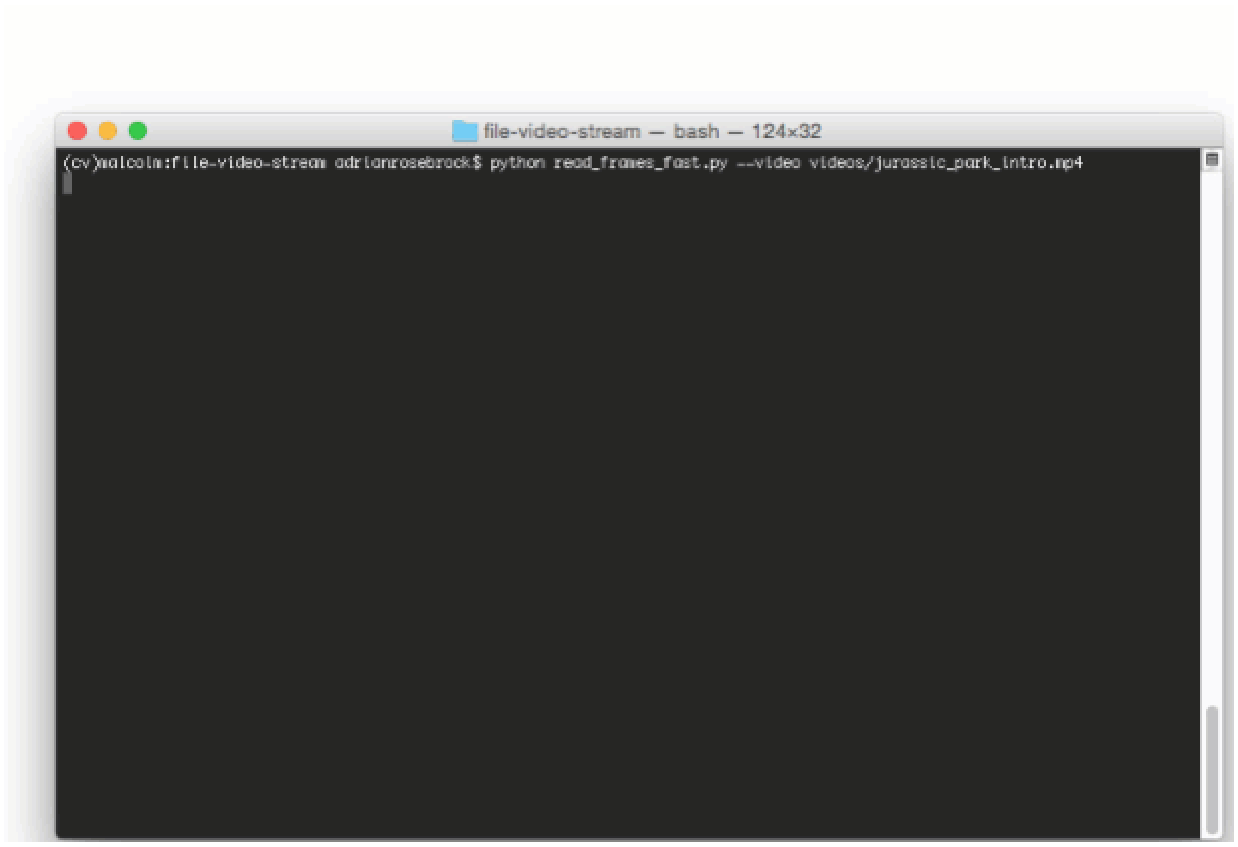
Faster video file FPS with cv2.VideoCapture and OpenCV

pyimagesearch.com/2017/02/06/faster-video-file-fps-with-cv2-videocapture-and-opencv

Adrian Rosebrock

February 6, 2017



```
(cv)malcolm:file-video-stream adrianrosebrock$ python read_frames_fast.py --video videos/jurassic_park_intro.mp4
```

Have you ever worked with a video file via OpenCV's

cv2.VideoCapture
 function and found that reading frames ***just felt slow and sluggish?***
I've been there — *and I know exactly how it feels.*

Your entire video processing pipeline crawls along, unable to process more than one or two frames per second — even though you aren't doing any type of computationally expensive image processing operations.

Why is that?

Why, at times, does it seem like an *eternity* for

cv2.VideoCapture
 and the associated
.read
 method to poll another frame from your video file?
The answer is almost always ***video compression and frame decoding.***

Depending on your video file type, the codecs you have installed, and not to mention, the physical hardware of your machine, much of your video processing pipeline can actually be consumed by ***reading*** and ***decoding*** the next frame in the video file.

That's just computationally wasteful — and there is a better way.

In the remainder of today's blog post, I'll demonstrate how to use threading and a queue data structure to ***improve your video file FPS rate by over 52%!***

Looking for the source code to this post?

Jump Right To The Downloads Section →
When working with video files and OpenCV you are likely using the

cv2.VideoCapture
 function.
First, you instantiate your

cv2.VideoCapture
 object by passing in the path to your input video file.
Then you start a loop, calling the

.read
 method of
cv2.VideoCapture
 to poll the next frame from the video file so you can process it in your pipeline.
The *problem* (and the reason why this method can feel slow and sluggish) is that you're ***both reading and decoding the frame in your main processing thread!***

As I've mentioned in previous posts, the

.read
 method is a *blocking operation* — the main thread of your Python + OpenCV application is entirely blocked (i.e., stalled) until the frame is read from the video file, decoded, and returned to the calling function.
By moving these blocking I/O operations to a separate thread and maintaining a queue of decoded frames ***we can actually improve our FPS processing rate by over 52%!***

This increase in frame processing rate (and therefore our overall video processing pipeline) comes from *dramatically reducing latency* — we don't have to wait for the

.read
method to finish reading and decoding a frame; instead, there is *always* a pre-decoded frame ready for us to process.
To accomplish this latency decrease our goal will be to move the reading and decoding of video file frames to an entirely separate thread of the program, freeing up our main thread to handle the actual image processing.

But before we can appreciate the *faster, threaded* method to video frame processing, we first need to set a benchmark/baseline with the slower, non-threaded version.

The slow, naive method to reading video frames with OpenCV

The goal of this section is to obtain a baseline on our video frame processing throughput rate using OpenCV and Python.

To start, open up a new file, name it

read_frames_slow.py
, and insert the following code:
Faster video file FPS with cv2.VideoCapture and OpenCV

```
# import the necessary packages
from imutils.video import FPS
import numpy as np
import argparse
import imutils
import cv2
# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-v", "--video", required=True,
help="path to input video file")
args = vars(ap.parse_args())
# open a pointer to the video stream and start the FPS timer
stream = cv2.VideoCapture(args["video"])
fps = FPS().start()
```

Lines 2-6 import our required Python packages. We'll be using my imutils library, a series of convenience functions to make image and video processing operations easier with OpenCV and Python.

If you don't already have

imutils
installed ***or*** if you are using a previous version, you can install/upgrade
imutils
by using the following command:
Faster video file FPS with cv2.VideoCapture and OpenCV

```
$ pip install --upgrade imutils
```

Lines 9-12 then parse our command line arguments. We only need a single switch for this script,

--video
, which is the path to our input video file.
Line 15 opens a pointer to the

--video
file using the
cv2.VideoCapture
class while **Line 16** starts a timer that we can use to measure FPS, or more specifically, the throughput rate of our video processing pipeline.
With

cv2.VideoCapture
instantiated, we can start reading frames from the video file and processing them one-by-one:
Faster video file FPS with cv2.VideoCapture and OpenCV

```
# loop over frames from the video file stream
while True:
# grab the frame from the threaded video file stream
(grabbed, frame) = stream.read()
# if the frame was not grabbed, then we have reached the end
# of the stream
if not grabbed:
break
# resize the frame and convert it to grayscale (while still
# retaining 3 channels)
frame = imutils.resize(frame, width=450)
frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
frame = np.dstack([frame, frame, frame])
# display a piece of text to the frame (so we can benchmark
# fairly against the fast method)
cv2.putText(frame, "Slow Method", (10, 30),
cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 255, 0), 2)
# show the frame and update the FPS counter
cv2.imshow("Frame", frame)
cv2.waitKey(1)
fps.update()
```

On **Line 19** we start looping over the frames of our video file.

A call to the

.read
method on **Line 21** returns a 2-tuple containing:

1. grabbed
   : A boolean indicating if the frame was successfully read or not.
2. frame
   : The actual video frame itself.

If

grabbed
 is
False
 then we know we have reached the end of the video file and can break from the loop (**Lines 25 and 26**).
Otherwise, we perform some basic image processing tasks, including:

1. Resizing the frame to have a width of 450 pixels.
2. Converting the frame to grayscale.
3. Drawing the text on the frame via the
   cv2.putText
    method. We do this because we'll be using the
   cv2.putText
    function to display our queue size in the fast, threaded example below and want to have a fair, comparable pipeline.

Lines 40-42 display the frame to our screen and update our FPS counter.

The final code block handles computing the approximate FPS/frame rate throughput of our pipeline, releasing the video stream pointer, and closing any open windows:

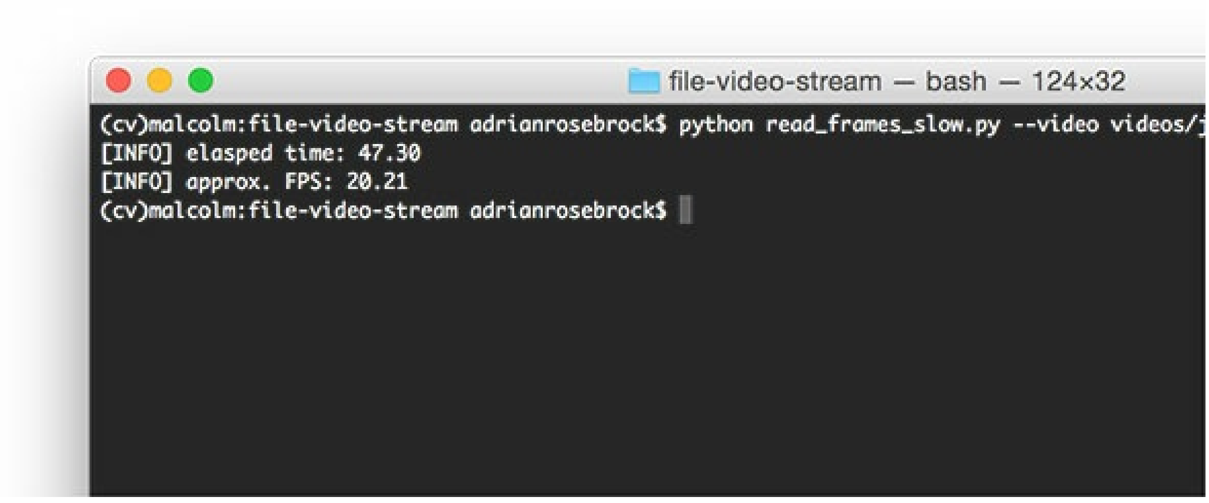
Faster video file FPS with cv2.VideoCapture and OpenCV
```
# stop the timer and display FPS information
fps.stop()
print("[INFO] elasped time: {:.2f}".format(fps.elapsed()))
print("[INFO] approx. FPS: {:.2f}".format(fps.fps()))
# do a bit of cleanup
stream.release()
cv2.destroyAllWindows()
```
To execute this script, be sure to download the source code + example video to this blog post using the ***"Downloads"*** section at the bottom of the tutorial.

For this example we'll be using the ***first 31 seconds*** of the *Jurassic Park* trailer (the .mp4 file is included in the code download):

Watch Video At: https://youtu.be/lc0UehYemQA

Let’s go ahead and obtain a baseline for frame processing throughput on this example video:

Faster video file FPS with cv2.VideoCapture and OpenCV
$ python read_frames_slow.py --video videos/jurassic_park_intro.mp4



```
(cv)malcolm:file-video-stream adrianrosebrock$ python read_frames_slow.py --video videos/j
[INFO] elasped time: 47.30
[INFO] approx. FPS: 20.21
(cv)malcolm:file-video-stream adrianrosebrock$ 
```

Figure 1: The slow, naive method to read frames from a video file using Python and OpenCV.

As you can see, processing *each individual frame* of the 31 second video clip takes approximately 47 seconds with a FPS processing rate of 20.21.

These results imply that it’s actually taking *longer* to read and decode the individual frames than the actual length of the video clip!

To see how we can speedup our frame processing throughput, take a look at the technique I describe in the next section.

Using threading to buffer frames with OpenCV

To improve the FPS processing rate of frames read from video files with OpenCV we are going to utilize *threading* and the queue data structure:

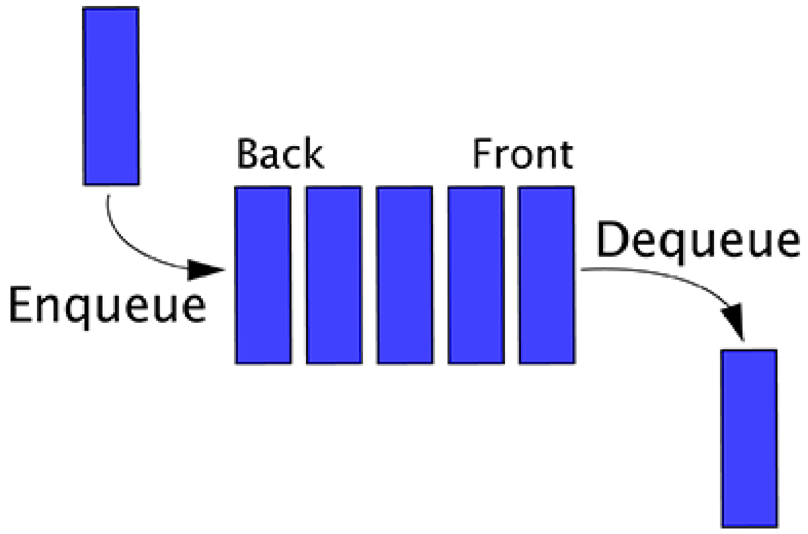


Figure 2: An example of the queue data structure. New data is enqueued to the back of the list while older data is dequeued from the front of the list. (source: Wikipedia)

Since the

.read
 method of
cv2.VideoCapture
 is a blocking I/O operation we can obtain a significant speedup simply by creating a *separate thread* from our main Python script that is solely responsible for reading frames from the video file and maintaining a queue.
Since Python's Queue data structure is thread safe, much of the hard work is done for us already — we just need to put all the pieces together.

I've already implemented the FileVideoStream class in imutils but we're going to review the code so you can understand what's going on under the hood:

Faster video file FPS with cv2.VideoCapture and OpenCV

```
# import the necessary packages
from threading import Thread
import sys
import cv2
# import the Queue class from Python 3
if sys.version_info >= (3, 0):
from queue import Queue
# otherwise, import the Queue class for Python 2.7
else:
from Queue import Queue
```

Lines 2-4 handle importing our required Python packages. The

Thread
 class is used to create and start threads in the Python programming language.
We need to take special care when importing the

Queue
 data structure as the name of the queue package is different based on which Python version you are using (**Lines 7-12**).
We can now define the constructor to

FileVideoStream
 :
Faster video file FPS with cv2.VideoCapture and OpenCV

```
class FileVideoStream:
def __init__(self, path, queueSize=128):
# initialize the file video stream along with the boolean
# used to indicate if the thread should be stopped or not
self.stream = cv2.VideoCapture(path)
self.stopped = False
# initialize the queue used to store frames read from
# the video file
self.Q = Queue(maxsize=queueSize)
```

Our constructor takes a single required argument followed by an optional one:

- path
 : The path to our input video file.
- queueSize
 : The maximum number of frames to store in the queue. This value defaults to 128 frames, but you depending on (1) the frame dimensions of your video and (2) the amount of memory you can spare, you may want to raise/lower this value.

Line 18 instantiates our

cv2.VideoCapture
 object by passing in the video
path
 .
We then initialize a boolean to indicate if the threading process should be stopped (**Line 19**) along with our actual

Queue
 data structure (**Line 23**).
To kick off the thread, we'll next define the

start
 method:

Faster video file FPS with cv2.VideoCapture and OpenCV

```
def start(self):
# start a thread to read frames from the file video stream
t = Thread(target=self.update, args=())
t.daemon = True
t.start()
return self
```

This method simply starts a thread *separate* from the main thread. This thread will call the

.update
 method (which we'll define in the next code block).
The

update
 method is responsible for reading and decoding frames from the video file, along with maintaining the actual queue data structure:

Faster video file FPS with cv2.VideoCapture and OpenCV

```
def update(self):
# keep looping infinitely
while True:
# if the thread indicator variable is set, stop the
# thread
if self.stopped:
return
# otherwise, ensure the queue has room in it
if not self.Q.full():
# read the next frame from the file
(grabbed, frame) = self.stream.read()
# if the `grabbed` boolean is `False`, then we have
# reached the end of the video file
if not grabbed:
self.stop()
return
# add the frame to the queue
self.Q.put(frame)
```

On the surface, this code is *very similar* to our example in the *slow, naive* method detailed above.

The key takeaway here is that this code is actually running in a *separate thread* — this is where our actual FPS processing rate increase comes from.

On **Line 34** we start looping over the frames in the video file.

If the

stopped
 indicator is set, we exit the thread (**Lines 37 and 38**).
If our queue is *not full* we read the next frame from the video stream, check to see if we have reached the end of the video file, and then update the queue (**Lines 41-52**).

The

read
 method will handle returning the next frame in the queue:
Faster video file FPS with cv2.VideoCapture and OpenCV

```
def read(self):
# return next frame in the queue
return self.Q.get()
```

We'll create a convenience function named

more
 that will return
True
 if there are still more frames in the queue (and
False
 otherwise):
Faster video file FPS with cv2.VideoCapture and OpenCV

```
def more(self):
# return True if there are still frames in the queue
return self.Q.qsize() > 0
```

And finally, the

stop
 method will be called if we want to stop the thread prematurely (i.e., before we have reached the end of the video file):
Faster video file FPS with cv2.VideoCapture and OpenCV

```
def stop(self):
# indicate that the thread should be stopped
self.stopped = True
```

The *faster, threaded* method to reading video frames with OpenCV

Now that we have defined our

FileVideoStream
 class we can put all the pieces together and enjoy a faster, threaded video file read with OpenCV.
Open up a new file, name it

read_frames_fast.py
 , and insert the following code:

Faster video file FPS with cv2.VideoCapture and OpenCV

```
# import the necessary packages
from imutils.video import FileVideoStream
from imutils.video import FPS
import numpy as np
import argparse
import imutils
import time
import cv2
# construct the argument parse and parse the arguments
ap = argparse.ArgumentParser()
ap.add_argument("-v", "--video", required=True,
help="path to input video file")
args = vars(ap.parse_args())
# start the file video stream thread and allow the buffer to
# start to fill
print("[INFO] starting video file thread...")
fvs = FileVideoStream(args["video"]).start()
time.sleep(1.0)
# start the FPS timer
fps = FPS().start()
```

Lines 2-8 import our required Python packages. Notice how we are using the

FileVideoStream
 class from the
imutils
 library to facilitate faster frame reads with OpenCV.

Lines 11-14 parse our command line arguments. Just like the previous example, we only need a single switch,

--video
, the path to our input video file.

We then instantiate the

FileVideoStream
 object and start the frame reading thread (**Line 19**).

Line 23 then starts the FPS timer.

Our next section handles reading frames from the

FileVideoStream
, processing them, and displaying them to our screen:

Faster video file FPS with cv2.VideoCapture and OpenCV

```
# loop over frames from the video file stream
while fvs.more():
# grab the frame from the threaded video file stream, resize
# it, and convert it to grayscale (while still retaining 3
# channels)
frame = fvs.read()
frame = imutils.resize(frame, width=450)
frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
frame = np.dstack([frame, frame, frame])
# display the size of the queue on the frame
cv2.putText(frame, "Queue Size: {}".format(fvs.Q.qsize()),
(10, 30), cv2.FONT_HERSHEY_SIMPLEX, 0.6, (0, 255, 0), 2)
# show the frame and update the FPS counter
cv2.imshow("Frame", frame)
cv2.waitKey(1)
fps.update()
```

We start a

while
 loop on **Line 26** that will keep grabbing frames from the
FileVideoStream
 queue until the queue is empty.
For each of these frames we'll apply the same image processing operations, including: resizing, conversion to grayscale, and displaying text on the frame (in this case, our text will be the number of frames in the queue).

The processed frame is displayed to our screen on **Lines 40-42**.

The last code block computes our FPS throughput rate and performs a bit of cleanup:

Faster video file FPS with cv2.VideoCapture and OpenCV

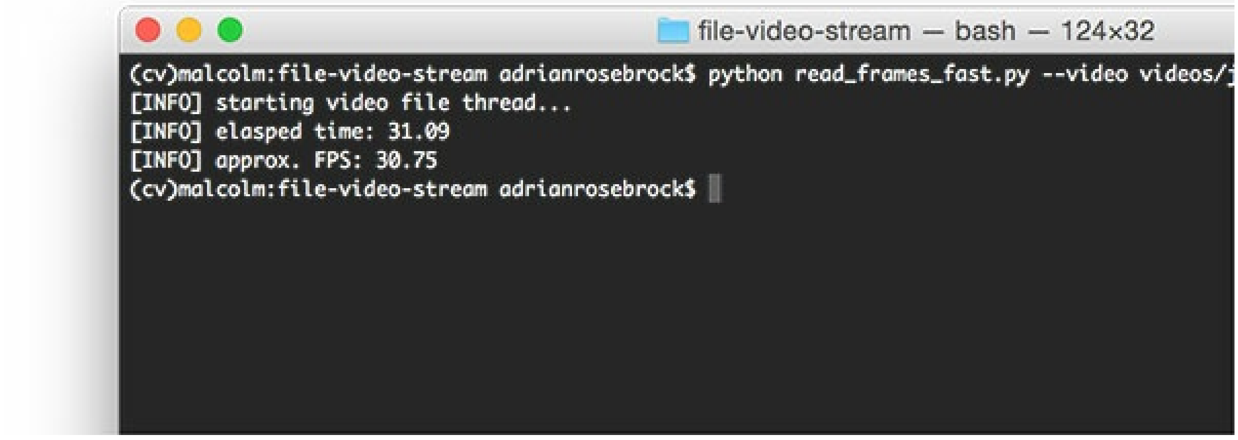
```
# stop the timer and display FPS information
fps.stop()
print("[INFO] elasped time: {:.2f}".format(fps.elapsed()))
print("[INFO] approx. FPS: {:.2f}".format(fps.fps()))
# do a bit of cleanup
cv2.destroyAllWindows()
fvs.stop()
```

To see the results of the

read_frames_fast.py
 script, make sure you download the source code + example video using the ***"Downloads"*** section at the bottom of this tutorial.
From there, execute the following command:

Faster video file FPS with cv2.VideoCapture and OpenCV

```
$ python read_frames_fast.py --video videos/jurassic_park_intro.mp4
```



```
(cv)malcolm:file-video-stream adrianrosebrock$ python read_frames_fast.py --video videos/j
[INFO] starting video file thread...
[INFO] elasped time: 31.09
[INFO] approx. FPS: 30.75
(cv)malcolm:file-video-stream adrianrosebrock$ 
```

Figure 3: Utilizing threading with *cv2.VideoCapture* and OpenCV leads to higher FPS and a larger throughput rate.

As we can see from the results we were able to process the entire 31 second video clip in ***31.09 seconds* — that's an improvement of *34%* from the slow, naive method!**

The actual frame throughput processing rate is much faster, **clocking in at *30.75 frames per second*, an improvement of *52.15%*.**

Threading can dramatically improve the speed of your video processing pipeline — use it whenever you can.

What about built-in webcams, USB cameras, and the Raspberry Pi? What do I do then?

This post has focused on using threading to improve the frame processing rate of *video files*.

If you're instead interested in speeding up the FPS of your built-in webcam, USB camera, or Raspberry Pi camera module, please refer to these blog posts:

Summary

In today's tutorial I demonstrated how to use threading and a queue data structure to improve the FPS throughput rate of your video processing pipeline.

By placing the call to

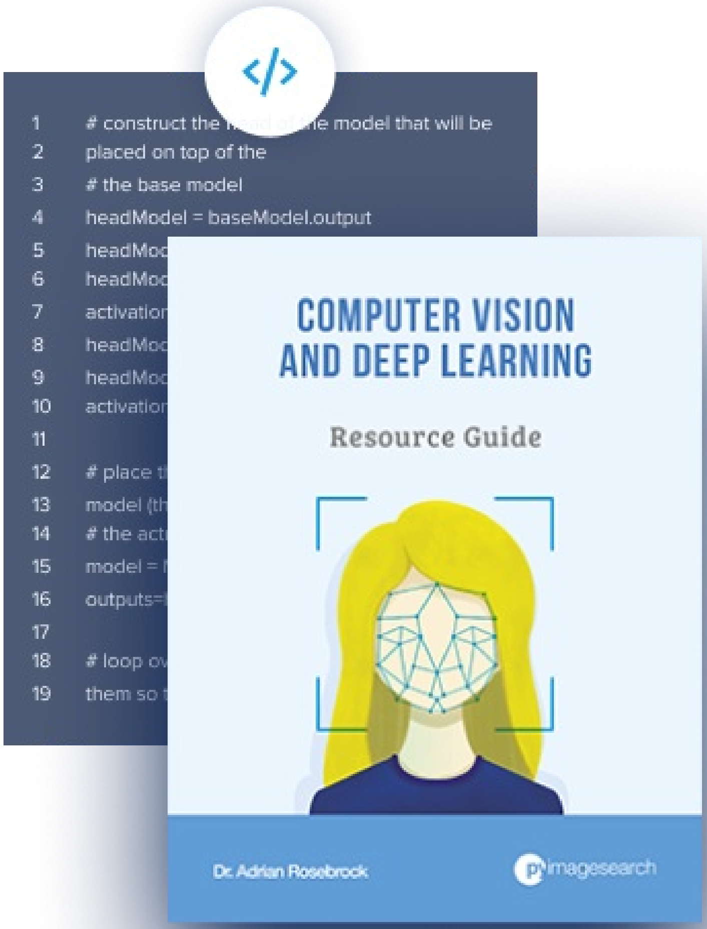
.read
 of a
cv2.VideoCapture
 object in a thread *separate* from the main Python script we can avoid blocking I/O operations that would otherwise dramatically slow down our pipeline.

Finally, I provided an example comparing *threading* with *no threading*. **The results show that by using threading we can improve our processing pipeline by up to 52%.**

However, keep in mind that the more steps (i.e., function calls) you make inside your

while

loop, the more computation needs to be done — therefore, your actual frames per second rate will drop, but you'll still be processing faster than the non-threaded version.
To be notified when future blog posts are published, be sure to enter your email address in the form below!



Download the Source Code and FREE 17-page Resource Guide

Enter your email address below to get a .zip of the code and a **FREE 17-page Resource Guide on Computer Vision, OpenCV, and Deep Learning.** Inside you'll find my hand-picked tutorials, books, courses, and libraries to help you master CV and DL!