you don't, then you can skip this part. And canaro installs TensorFlow by default, so just keep that in mind.

Getting Data

So with all the installations out of the way, let's move on to the data that we're going to be using. So the data set that we're going to be using is the Simpsons character data set available on Kaggle (Follow this link to get it). You will have to sign in or register on Kaggle to be able to use the dataset. Go to the sidebar, and find Data Explorer.



The actual data that we're interested in lies in the simpsons_dataset folder. This basically consists of a number of folders with several images inside each subfolder. As you can see, Maggie Simpson has about 128 images. Homer Simpson has about 2200 images. Abraham has about 913 images.

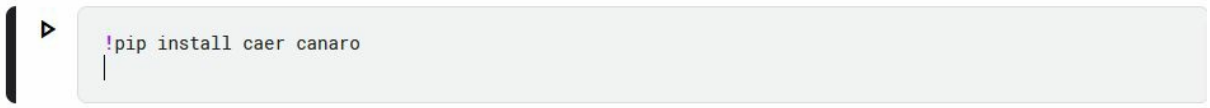
So essentially, what we're going to do is we are going to use these images and feed them into our model to classify between these characters essentially.

So the first thing we want to do is go to kaggle.com/notebooks, go ahead and click on create a new notebook. And under Advanced Settings, make sure that the GPU is selected since we're going to be using the GPU and click Create. If you are a new Kaggle user, you may have to verify your phone number to be able to use the GPU accelerator service. After you have done all the necessary details, you should get a notebook.

Rename your notebook to Simpsons. Also, remember to toggle on the Internet option on the sidebar since we're installing a couple of packages over the internet in Kaggle.

So to use the Simpsons character data set in our notebook, you need to go head to the top right-hand side of the page and click on add data. Search for Simpsons. And the first result by Aleaxattia should pop up, go ahead and click Add on that one. And we can now use this data set inside a notebook.

Next, clear the editor and type in pip install caer canaro again in Kaggle. And the reason why we will need to do this yet again is that Kaggle does not come preinstalled with Caer and Canaro. And click on the play icon in front of the editor.



Now I did tell you to install it on your machine is so that you can work with it and experiment with it. So once that's done, go ahead to a new cell. And let's import all the packages that we're going to need.

import os

import caer

import canaro

import numpy as np

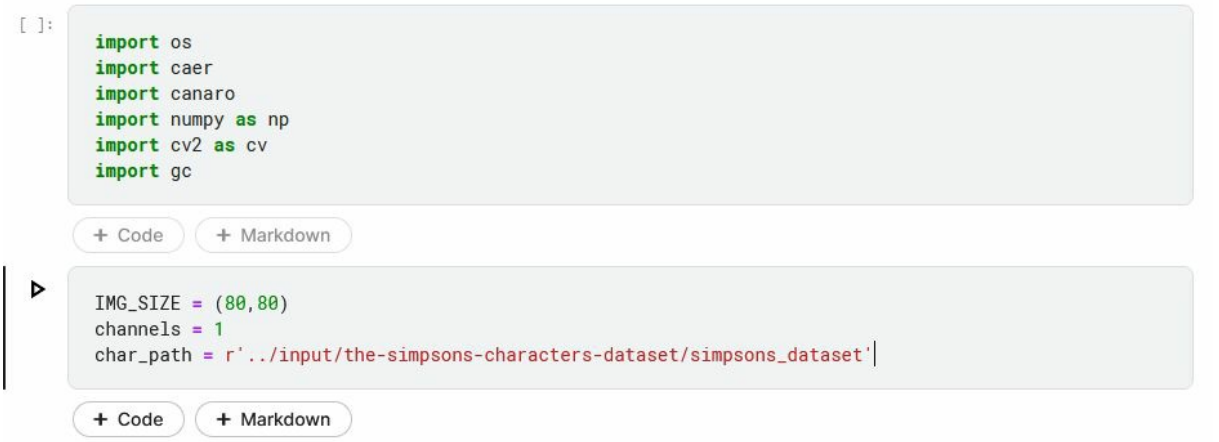
import cv2 as cv

import gc

GC is for garbage collection.

All new lines of code are in a new cell. Then next, what we want to do is basically, when building deep computer vision models, your model expects all your data or your image data to be of the same size. So since we're working with image data, this size is the image size. So all the data or the images in our data set will have to be resized to a particular size before we can feed that into the network.

Now, with many experiments, I found that an image size of 80 by 80 works well, especially for this Simpsons data set. The next variable we need is the channels. So how many channels do we want in our image? And since we do not require color in our image, we're going to set this to 1, that is, grayscale.



The last line in that cell is to get the path of the data set we want to use in the learning process. That is the base path where all the data is, and that is in this simpsons_dataset. This is the base folder for where all our images are stored in. So go ahead and hover over the folder and copy the file path. You can then paste in the code next to the r.

So essentially, what we're going to be doing now is going to grab the top 10 characters with the most number of images for that class. And the way we're

going to do that is we are going to go through every folder inside the simpsons_dataset, get the number of images that are stored in that data set, store all of that information inside a dictionary so that dictionary in descending order, and then grab the first 10 elements in the dictionary.

So what we're going to do is we're going to say create an empty dictionary. Using a for loop, we're going to say for each character in our list, join the char_pump with char. So essentially, all that we're doing is going through every folder or grabbing the name of the folder, and we're getting the number of images in that folder. And we're storing all that information inside the dictionary called char_dict.

```
char_dict ={}

for char in os.listdir(char_path):

    char_dict[char] = len(os.listdir(os.path.join(char_path, char)))
```

Once that's done, we can sort this dictionary in descending order and the way we do that is with a caer.sort_dict function. Write the following code next:

```
char_dict ={}

for char in os.listdir(char_path):

    char_dict[char] = len(os.listdir(os.path.join(char_path, char)))


char_dict = caer.sort_dict(char_dict, descending=True)

char_dict
```

```
[('homer_simpson', 2246),
 ('ned_flanders', 1454),
 ('moe_szyslak', 1452),
 ('lisa_simpson', 1354),
 ('bart_simpson', 1342),
 ('marge_simpson', 1291),
 ('krusty_the_clown', 1206),
 ('principal_skinner', 1194),
 ('charles_montgomery_burns', 1193),
 ('milhouse_van_houten', 1079),
 ('chief_wiggum', 986),
 ('abraham_grampa_simpson', 913),
 ('sideshow_bob', 877),
 ('apu_nahasapeemapetilon', 623),
 ('kent_brockman', 498),
 ('comic_book_guy', 469),
 ('edna_krabappel', 457),
 ('nelson_muntz', 358),
 ('lenny_leonard', 310),
 ('mayor_quimby', 246),
 ('waylon_smithers', 181),
 ('maggie_simpson', 128),
 ('groundskeeper_willie', 121),
 ('barney_gumble', 106),
 ('selma_bouvier', 103),
 ('carl_carlson', 98),
 ('ralph_wiggum', 89),
 ('patty_bouvier', 72),
 ('martin_prince', 71),
 ('professor_john_frink', 65),
 ('snake_jailbird', 55),
 ('cletus_spuckler', 47),
 ('rainier_wolfcastle', 45),
 ('simpsons_dataset', 42),
 ('agnes_skinner', 42),
```

So it returned the dictionary that we have, and it is an extensive list in descending order. As you can see, Homer Simpson has the most number of

images at close to 2300. And we go all the way down to Lionel, who has only three images in the data.

So what we're going to do is now that we have this dictionary, what we're going to do is we are going to grab the names of the first 10 elements in this dictionary and store that in a list of characters list. Use the following code:

```
characters = []

count = 0

for i in char_dict:

    characters.append(i[0])

    count += 1

    if count >= 10:

        break
```

So with the above code, we have essentially grabbed the names of the characters. So with that done, we can go ahead and create the training data.

Training Data

And to create a training data is as simple as creating a new variable called train or whatever you like and adding the following as in the code below:

```
train = caer.preprocess_from_dir(char_path, characters, channels=channels,
IMG_SIZE=IMG_SIZE, isShuffle=True)
```

So essentially, what this will do is it will go through every folder inside char_path, which is Simpsons_dataset. And look at every element inside the characters dictionary. So essentially, it is going to look for Homer Simpson inside the Simpsons_dataset. It will find Homer Simpson, go to that folder, grab all the images inside that folder, and essentially add them to our training set.

As you may recall, in the previous chapter, a training dataset was essentially a list. Each element in that list was another list of the image and the corresponding label.

The label that we had was the index of that particular string in the characters list. So that's essentially the same type of mapping that we're going to use. So Homer Simpson will have a label of zero, Ned will have one, Liza will have a label of three, and so on. So once that's done, go ahead and run this. Now, the progress is displayed at the terminal.

Because there are a lot of images inside this data set, this may take a while, depending on how powerful your machine is.

Now, here is something we can do. We can make the system show how many images there are in this training set. And we have 13,811 images inside this training set.

Now, we can try to visualize the images that are present in this dataset. For that, we will use matplotlib. Go ahead and import that to your script. Add these lines in a new cell:

```
import matplotlib.pyplot as plt

plt.figure(figsize=(30,30))

plt.imshow(train [0][0], cmap='gray')

plt.show()
```

And we can display the image in grayscale.

```
import matplotlib.pyplot as plt
plt.figure(figsize=(30,30))
plt.imshow(train [0][0], cmap='gray')
plt.show()
```

Now I'm not using openCV to display this image because, for some reason, openCV does not display correctly in Jupyter Notebook. So that's why we're using matplotlib.

Looking at that image, you may feel that it is blurry, and you cannot identify the object, but to a machine, that is a valid image.

Features and Labels

The next thing we want to do is we want to separate the training set into the features and labels. Right now. The train data is a list of 13,811 lists inside it. Inside each of the sub lists are two elements, the actual array and the labels themselves.

So we're going to separate the feature set, or the arrays and the labels into separate lists. We can do that with the code below:

featureSet, labels = caer.sep_train(train, IMG_SIZE=IMG_SIZE)

So basically, this will separate the training set into the feature set and labels and reshape this feature set into a four-dimensional tensor so that it can be fed into the model with no restrictions whatsoever. So go ahead and run that. And once that's done, let's try to normalize the feature sets.

Normalize FeatureSet

Now, we want to normalize the data to be in the range of to be the range of 0,1. And the reason for this is that if you normalize the data, the network will be able to learn the features much faster than when they are not normalized.

Use the following code:

```
from tensorflow.keras.utils import to_categorical

featureSet = caer.normalize(featureSet)

labels = to_categorical(labels, len(characters))
```

We, first of all, used the caer.normalize() method to normalize the featureSet. Now we don't have to normalize the labels. But we need to one hot encode them that converts them from numerical integers to binary class vectors. And the way we do that is by using the to_categorical method from TensorFlow.Keras.utils.

We get the possible labels and the number of categories, which is the length of this characters list.

Create Training & Validation Data

Once that's done, we can actually move ahead and try to create our training and validation data. Don't worry too much if you don't know what these are. The model is going to train on the training data and test itself on the validation data.

```
x_train, x_val, y_train, y_val = caer.train_val_split(featureSet, labels, val_ratio=.2
```

With this code, we're splitting the feature set and labels into training sets and validation sets with using a particular validation ratio of 20% of this data will go to the validation set, and 80% will go to the training set.

Now, just to save on some memory, we can remove and delete some of the variables we're not going to be using.

Type in this code to remove the variables and collect the garbage.

```
del train
```

del featureSet

del labels

gc.collect()

Image Data Generator

Now moving on, we need to create an image data generator. This is an image generator that will essentially synthesize new images from existing images to help introduce some randomness to our network and make it perform better. We will use the following code:

datagen = canaro.generators.imageDataGenerator()

train_gen = datagen.flow(x_train, y_train, batch_size=BATCH_SIZE)

So with that done, we can proceed to build our model.

Creating The Model

There are many ways and models to use, but I am using one I have found to provide the highest level of accuracy. So that's the same model, the same model architecture that we're going to be using.

model = canaro.models.createSimpsonsModel(IMG_SIZE=IMG_SIZE, channels=channels, output_dims=len(characters), loss='binary_crossentropy',decay=1e-6, learning_rate=0.001, momentum=0.9, nesterov=True)

So this will essentially create the model using this architecture and compile

the model so that we can use it. When you run that, you can print a summary of the model by typing model.summary().

```
model.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_30 (Conv2D)           (None, 80, 80, 32)        320
_________________________________________________________________
conv2d_31 (Conv2D)           (None, 78, 78, 32)        9248
_________________________________________________________________
max_pooling2d_15 (MaxPooling (None, 39, 39, 32)        0
_________________________________________________________________
dropout_20 (Dropout)         (None, 39, 39, 32)        0
_________________________________________________________________
conv2d_32 (Conv2D)           (None, 39, 39, 64)        18496
_________________________________________________________________
conv2d_33 (Conv2D)           (None, 37, 37, 64)        36928
_________________________________________________________________
max_pooling2d_16 (MaxPooling (None, 18, 18, 64)        0
_________________________________________________________________
dropout_21 (Dropout)         (None, 18, 18, 64)        0
_________________________________________________________________
conv2d_34 (Conv2D)           (None, 18, 18, 256)       147712
_________________________________________________________________
conv2d_35 (Conv2D)           (None, 16, 16, 256)       590080
_________________________________________________________________
max_pooling2d_17 (MaxPooling (None, 8, 8, 256)         0
_________________________________________________________________
dropout_22 (Dropout)         (None, 8, 8, 256)         0
_________________________________________________________________
flatten_5 (Flatten)          (None, 16384)             0
_________________________________________________________________
dropout_23 (Dropout)         (None, 16384)             0
_________________________________________________________________
dense_7 (Dense)              (None, 1024)              16778240
```

And so essentially, what we have is a functional model since we're using Keras' functional API. And this essentially has a bunch of layers and about 17 million parameters to train on.

So another thing that I want to do is create something called a callbacks list. Now, this callbacks list will contain something called a learning rate schedule that will essentially schedule the learning rate at specific intervals so that our network can effectively train better.

Type the following code in a new cell:

```
from tensorflow.keras.callbacks import LearningRateScheduler

callbacks_list = [LearningRateScheduler(canaro.lr_schedule)]
```

Training The Model

So let's actually go ahead and train the model. We will use the following code:

```
training = model.fit(train_gen,
                     steps_per_epoch=len(x_train)//BATCH_SIZE,
                     epochs=EPOCHS,
                     validation_data=(x_val,y_val),
                     validation_steps=len(y_val)//BATCH_SIZE,
                     callbacks=callbacks_list)
```

```
Epoch 1/10
345/345 [==============================] - 19s 34ms/step - loss: 0.3416 - accuracy: 0.1512 - val_loss: 0.3313 - val_accurac
y: 0.1624
Epoch 2/10
345/345 [==============================] - 9s 26ms/step - loss: 0.3236 - accuracy: 0.1615
Epoch 3/10
345/345 [==============================] - 9s 26ms/step - loss: 0.3236 - accuracy: 0.1581
Epoch 4/10
345/345 [==============================] - 9s 27ms/step - loss: 0.3229 - accuracy: 0.1623
Epoch 5/10
345/345 [==============================] - 9s 27ms/step - loss: 0.3199 - accuracy: 0.1715
Epoch 6/10
345/345 [==============================] - 9s 26ms/step - loss: 0.3087 - accuracy: 0.2229
Epoch 7/10
345/345 [==============================] - 10s 28ms/step - loss: 0.3040 - accuracy: 0.2413
Epoch 8/10
345/345 [==============================] - 9s 27ms/step - loss: 0.3003 - accuracy: 0.2641
Epoch 9/10
345/345 [==============================] - 9s 27ms/step - loss: 0.2971 - accuracy: 0.2724
Epoch 10/10
345/345 [==============================] - 9s 26ms/step - loss: 0.2955 - accuracy: 0.2845
```

And with that done, we end up with a baseline accuracy of close to 70%. So here comes the exciting part, we're now going to use OpenCV to test how good our model is. So what we're going to do is we're going to use OpenCV to read in an image at a particular file path. And we're going to pass that to our network and see what the model spits out.

Testing and Predicting

There is a test set in the Simpson data set we imported to the notebook. Go to that test set and pick one image. We can start with Bart Simpson. Pick any of the images and copy the file path. Implement the following code:

```
test_path = r'../input/the-simpsons-characters-dataset/kaggle_simpson_testset/kaggle_simpson_testset/bart_simpson_27.jpg'

img = cv.imread(test_path)


def prepare(img):
    img = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
    img = cv.resize(img, IMG_SIZE)
    img = caer.reshape(img, IMG_SIZE, 1)
    return img
```

So what this code did is to create a function called prepare, which will prepare our image to be of the same size and shapes and dimensions as the images we use to prepare the model. So this will take in a new image. And what this will do is we'll, we'll convert this image to grayscale

Now, we want to run the predictions clause.

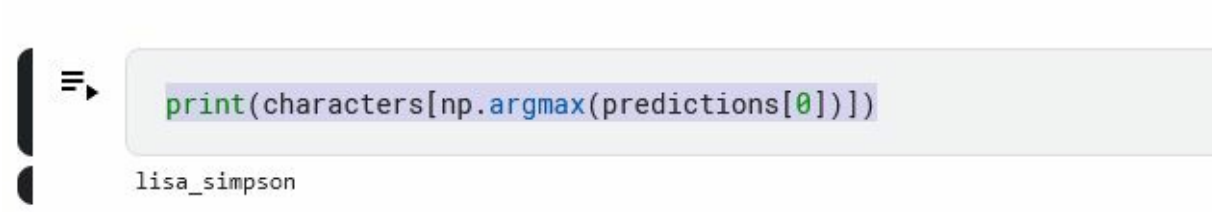
```
predictions = model.predict(prepare(img))
```

And we can visualize this predictions by typing predictions.

```
print(characters[np.argmax(predictions[0])])
```

This code is to print the actual class, that is, the name of the character in the image we have picked.

Our model thinks that Bart Simpson is Lisa Simpson when I run that. Too bad.



Let's try another image of Charles Montgomery. What did you get? Did you get it to tell you that was Van Hughton? Not the best model that we could have asked for. This is a model. Right now, this base discounting has a

baseline accuracy of 70%. Although I would have liked it to go to at least 85%. But again, this is to be expected.

So, when building your model, ensure that it is around 0.9000000 for 90% accuracy in the training.

Building deep computer vision models is a bit of an art. And it takes time to figure out the best model for your project.

So that's it for this Python and OpenCV course. While we obviously can't cover everything that OpenCV can do, you have learned what's relevant today in computer vision. And really, one of its most exciting parts is building deep learning models, which is, in fact, where the future is self-driving vehicles, medical diagnosis, and tons of other things that computer vision is changing the world.

End Game

You can use Opencv in a variety of computer vision applications such as video and CCTV footage analytics and image processing. More than 2,500 optimized algorithms make up OpenCV, written in Python. We can use this library to focus on real-world problems when developing computer vision applications that don't need to be built from the ground up. Many companies, including Google, Amazon, Microsoft, and Toyota, currently make use of this library. Many scientists and programmers contribute to this effort. It's simple to set up on a variety of operating systems, including Windows, Ubuntu, and macOS.

Using image processing, you can enhance or extract information from an image by applying various operations to it.

"Image processing is the analysis and manipulation of a digitized image, especially in order to improve its quality," according to the most basic definition.

The pixel value describes how bright and what color the pixel should be in the image. In other words, an image is nothing more than a two-dimensional matrix (3-D in the case of colored images) defined by the mathematical function f(x, y).

Image analysis and processing can be done with OpenCV. This means that it is excellent at extracting information from video or a stereoscopic camera's two images and running algorithms to do so. For example, if you wanted to capture images and track a specific object as it moved around, you could use OpenCV. Mathematical tools that can extract this information are available, but they are not provided directly. Even though it can do other things, such as stretch an image or change its color, its primary function is not to serve as an image processing engine like Photoshop or something similar. To be as fast as possible, it should be able to perform all kinds of functions like Fourier transforms in near-real time if the hardware supports it, and then let you either extract information from or transform the stream of images as you wish.

When it comes to video/image analysis and processing, it can be used for

everything from facial recognition to tracking objects to determining their distance from stereoscopic inputs and more. There are no ready-made solutions here, but the mathematical tools are there for you to use.